conference
proceedings

# 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)

*Hollywood, CA, USA*
*October 8–10, 2012*

Sponsored by
usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
in cooperation with ACM SIGOPS

usenix ASSOCIATION

Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation *Hollywood, CA, USA October 8–10, 2012*

ISBN 978-1-931971-96-6

USENIX Association

# Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)

October 8–10, 2012
Hollywood, CA

# Conference Organizers

## Program Co-Chairs

Chandu Thekkath, *Microsoft Research Silicon Valley*
Amin Vahdat, *Google and University of California, San Diego*

## Program Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Mike Dahlin, *University of Texas, Austin*
Jason Flinn, *University of Michigan*
Steve Gribble, *University of Washington*
Tim Harris, *Oracle Labs*
Jon Howell, *Microsoft Research Redmond*
Dejan Kostić, *École Polytechnique Fédérale de Lausanne (EPFL)*
Jinyang Li, *New York University*
Shan Lu, *University of Wisconsin—Madison*
Petros Maniatis, *Intel Labs*
Jeff Mogul, *HP Labs*
Robert Morris, *Massachusetts Institute of Technology*
Florentina Popovici, *Google*
Timothy Roscoe, *ETH Zurich*
Stefan Savage, *University of California, San Diego*
Emin Gün Sirer, *Cornell University*
Ion Stoica, *University of California, Berkeley*
John Wilkes, *Google*
Junfeng Yang, *Columbia University*
Yuan Yu, *Microsoft Research Silicon Valley*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Lidong Zhou, *Microsoft Research Asia*

## Poster Session Co-Chairs

Shan Lu, *University of Wisconsin—Madison*
Junfeng Yang, *Columbia University*

## Steering Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Brad Chen, *Google*
Brian Noble, *University of Michigan*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*

# External Review Committee

Atul Adya, *Google*
Katerina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*
Ranjita Bhagavan, *Microsoft Research*
Frank Dabek, *Google*
Rodrigo Fonseca, *Brown University*
Roxana Geambasu, *Columbia University*
Krishna Gummadi, *Max Planck Institute for Software Systems (MPI-SWS)*
Chip Killian, *Purdue University*
Rama Kotla, *Microsoft Research*
Harsha Madhyastha, *University of California, Riverside*
J.P. Martin, *Microsoft Research*
George Porter, *University of California, San Diego*
Michael Walfish, *University of Texas, Austin*

# 10th USENIX Symposium on Operating Systems Design and Implementation
# October 8–10, 2012
# Hollywood, CA, USA



## Monday, October 8

**Big Data**



**Privacy**



**Mobility**



## Tuesday, October 9

**Distributed Systems and Networking**



**(Tuesday, October 9, continues on p. iv)**

## Wednesday, October 10

# Message from the USENIX OSDI '12 Program Co-Chairs

We are delighted to welcome you to the 10th USENIX Symposium on Operating Systems Design and Implementation. In recent years, OSDI has rightly come to be regarded as the strongest conference for systems research held in even years, and this year's program continues that trend. The program contains 25 papers representing some of the best research done in academia and industry.

This year's program was selected in three rounds of double-blind reviews from 215 submissions, followed by a day and a half Program Committee meeting. The Program Committee of 25 did the bulk of the reviews, relying on a 13-member External Review Committee for a set of first-round reviews. Each paper discussed in the final round at the PC meeting had nine reviews, written primarily by PC members, each of whom wrote at least 40 reviews during the course of about two months. The resulting discussions were collegial and the final papers were selected through broad and well informed consensus. No paper was accepted or rejected with less than a strong majority of the PC in agreement. While the workload for the PC was substantial, we benefitted from discussions involving typically half of the PC for each paper. We also benefitted from allotting an extra half-day to the PC meeting, allowing decisions to be made in a deliberate manner.

In addition to the accepted papers, we also have a keynote by David Haussler from UC Santa Cruz, and we would like to record our thanks to him. We also have two poster sessions where you will see works-in-progress as well as posters representing accepted papers.

We would like to thank the many authors who submitted papers and posters to OSDI; it was a privilege to get a preview of some of the current research in our field. We also would like to thank the External Review Committee members for their work, which was essential given the number of submissions to OSDI. Finally, we would like to express our thanks to the PC members for the tremendous amount of work that went into the reviews; this was exemplary service on their part for our community.

**Amin Vahdat,** ***Google and University of California, San Diego***
**Chandu Thekkath,** ***Microsoft Research Silicon Valley***
**OSDI '12 Program Co-Chairs**

# Flat Datacenter Storage

Edmund B. Nightingale, Jeremy Elson,
Jinliang Fan, Owen Hofmann,* Jon Howell, and Yutaka Suzue

Microsoft Research University of Texas at Austin*

## Abstract

Flat Datacenter Storage (FDS) is a high-performance, fault-tolerant, large-scale, locality-oblivious blob store. Using a novel combination of full bisection bandwidth networks, data and metadata striping, and flow control, FDS multiplexes an application's large-scale I/O across the available throughput and latency budget of *every* disk in a cluster. FDS therefore makes many optimizations around data locality unnecessary. Disks also communicate with each other at their full bandwidth, making recovery from disk failures extremely fast. FDS is designed for datacenter scale, fully distributing metadata operations that might otherwise become a bottleneck.

FDS applications achieve single-process read and write performance of more than 2 GB/s. We measure recovery of 92 GB data lost to disk failure in 6.2 s and recovery from a total machine failure with 655 GB of data in 33.7 s. Application performance is also high: we describe our FDS-based sort application which set the 2012 world record for disk-to-disk sorting.

## 1 Introduction

A shared and centralized model of storage is one of simplicity. Consider a centralized file server in a small computer science department. Data stored by any computer can be retrieved by any other. This conceptual simplicity makes it easy to use: computation can happen on any computer, even in parallel, without regard to first putting data in the right place. As a result, applications are much less complex than they would be without a shared filesystem.

At the scale of large, big-data clusters that routinely exceed thousands of computers, this "flat" model of storage is still highly desirable. While some blob storage systems such as Amazon S3 [10] provide one, they come with a significant performance penalty because networks at datacenter scales have historically been *oversubscribed*. Individual machines were typically attached in a tree topology [12]; for cost efficiency, links near the root had significantly less capacity than the aggregate capacity below them. Core oversubscription ratios of hundreds to one were common, which meant communication was fast within a rack, slow off-rack, and worse still for nodes whose nearest ancestor was the root.

The datacenter bandwidth shortage has had unfortunate and far-reaching consequences in systems where performance is paramount. Software developers accustomed to treating the network as an abstraction are forced to think in terms of "rack locality." New programming models (e.g., MapReduce [13], Hadoop [1], and Dryad [19]) emerged to help exploit locality, preferentially moving computation to data rather than vice-versa. They effectively expose a cluster's aggregate disk throughput for tasks with high reduction factors (searching for a rare string), but many important computations (sort, distributed join, matrix operations) fundamentally require data movement and are still not well served by systems whose performance is locality-dependent. Software must also be expressed in a data-parallel style, which is unnatural for many tasks.

Counterintuitively, locality constraints can sometimes even *hinder* efficient resource utilization. One example is stragglers: if data is singly replicated, a single unexpectedly slow machine can preclude an entire job's timely completion even while most of the resources are idle. The preference for local disks also serves as a barrier to quickly retasking nodes: since a CPU is only useful for processing the data resident there, retasking requires expensive data movement. If the resident data is not needed, the CPUs may not be usable. In addition, because a node has a fixed ratio of CPUs to disks, tasks that run there will nearly always leave either CPUs or disks partially idle, depending on the resource ratio required by the task.

The root of this cascade of consequences was the locality constraint, itself rooted in the datacenter bandwidth shortage. When bandwidth was scarce, these sacrifices were necessary to achieve the best performance. However, recently developed CLOS networks [16, 15, 24]—large numbers of small commodity switches with redundant interconnections—have made it economical to build non-oversubscribed full bisection bandwidth networks at the scale of a datacenter for the first time. Flat Datacenter Storage (FDS) is a datacenter storage system designed from first principles under the formerly unrealistic assumption that datacenter bandwidth is abundant.

*Work completed during a Microsoft Research internship.

Unconventionally for a system of this scale, FDS returns to the flat storage model: all compute nodes can access all storage with equal throughput. Even when computation is co-located with storage, all storage is treated as remote; in FDS, there are no "local" disks. By spreading data over disks uniformly at a relatively fine grain (§2.2), FDS statistically multiplexes workloads over all of the disks in a cluster. FDS effectively eliminates the need to imbue locality constraints into storage systems, schedulers, or programming models.

Despite the conceptual simplicity afforded by the flat storage model, FDS achieves cluster-wide I/O performance on par with systems that exploit locality. Single-process read and write performance exceeds 2 GB/s (§5.2), and FDS dramatically accelerates data movement workloads. For example, our sorting application beat a world record for disk-to-disk sort performance (§6.1) by a factor of 2.8 while using about 1/5 as many disks. It is the first system in the competition's history to do so without exploiting locality.

A consequence of our design is that disk-to-disk bandwidth is also extremely high, facilitating fast recovery from disk and machine failures. In our 1,000 disk cluster, FDS recovers 92 GB lost from a failed disk in 6.2 seconds. Recovery of 655 GB lost from a failed 7-disk machine takes 33.7 seconds (§5.3).

FDS is more efficient for many workloads because every job can use the cluster's I/O bandwidth and CPU resources in exactly the ratio required. FDS therefore moves away from conflating high performance with data-parallel programming. Further, since data locality is immaterial to compute nodes, they are easily and quickly retasked. This can even be done at a fine grain within a single task; as demonstrated in §2.4, *dynamic work allocation* retasks at the granularity of individual data reads to dramatically reduce the effect of stragglers.

Much past research has been directed towards solving the individual problems brought about by the need for locality. FDS, in contrast, is a clean redesign from which many of these solutions fall out naturally. Our goal is to move datacenters back to a flat storage model so that these benefits may be widely realized.

The remainder of this paper is organized as follows. Section 2 gives an overview of the design of FDS. Section 3 describes FDS' strategy for replication and failure recovery. Section 4 explores our network in greater detail and describes our novel congestion avoidance strategy needed in full bisection bandwidth networks. Section 5 presents microbenchmarks. Section 6 reviews how FDS can speed up real workloads, including sort and serving a web index. We review related work in Section 7 and conclude in Section 8.

## 2 Design Overview

FDS' main goal is to expose all of a cluster's disk bandwidth to applications. Blobs are divided into *tracts* (§2.1), parallelizing both storage of data (§2.2) and handling of metadata (§2.3) across all disks. We exploit our locality-oblivious storage to dynamically assign work to workers, preventing stragglers (§2.4). FDS provides the best of both worlds: a scale-out system with aggregate I/O throughput equal to systems that exploit local disks combined with the conceptual simplicity and flexibility of a logically centralized storage array.

For simplicity we will first describe the system without regard to fault tolerance. §3 will describe replication, failure recovery, and cluster growth. In addition, this section assumes the network core is never congested; our network is described in §4.

### 2.1 Blobs and Tracts

In FDS, data is logically stored in *blobs*. A blob is a byte sequence named with a 128-bit GUID. The GUID can either be selected by the application or assigned randomly by the system. Blobs can be any length up to the system's storage capacity. Reads from and writes to a blob are done in units called *tracts*. Each tract within a blob is numbered sequentially starting from 0. Blobs and tracts are mutable; nothing prevents a client from overwriting a previously-written tract with new data.

Tracts are sized such that random and sequential access achieves nearly the same throughput. In our cluster, tracts are 8 MB (§5.1). The tract size is set when the cluster is created based upon cluster hardware. For example, if flash were used instead of disks, the tract size could be made far smaller (e.g., 64 kB).

Every disk is managed by a process called a *tractserver* that services read and write requests that arrive over the network from clients. Tractservers do not use a file system. Instead, they lay out tracts directly to disk by using the raw disk interface. Since there are only about $10^6$ tracts per disk (for a 1 TB disk), all tracts' metadata is cached in memory, eliminating many disk accesses.

Tractservers and their network protocol are not exposed directly to FDS applications. Instead, these details are hidden in a client library with a narrow and straightforward interface. Figure 1 shows a simplified version of it; some parameters and return values have been elided. In addition to the listed parameters, each function takes a callback function and an associated context pointer. All calls in FDS are non-blocking; the library invokes the application's callback when the operation completes.

The application's callback functions must be reentrant; they are called from the library's threadpool and may overlap. Tract reads are not guaranteed to arrive in order of issue. Writes are not guaranteed to be committed in order of issue. Applications with ordering require-

| Getting access to a blob |
|---|
| `CreateBlob(UINT128 blobGuid)` |
| `OpenBlob(UINT128 blobGuid)` |
| `CloseBlob(UINT128 blobGuid)` |
| `DeleteBlob(UINT128 blobGuid)` |
| **Interacting with a blob** |
| `GetBlobSize()` |
| `ExtendBlobSize(UINT64 numberOfTracts)` |
| `WriteTract(UINT64 tractNumber, BYTE *buf)` |
| `ReadTract(UINT64 tractNumber, BYTE *buf)` |
| `GetSimultaneousLimit()` |

Figure 1: FDS API

ments are responsible for issuing operations after previous acknowledgments have been received, rather than concurrently. FDS guarantees atomicity: a write is either committed or failed completely.

The non-blocking API helps applications achieve good performance. By spreading a blob's tracts over many tractservers (§2.2) and issuing many requests in parallel, many tractservers can begin reading tracts off disk and transferring them back to the client simultaneously. In addition, deep read-ahead allows a tract to be read off disk into the tractserver's cache while the previous one is transferred over the network. The FDS API `GetSimultaneousLimit()` tells the application how many reads and writes to issue concurrently. A typical simultaneous limit is 50 tracts, though the exact value depends on the client's bandwidth.

## 2.2 Deterministic data placement

A key issue in parallel storage systems is data placement and rendezvous, that is: how does a writer know where to send data? How does a reader find data that has been previously written?

Many systems solve this problem using a metadata server that stores the location of data blocks [14, 30]. Writers contact the metadata server to find out where to write a new block; the metadata server picks a data server, durably stores that decision and returns it to the writer. Readers contact the metadata server to find out which servers store the extent to be read. This method has the advantage of allowing maximum flexibility of data placement and visibility into the system's state. However, it has drawbacks: the metadata server is a central point of failure, usually implemented as a replicated state machine, that is on the critical path for all reads and writes.

In FDS, we took a different approach. FDS uses a metadata server, but its role during normal operations is simple and limited: collect a list of the system's active tractservers and distribute it to clients. We call this list the *tract locator table*, or TLT. In a single-replicated system, each TLT entry contains the address of a single tractserver. With $k$-way replication, each entry has $k$ tractservers; see §3.3.

To read or write tract number $i$ from a blob with GUID $g$, a client first selects an entry in the TLT by computing an index into it called the *tract locator*, designed to both be deterministic and produce uniform disk utilization:

$$\text{Tract\_Locator} = (\text{Hash}(g) + i) \bmod \text{TLT\_Length}$$

Hashing the GUID "randomizes" each blob's starting point in the table, ensuring clients better exploit the available parallelism whether or not the GUIDs themselves are assigned randomly. FDS uses SHA-1 for this hash.

Adding the tract number *outside* the hash ensures that large blobs use all entries in the TLT uniformly. An early (discarded) locator equation used $\text{Hash}(g+i)$. This effectively selected a TLT entry independently at random for each tract, producing a binomial rather than a uniform distribution of tracts on tractservers. As the number of tractservers increased, so did the occupancy deviation between the most-filled and least-filled disk. In the rejected design, writing 1 TB of 8 MB tracts to 1,000 tractservers was expected to write between 92 and 161 tracts to each ($\mu = 125$; $\sigma = 11.2$). The one tractserver with 29% more data than average was an unwanted straggler.

Once clients find the proper tractserver address in the TLT, they send read and write requests containing the blob GUID, tract number, and (in the case of writes) the data payload. Readers and writers rendezvous because tractserver lookup is deterministic: as long as a reader has the same TLT the writer had when it wrote a tract, a reader's TLT lookup will point to the same tractserver.

In a single-replicated system, the TLT is constructed by concatenating $m$ random permutations of the tractserver list. Using only a single permutation can lead to unwanted client convoys. Sequential reads from a blob use TLT entries sequentially, so clients that bunch up in the queue of a slow tractserver will move in lockstep through the TLT, overloading some tractservers while many others are idle. Setting $m > 1$ ensures that after being delayed in a slow queue, clients will fan out to $m$ other tractservers for their next operation. Our system uses $m = 20$, but we have not tested the system's sensitivity to this parameter.

In the case of non-uniform disk speeds, the TLT is weighted so that different tractservers appear in proportion to the measured speed of the disk.

To be clear, the TLT *does not* contain complete information about the location of individual tracts in the system. It is not modified by tract reads and writes. The only way to determine if a tract exists is to contact the tractserver that *would be* responsible for the tract if it does exist. Since the TLT changes only in response to cluster reconfiguration or failures it can be cached by clients

for a long time. Its size in a single-replicated system is proportional to the number of tractservers in the system (hundreds, or thousands), not the number of tracts stored (millions or billions).

When the system is initialized, tractservers locally store their position in the TLT. This means the metadata server does not need to store durable state, simplifying its implementation. In case of a metadata server failure, the TLT is reconstructed by collecting the table assignments from each tractserver.

To summarize, our metadata scheme has a number of nice properties:

- The metadata server is in the critical path only when a client process starts. This is the key factor that allows us to practically keep tract sizes arbitrarily small. Systems such as GFS [14] require larger chunks partially to reduce load on the metadata server.
- The TLT can be cached long-term since it changes only on cluster configuration, not each read and write, eliminating all traffic to the metadata server in a running system under normal conditions.
- The metadata server stores metadata only about the hardware configuration, not about blobs. Since traffic to it is low, its implementation is simple and lightweight.
- Since the TLT contains random permutations of the list of tractservers, sequential reads and writes by independent clients are highly likely to utilize all tractservers uniformly and are unlikely to organize into synchronized convoys.

Our design is enabled by running on a full bisection bandwidth network. The locality-oblivious uniform access pattern would cause crippling congestion on a traditional network with hierarchical oversubscription.

## 2.3 Per-Blob Metadata

Each blob has metadata such as its length. FDS stores it in each blob's special metadata tract ("tract $-1$"). Clients find a blob's metadata on a tractserver using the same TLT used to find regular data. Distributed metadata is a particular advantage for atomic blob operations that require serialization to avoid inconsistency (e.g. `CreateBlob`, `DeleteBlob` and `ExtendBlobSize`). Even if thousands of clients are requesting atomic operations on blobs simultaneously, operations that can be parallelized (by virtue of referring to different blobs) are likely serviced in parallel by independent tractservers.

When a blob is created, the tractserver responsible for its metadata tract creates that tract on disk and initializes the blob's size to 0. When a blob is deleted, that tractserver deletes the metadata. A scrubber application scans each tractserver for orphaned tracts with no associated metadata, making them eligible for garbage collection.

Newly created blobs have a length of 0 tracts. Applications must *extend* a blob before writing past the end of it. The extend operation is atomic, is safe to execute concurrently with other clients, and returns the new size of the blob as a result of the client's call. A separate API tells the client the blob's current size. Extend operations for a blob are sent to the tractserver that owns that blob's metadata tract. The tractserver serializes it, atomically updates the metadata, and returns the new size to each caller. If all writers follow this pattern, the extend operation provides a range of tracts the caller may write without risk of conflict. Therefore, the extend API is functionally equivalent to the Google File System's "atomic append." Space is allocated lazily on tractservers, so tracts claimed but not used do not waste storage.

## 2.4 Dynamic Work Allocation

A result that flows naturally from FDS is that the assignment of work to worker can be done at very short timescales. This enables FDS to mitigate stragglers—a significant bottleneck in large systems because a task is not complete until its slowest worker is complete [5].

Hadoop- and MapReduce-style clusters that primarily process data locally are very sensitive to machines that are slow due to factors such as misbehaving hardware, jobs running concurrently, hotspots in the network, and non-uniformity in the input. If a node falls behind, there are not many options for recovery other than restarting its computation elsewhere [5]. The straggler period can also represent a great loss in efficiency if most resources are idle while waiting for a slow task to complete.

In FDS, since storage and compute are no longer colocated, the assignment of work to worker can be done dynamically, at fine granularity, *during* task execution. The best practice for FDS applications is to centrally (or, at large scale, hierarchically) give small units of work to each worker as it nears completion of its previous unit. This self-clocking system ensures that the maximum dispersion in completion times across the cluster is only the time required for the slowest worker to complete a single unit. Such a scheme is not practical in systems where the assignment of work to workers is fixed in advance by the requirement that data be resident at a particular worker before the job begins.

In many applications, the effect is significant. For example, in our sort application (§6.1), elimination of stragglers in the reading phase accounted for a 1/3 reduction in total job runtime.

# 3 Replication and Failure Recovery

Thus far, we have described FDS as single-replicated and thus not resilient to disk failures. To improve durability and availability, FDS supports higher levels of replication. When a disk fails, redundant copies of the lost data are used to restore the data to full replication.

The use of full bisection bandwidth networks means that, with appropriate data layout (§3.3), FDS can perform failure recovery dramatically faster than many other systems. In an $n$-disk cluster where one disk fails, roughly $1/n$th of the replicated data will be found on all $n$ of the other disks. All remaining disks send the under-replicated data to each other *in parallel*, restoring the cluster to full replication very quickly. Performance is bounded only by the aggregate disk and network bandwidth. Thus, as the size of the cluster grows failure recovery gets *faster*. As we will see in §5.3, our cluster of about 1,000 disks recovers 92 GB of data lost from a single disk in only 6.2 s, and 655 GB lost from 7 disks on a failed machine in 33.7 s. Though the broad approach of FDS' failure recovery is similar to RAMCloud [26], RAMCloud recovers data to DRAM and uses replication only for fault tolerance. FDS uses replication both for availability and fault tolerance while recovering data *back to stable storage*. Such fast failure recovery significantly improves durability because it reduces the window of vulnerability during which additional failures can cause unrecoverable data loss.

## 3.1 Replication

As described in §2.2, each entry of the TLT in an $n$-way replicated cluster contains $n$ tractservers. (Construction of such a TLT is described in §3.3.) When an application writes a tract, the client library finds the appropriate row of the TLT and sends the write to *every* tractserver it contains. Reads select a single tractserver at random. Applications are notified that their writes have completed only after the client library receives write acknowledgments from all replicas.

Replication also requires changes to `CreateBlob`, `ExtendBlobSize`, and `DeleteBlob`. Each mutates the state of the metadata tract and must guarantee that updates are serialized. Clients send these operations only to the tractserver acting as the primary replica, marked as such in the TLT. When a tractserver receives one of these operations, it executes a two-phase commit with the other replicas. The primary replica does not commit the change until all other replicas have completed successfully. Should the prepare fail, the operation is aborted.

FDS also supports per-blob *variable replication*, for example, to single-replicate intermediate computations for write performance, triple-replicate archival data for durability, and over-replicate popular blobs to increase read bandwidth. The maximum possible replication level is determined when the cluster is created and drives the number of tractservers listed in each TLT entry. Each blob's actual replication level is stored in the blob's metadata tract and retrieved when a blob is opened. For an $n$-way replicated blob, the client uses only the first $n$ tractservers in each TLT entry.

## 3.2 Failure recovery

We begin with the simplest failure recovery case: the failure of a single tractserver. Later sections will describe concurrent tractserver failures, support for failure domains, and metadata server failures.

As described earlier, each row of the TLT lists several tractservers. However, each row also has a *version number*, canonically assigned by the metadata server. When a tractserver is assigned to a row and column of the TLT, it is also given the row's current version number.

Tractservers send heartbeat messages to the metadata server. When the metadata server detects a tractserver timeout, it declares the tractserver dead. Then, it:

- invalidates the current TLT by incrementing the version number of each row in which the failed tractserver appears;
- picks random tractservers to fill in the empty spaces in the TLT where the dead tractserver appeared;
- sends updated TLT assignments to every server affected by the changes; and
- waits for each tractserver to ack the new TLT assignments, and then begins to give out the new TLT to clients when queried for it.

When a tractserver receives an assignment of a new entry in the TLT, it contacts the other replicas and begins copying previously written tracts. When a failure occurs, clients must wait only for the TLT to be updated; operations can continue while re-replication is still in progress.

All operations are tagged with the client's TLT version. If a client attempts an operation using a stale TLT entry, the tractserver detects the inconsistency and rejects the operation. This prompts the client to retrieve an updated TLT from the metadata server. Client operations to tractservers not affected by the failure proceed as usual.

Table versioning prevents a tractserver that failed and then returned, e.g., due to a transient network outage, from continuing to interact with applications as soon as the client receives a new TLT. Further, any attempt by a failed tractserver to contact the metadata server will result in the metadata server ordering its termination.

After a tractserver failure, the TLT immediately converges to provide applications the current location to read or write data. This convergence property differentiates it from other hash-based approaches, such as those used within distributed hash tables, which may cause requests to be routed multiple times through the network before determining an up-to-date location for data.

### 3.2.1 Additional failure scenarios

Thus far, we have considered only the case where a single tract server fails. We now extend our description to concurrent and cascading tractserver failures as well as metadata server failures.

| Row | Version | Replica 1 | Replica 2 | Replica 3 |
|---|---|---|---|---|
| 1 | 8 | A | F | B |
| 2 | 17 | B | C | L |
| 3 | 324 | E | D | G |
| 4 | 3 | T | A | H |
| 5 | 456 | F | B | G |
| 6 | 723 | G | E | B |
| 7 | 235 | D | V | C |
| 8 | 312 | H | E | F |

| Row | Version | Replica 1 | Replica 2 | Replica 3 |
|---|---|---|---|---|
| 1 | 9 | A | F | H |
| 2 | 18 | I | C | L |
| 3 | 324 | E | D | G |
| 4 | 3 | T | A | H |
| 5 | 457 | F | C | G |
| 6 | 724 | G | E | A |
| 7 | 235 | D | V | C |
| 8 | 312 | H | E | F |

Figure 2: A Tract Locator Table before (*left*) and after (*right*) Disk B fails. B's appearances in the table are replaced with different disks and the version numbers of affected rows are incremented. All the disks in rows that contained B participate in failure recovery in parallel.

When multiple tractservers fail, the metadata server's only new task is to fill more slots in the TLT. Similarly, if failure recovery is underway and additional tractservers fail, the metadata server executes another round of the protocol by filling in empty table entries and incrementing their version. Data loss occurs when all the tractservers within a row fail within the recovery window.

Though not yet implemented, transient failures can be handled gracefully with *partial failure recovery*. A tractserver failure triggers replication of lost data as usual. If the tractserver later returns to service, the metadata server has two options: complete failure recovery as if the disk had never returned or use other replicas to recover the writes the returning tractserver missed while it was away. The metadata server will choose the option that requires copying less data. If it completes failure recovery, the returning tractserver is added to the empty disk pool. Otherwise, the tractserver resumes its positions in the TLT, failure recovery is halted, and copying of missed writes begins. Tractservers identify missed writes by examining the version of the TLT with which each tract was written. To further mitigate the effects of transient faults, the metadata server could separate the replacement of a failed server in the TLT from the initiation of the data movement required for failure recovery.

Network partitions complicate recovery from metadata server failures. A simple primary/backup scheme is not safe because two active metadata servers will cause cluster corruption. Our current solution is simple: only one metadata server is allowed to execute at a time. If it fails, an operator must ensure the old one is decommissioned and start a new one. We are experimenting with using Paxos leader election to safely make this process automatic and reduce the impact to availability. Once a new metadata server starts, tractservers contact it with their TLT assignments. After a timeout period, the new metadata server reconstructs the old TLT.

Tractservers that fail concurrently with a metadata server are detected by the new metadata server as missing TLT entries; the normal failure recovery protocol is executed. One strength our failure recovery protocol is its simplicity; all tractserver failure cases use the same protocol and exercise the same code path. This leads to small, simple, more obviously-correct code.

## 3.3 Replicated data layout

As mentioned previously, a $k$-way replicated system has $k$ tractservers (disks) listed in each TLT entry. The selection of *which* $k$ disks appear has an important impact on both durability and recovery speed.

Imagine first that we wish to double-replicate ($k = 2$) all data in a cluster with $n$ disks. A simple TLT might have $n$ rows with each row listing disks $i$ and $i+1$. While data will be double-replicated, the cluster will not meet our goal of fast failure recovery: when a disk fails, its backup data is stored on only two other disks ($i+1$ and $i-1$). Recovery time will be limited by the bandwidth of just two disks. A cluster with 1 TB disks with 100 MB/s read performance would require 90 minutes for recovery. A second failure within that time would have roughly a $2/n$ chance of losing data permanently.[1]

A better TLT has $O(n^2)$ entries. Each possible *pair* of disks (ignoring failure domains; §3.3.1) appears in an entry of the TLT. Since the generation of tract locators is pseudo-random (§2.2), any data written to a disk will have high diversity in the location of its replica. When a disk fails, replicas of $1/n$th of its data resides on the other $n$ disks in the cluster. When a disk fails, all $n$ disks can exchange data in parallel over FDS' full bisection bandwidth network. Since all disks recover in parallel, larger clusters recover from disk loss more quickly.

While such a TLT recovers from single-disk failure quickly, a second failure while recovery is in progress is

[1] More precisely, $1-(\frac{n-1}{n})^2$

guaranteed to lose data. Since all pairs of disks appear as TLT entries, any pair of failures will lose the tracts whose TLT entry contained the pair of failed disks. Replicated FDS clusters therefore have a minimum replication level of 3. Perhaps counterintuitively, no level of replication ever needs a TLT larger than $O(n^2)$. For any replication level $k > 2$, FDS starts with the "all-pairs" TLT, then expands each entry with $k-2$ additional random disks (subject to failure domain constraints).

Constructing the replicated TLT this way has several important properties. First, performance during recovery still involves every disk in the cluster since every pair of disks is still represented in the TLT.

Second, a triple disk failure within the recovery window now has only about a $2/n$ chance[1] of causing permanent data loss. To understand why, imagine two disks fail. Find the entries in the TLT that contain those two disks. We expect to find 2 such entries. There is a $1/n$ chance that a third disk failure will match the random third disk in that TLT entry.

Finally, adding more replicas decreases the probability of data loss. Consider now a 4-way replicated cluster. Each entry in the $O(n^2)$-length TLT has *two* random disks added instead of one. 3 or fewer simultaneous failures are safe; 4 simultaneous failures have a $1/n^2$ chance of losing data. Similarly, 5-way replication means that 4 or fewer failures are safe and 5 simultaneous failures have a $1/n^3$ chance of loss.

One possible disadvantage to a TLT with $O(n^2)$ entries is its size. In our 1,000-disk cluster, the in-memory TLT is about 13 MB. However, on larger clusters, quadratic growth is cumbersome: 10,000 disks would require a 600 MB TLT.

We have two (unimplemented) strategies to mitigate TLT size. First, a tractserver can manage multiple disks; this reduces $n$ by a factor of 5–10. Second, we can limit the number of disks that participate in failure recovery. An $O(n^2)$ TLT uses every disk for recovery, but 3,000 disks are expected to recover 1 TB in less than 20 s (§5.3). The marginal utility of involving more disks may be small. To build an $n$-disk cluster where $m$ disks are involved in recovery, the TLT only needs $O(\frac{n}{m} \times m^2)$ entries. For 10,000 to 100,000 disks, this also reduces table size by a factor of 5–10. Using both optimizations, a 100,000 disk cluster's TLT would be a few dozen MB.

#### 3.3.1 Failure domains

A *failure domain* is a set of machines that have a high probability of experiencing a correlated failure. Common failure domains include machines within a rack, since they will often share a single power source, or machines within a container, as they may share common cooling or power infrastructure.

FDS leaves it up to the administrator to define a failure domain policy for a cluster. Once that policy is defined, FDS adheres to that policy when constructing the tract locator table. FDS guarantees that none of the disks in a single row of the TLT share the same failure domain. This policy is also followed during failure recovery: when a disk is replaced, the new disk must be in a different failure domain than the other tractservers in that particular row.

### 3.4 Cluster growth

FDS supports the dynamic growth of a cluster through the addition of new disks and machines. For simplicity, we first consider cluster growth in the absence of failures.

Cluster growth adds both storage capacity and throughput to the system. The FDS metadata server rebalances the assignment of table entries so that both existing data and new workloads are uniformly distributed. When a tractserver is added to the cluster, TLT entries are taken away from existing tractservers and given to the new server. These assignments happen in two phases. First, the new tractserver is given the assignments but they are marked as "pending" and the TLT version for each entry is incremented. The new tractserver then begins copying data from other replicas. During this phase, clients write data to both the existing servers and the new server so that the new tractserver is kept up-to-date. Once the tractserver has completed copying old data, the metadata server 'commits' the TLT entry by incrementing its version and changing its assignment to the new tractserver. It also notifies the now replaced tractserver that it can safely garbage collect all tracts associated with that TLT entry.

If a new tractserver fails while its TLT entries are pending, the metadata server increments the TLT entry version and expunges it from the list of new tractservers. If an existing server fails, the failure recovery protocol executes. However, tractservers with pending TLT entries are not eligible to take over for failed servers as they are already busy copying data.

Variable replication complicates cluster growth because each replica will have different data depending upon the replication level of each blob written to it. Therefore, new tractservers must read from the existing tractserver whose TLT entry it is replacing. Should that server fail, the new tractserver may read from another tractserver "to the left" of its own entry, since it will have a superset of the data required. For example, the first replica in the TLT has all single-replicated tracts whereas the third replica has all triple-replicated tracts.

### 3.5 Consistency guarantees

The current protocol for replication depends upon the client to issue all writes to all replicas. This decision means that FDS provides weak consistency guarantees to clients. For example, if a client writes a tract to 1 of 3 replicas and then crashes, other clients reading dif-

ferent replicas of that tract will observe differing state. Weak consistency guarantees are not uncommon; for example, clients of the Google File System [14] must handle garbage entries in files. However, if strong consistency guarantees are desired, FDS could be modified to use chain replication [33] to provide strong consistency guarantees for all updates to individual tracts.

Tractservers may also be inconsistent during failure recovery. A tractserver recently assigned to a TLT entry will not have the same state as the entry's other replicas until data copying is complete. While in this state, tractservers reject read requests; clients use other replicas instead.

# 4 Networking

FDS' main goal is to expose all of a cluster's disk bandwidth to applications. FDS creates an uncongested path from disks to CPUs by:

- Giving each storage node *network bandwidth equal to its disk bandwidth*, preventing bottlenecks between the disk and network;
- Using a *full bisection bandwidth network*, preventing bottlenecks in the network core; and
- Giving compute nodes as much network bandwidth as they need I/O bandwidth, preventing bottlenecks at the client

Our FDS testbed uses a two-layer CLOS network [15, 16], which in its largest configuration consists of 8 "spine" routers and 14 "TORs" (Top-Of-Rack routers). Each router is a 64×10Gbps Blade G8264. Each TOR has a 40Gbps link (4 bonded 10Gbps ports) to each spine router, giving it 320Gbps total bandwidth to the spine layer. The other 320Gbps of each TOR's bandwidth attaches to NICs. In total, this provides about 5.5Tbps of bisection bandwidth for an infrastructure cost of about $250k. The routers are factory-standard, running the manufacturer's OS (BladeOS v6.8.4). We use BGP for route distribution with each TOR on its own IP subnet.

The TORs load-balance traffic to the spine using ECMP (equal-cost multipath routing), a standard router feature that selects a spine route for each TCP flow based on the hash of the TCP destination. This gives the network full bisection bandwidth without the need for global scheduling, and has an important advantage over round-robin route selection: it ensures packets within a flow are not reordered.

One drawback of ECMP is that full bisection bandwidth is not guaranteed, but only stochastically *likely* across multiple flows. Long-lived, high-bandwidth flows are known to be problematic with ECMP [3]. FDS, however, was designed to use a large number of short-lived (tract-sized) flows to a large, pseudo-random sequence of destinations. This was done in part to satisfy the stochastic requirement of ECMP.

Each computer in our cluster has a dual-port 10Gbps NIC, primarily the Intel X520. One or both of these ports are connected to a TOR depending on the server's role (compute vs. storage) and the number of data disks it has. The NICs are configured with large-send offload, receive-side scaling, and 9kB (jumbo) Ethernet frames. The TCP stack is configured with a reduced MinRTO to quickly recover from loss.

We have found it difficult to saturate a 10G NIC using a single TCP flow: a single CPU core typically cannot keep up with a 10Gbps NIC's interrupt load. Our operating system (Windows Server 2008 R2), in conjunction with the Intel NIC, uses RSS (Receive Side Scaling) to spread the interrupt load across cores. Similar to ECMP, RSS prevents in-flow packet reordering by hashing the TCP 4-tuple to select a core. As a result, multiple flows are needed to spread interrupt load. We need 5 flows per 10Gbps port to reliably saturate the NIC. This is easily satisfied in FDS because its design dictates that many flows are active simultaneously.

At 20Gbps, a zero-copy architecture is mandatory. FDS' data interfaces pass the zero-copy model all the way to the application. For clarity, Section 2.1 showed conventional one-copy versions of `WriteTract` and `ReadTract` interfaces; our applications actually use the preferred zero-copy versions. We also use buffer pools to avoid the large page fault penalty associated with frequent allocation of large buffers.

## 4.1 RTS/CTS

By design, at peak load, all FDS nodes simultaneously saturate their NICs with short, bursty flows. A disadvantage of short flows is that TCP's bandwidth allocation algorithms perform poorly. Under the high degree of fan-in seen during reads, high packet loss can occur as queues fill during bursts. The reaction of standard TCP to such losses can have a devastating effect on performance. This is sometimes called *incast* [34].

Schemes such as DCTCP [4] ameliorate incast in concert with routers' explicit congestion notification (ECN). However, because our network has full bisection bandwidth, collisions mostly occur *at the receiver*, not in the network core, providing us an opportunity to prevent them in the application layer. FDS does so with a request-to-send/clear-to-send (RTS/CTS) flow-scheduling system. Large sends are queued at the sender, and the receiver is notified with an RTS. The receiver limits the number of CTSs outstanding, thus limiting the number of senders competing for its receive bandwidth and colliding at its switch port.

RTS/CTS adds an RTT to each large message. This has the potential to reduce performance by preventing the network pipeline from ever filling. However, the FDS API encourages deep read-ahead and write-ahead, ensur-

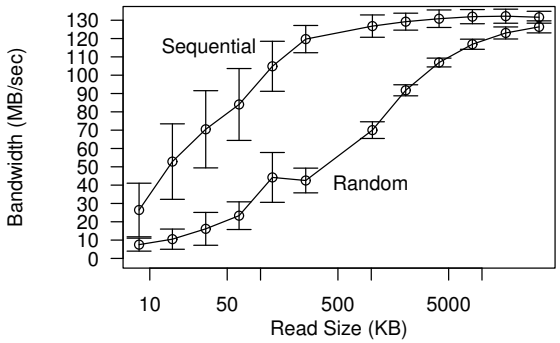



Figure 3: Performance of a single process reading to a single 10,000 RPM disk. Each point is the mean across 24 disks. Error bars show the standard deviation.

ing the FDS network library always has a large queue of future messages. This allows it to send an RTS for future messages in parallel with other data transfers, allowing the pipeline to fill.

Small and large messages are delivered using separate TCP flows, reducing the latency of control messages by enabling them to bypass long queues. FDS network message sizes are bimodal: large messages are almost all about 8 MB, and most others are a few kB or less.

# 5 Microbenchmarks

In this section and §6, we describe microbenchmarks and application benchmarks. All of our tests were conducted on a heterogeneous cluster of up to 256 HP, Dell and Silicon Mechanics servers acquired between 2008 and 2012. The machines had between 12 and 96 GB RAM, between 2 and 24 cores, and between 0 and 20 data disks (plus one OS disk). The disks were a combination of 147 GB and 300 GB 10,000 RPM 2.5" dual-port SAS drives; and 500 GB and 1 TB 7,200 RPM 3.5" SATA drives. Dynamic work allocation (§2.4) was key to efficiently using such a heterogeneous cluster. Since some machines were on loan to us during our quest to break the sort record (§6.1), some experiments used a subset of the cluster. In all but one (§6.3) of our experiments, computation and storage are located on different machines, which guarantees all experiments rely exclusively on remote access to storage. Unless otherwise noted, each test used an unreplicated cluster with CRC checks disabled. The network configuration is described in §4.

## 5.1 Raw Disk Performance

We start with a simple test: a single process writing to a single local disk. This benchmark establishes the baseline performance of our cluster's disks and shows how large random reads must be to amortize the cost of disk seeks. Like the tractserver, this benchmark uses the raw disk interface (that is, does not use NTFS). Results are shown in Figure 3. Each point is the median performance of 24 10,000 RPM SAS disks reading continuously using system calls ranging from 8KB to 32MB. Reads are sequential on the upper line, random on the lower line. Sequential performance peaks at about 131 MB/s. 8 MB random reads reach ≈117 MB/s, or ≈88% of sequential performance. Write performance, not shown, is similar to read performance.

## 5.2 Remote Reading and Writing

*SimpleTestClient* is the "hello world" FDS application. It uses the FDS client library to read and write blobs from and to tractservers. As in a real application, data are sent over the network and read from or written to disk. We tested its throughput and scalability by running successively larger numbers of SimpleTestClient instances concurrently against 1,033 tractservers. Each client instance had a single 10G NIC assigned to it. The tract size was 8 MB. For each configuration we adjusted the blob size so that the total read or write lasted between 5 and 10 minutes. We plot the number of clients against their aggregate throughput averaged over the entire experiment.

Figure 4a shows 1 to 180 clients reading and writing blobs sequentially against an unreplicated cluster. Two pairs of curves are plotted: a 516-disk and 1,033-disk cluster. At the left of the graph, the total tractserver bandwidth far exceeds the client bandwidth; performance is constrained by the number of clients. Throughput increases linearly at the rate of about 1,150 MB/s/client for writing and 950 MB/s/client for reading, roughly 90% and 74%, respectively, of the 10 Gbps interface. Reading tends to be slower than writing due to the difficulty of maintaining NIC saturation under fan-in (§4.1).

The right of the curves flatten as client bandwidth grows larger than tractserver bandwidth, which saturates them. Performance peaks at 32 GB/s in the 516-disk configuration and 67 GB/s in the 1,033-disk configuration, showing near-linear scalability. In both cases, FDS achieves remote reading and writing at roughly half the locally achievable disk throughput measured in §5.1. Though much better than many existing blob storage systems (see Table 2) there is room for improvement.

A similar test using *random* reads and writes is shown in Figure 4b. As expected, the performance is substantially the same as the sequential read and write tests. This validates our design goal that for 8MB tracts, random and sequential I/O deliver the same performance.

Figure 4c shows the bandwidth of sequentially reading and writing clients against a 1,033 disk *triple*-replicated cluster. As expected, the write bandwidth is about one-third of the read bandwidth since clients must send three copies of each write. Scaling properties are similar to that seen in Figures 4a and 4b.

Finally, to test the maximum speed achievable by a single client process, we tested a single instance of Sim-

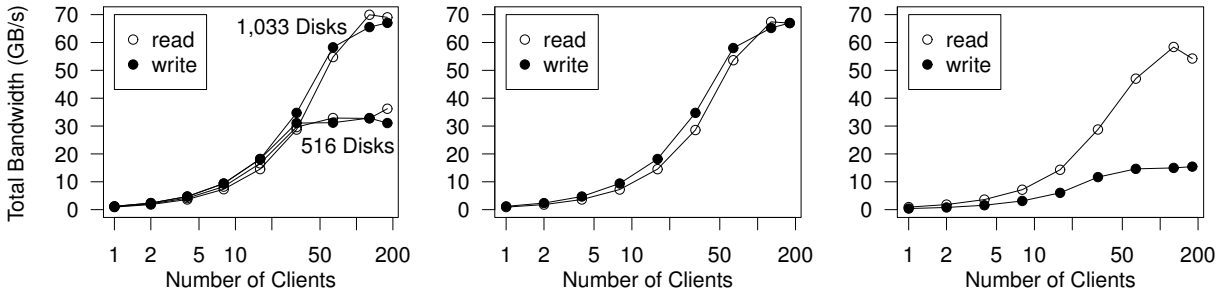



Figure 4: Mean aggregate throughput of 1 to 180 clients reading and writing 8 MB tracts on a 1,033-disk cluster. Standard deviation is less than 1% of the mean of each point. The *x* axes use logarithmic scales. *(a)* Sequential reading and writing in a single-replicated cluster. Results for a 516-disk cluster are also shown. *(b)* Random reading and writing in a single-replicated cluster. *(c)* Sequential reading and writing in a triple-replicated cluster.

| Disk count | 100 | | 1,000 | | |
|---|---|---|---|---|---|
| Disks failed | 1 | 1 | 1 | 1 | 7 |
| Total (TB) | 4.7 | 9.2 | 47 | 92 | 92 |
| GB/disk | 47 | 92 | 47 | 92 | 92 |
| GB recov. | 47 | 92 | 47 | 92 | 655 |
| Recovery time (s) | 19.2 ±0.7 | 50.5 ±16.0 | 3.3 ±0.6 | 6.2 ±0.4 | 33.7 ±1.5 |

Table 1: Mean and standard deviation of recovery time after disk failure in a triple-replicated cluster. The high variance in one experiment is due to a single 80 s run.

pleTestClient with 20 Gbps assigned to it, rather than 10 Gbps as in the previous tests. We wrote, then read, 10,000 tracts against a single-replicated cluster of 30 tractservers. Over 5 trials, SimpleTestClient achieved a mean (and standard deviation) write bandwidth of 2,187 ± 32 MB/s; for read, 2,045 ± 15 MB/s.

## 5.3 Failure Recovery

Table 1 shows the time taken to re-replicate lost data after one or more disk failures. Each experiment used a triple-replicated cluster of 100 or 1,000 146 GB, 10 K RPM SAS disks that contained enough data so that each tractserver held either 47 GB or 92 GB. We then killed a random tractserver process and measured the time until the cluster reported failure recovery was complete. Due to the random nature of the TLT, the exact amount of data recovered in each test varied slightly. We ran each test 5 times and we report the mean and standard deviation.

With 100 disks, FDS recovered 47 GB in 19.2 s and 92 GB in 50.5 s. Scaling the number of disks up by 10× to 1,000, recovery times improved by 6.8× to 3.3 s for 100 disks and by 8× to 6.2 s for 1,000 disks. Scaling is not quite linear due to the fixed cost of generating and distributing a new TLT.

We measured whole-machine failure by resetting a machine running 7 tractservers containing ≈655 GB of data. FDS recovered the lost data in 33.7 ± 1.5 s. Compared to single-disk failures, the whole-machine failure test recovered 7× the data in only 5× the time, as the fixed costs of recovery were amortized over more linear recovery time.

Recovery of 655 GB involves reading and writing a total of 1310 GB. Doing so in 33.7 s with 993 tractservers implies an average total read/write bandwidth of ≈40 MB/s/disk. Since the disks in these tests were nearly full, recovery wrote to the innermost, slowest disk tracks.

These results imply that a 1 TB disk in a 3,000 disk cluster could be recovered in ≈17 s. Such a small recovery window dramatically lessens the probability of data loss. Further, it lowers the impact of disk failures on availability and application performance. Though we are recovering to disk, our recovery time is comparable to the best known technique for recovering to RAM [26].

# 6 Applications

## 6.1 Sort

Sorting is an important primitive in many big-data applications. Its load pattern is similar to other common tasks such as distributed database joins and large matrix operations. This has made it an important benchmark since at least 1985 [9]. A group informally sponsored by SIGMOD curates an annual disk-to-disk sort performance competition with divisions for speed, cost efficiency, and energy efficiency [2]. Each has sub-divisions for general purpose applications ("Daytona") and implementations that are allowed to exploit the specifics of the competition ("Indy"), such as assuming 100-byte records and uniformly distributed 10-byte keys.

In April 2012, our FDS-based sort application set the world record for sort in both the Indy and Daytona categories of MinuteSort [7], which measures the amount of data that can be sorted in 60 seconds. Using a cluster of 1,033 disks and 256 computers (136 for tractservers, 120 for the application), our Daytona-class app sorted

| System | Computers | Data Disks | Sort Size | Time | Implied Disk Throughput |
|---|---|---|---|---|---|
| MinuteSort—Daytona class (general purpose) | | | | | |
| FDS, 2012 | 256 | 1,033 | 1,401 GB | 59 s | 46 MB/s |
| Yahoo!, Hadoop, 2009 [25] | 1,408 | 5,632 | 500 GB | 59 s | 3 MB/s |
| Yahoo!, Hadoop, 2009 [25] (unofficial 1 TB run) | 1,408 | 5,632 | 1,000 GB | 62 s | 5.7 MB/s |
| MinuteSort—Indy class (benchmark-specific optimizations allowed) | | | | | |
| FDS, 2012 | 256 | 1,033 | 1,470 GB | 59.4 s | 47.9 MB/s |
| UCSD TritonSort, 2011 [27] | 66 | 1,056 | 1,353 GB | 59.2 s | 43.3 MB/s |

Table 2: Comparison of FDS MinuteSort results with the previously standing records. In accordance with sort benchmark rules, all reported times are the median of 15 runs and 1 GB = $10^9$ bytes.

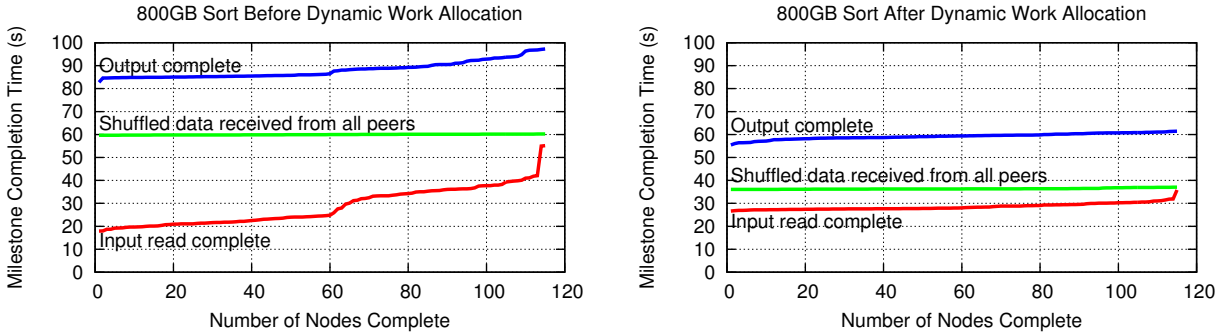



Figure 5: Visualization of the time to reach three milestones in the completion of a sort. The results are shown before (*left*) and after (*right*) implementation of dynamic work allocation. Both experiments depict 115 nodes sorting 800 GB.

1,401 GB[2] in 59.4 s. This bested the standing Daytona record by a factor of 2.8x while using about 1/5 as many CPUs and disks. Our Indy-class app sorted 1,470 GB in 59.0 s, breaking the standing record by about 8%. Our Indy sort was identical to Daytona except for assuming a uniform key distribution instead of sampling the input. Our results and comparisons to the previous record-holders are shown in Table 2. The sort application consists of one head process and $n$ worker processes. The input is given in a single FDS blob, from which each worker reads a separate subset of tracts. The sort occurs in three phases: input sampling, reading, and writing.

In the input sampling phase, the head process reads 1.5 million records: 6,000 from each of 256 tracts selected randomly from the input blob. It computes the key distribution and hence the assignment of key ranges to buckets. It then unicasts the computed distribution to all the other sort processes. In the Indy sort, this phase is skipped; the key distribution is assumed to be uniform and the bucket partitions are pre-computed.

In the read phase, each sort process performs three tasks simultaneously. First, each sort process reads tracts from its assigned region of the input. Simultaneously, as each tract arrives (in arbitrary order), the sort process shuffles the tract's records into output bins according to the bucket partitions received after the sampling phase. As each bin fills, the process sends the bin to the appropriate "bucket-receiver," a peer sort process. The buckets form an ordered partition of the keyspace. Finally, it receives bins from peer sort nodes.

In the write phase, each sort process sorts its bucket and writes it to a separate output blob. The sorted result is distributed among $n$ blobs.

Sort relied heavily on dynamic work allocation (§2.4). At first, all workers were responsible for reading equal portions of the input; 1/3 of the total sort time was lost to stragglers, partially because of our cluster's heterogeneity. In later versions, the head node coordinated work assignments: nodes requested an input range as they neared completion of their previously assigned range.

Figure 5 shows time diagrams for a sort experiment before and after dynamic work allocation was implemented. (No other changes were made between the two experiments shown.) Each graph has three lines that depict the time at which each worker reached three milestones: completing the read phase, starting the write

[2] In this section only, we use the sort benchmark's definition of 1 GB = $10^9$ bytes.

phase, and completing the write phase. Horizontal lines would be ideal, indicating every node reached a milestone simultaneously. There is a global barrier between the read and write phases; stragglers in the read phase cause significant performance loss as most nodes wait idly for the last readers to complete. Dynamic work allocation, enabled by FDS' freeing nodes from data locality, significantly improved overall sort time by reducing the time nodes were waiting at the barrier.

FDS is the first system in the history of the sort benchmark to set the record using general-purpose remote storage exclusively, without exploiting locality at all. Parallel sorts were previously accomplished by reading input data off a local disk, bucketing over the network, then writing to a local disk. In FDS, all storage is remote: our sort reads data from remote tractservers, buckets, then writes back to tractservers, sending data over the network three times. This demonstrates that FDS achieves world-class performance while still exposing a simple single-disk model that frees applications from the complexity associated with reasoning about locality.

In 2011, TritonSort [27] set the MinuteSort record with code carefully optimized for sorting. The FDS sort achieved similar per-disk bandwidth despite being built on top of a general blob store, accessing disks remotely. However, the FDS result did use approximately 4× as many computers. Part of this difference is superficial: tractservers were run on older machines, recycled from other projects; they did not have sufficient CPU or memory to act as sort clients. A single modern machine could act as both tractserver and client whereas our cluster required two. However, some of the difference is fundamental; it demonstrates the inefficiency of records traversing the network three times rather than once.

## 6.2 Cointegration

A colleague had an application that performs cointegration (a statistical technique) on stock market data to find correlations in price fluctuations. The raw input is a time-series of all stock trades. The application's first phase reads the time-series data and, for each ticker symbol it finds, generates an "order book," a list of trades for that stock on that day. The second phase compares pairs of order books to find correlations.

This application was originally implemented on a publicly available cloud-computing provider. Measurements showed it was I/O-bound, so he ported it to FDS. The public cloud implementation used one single-core VM for compute and the cloud service's blob store for storage. The FDS version used one 8-core machine for compute and 98 tractservers in an older version of our cluster that used 9 1 Gbps Ethernet interfaces per machine. Accounting for differences in hardware, the FDS version was at least 6× faster, as shown in Table 3.

| System | Cores | Time (sec) | Speedup Per Node | Speedup Per Core |
|---|---|---|---|---|
| Public cloud | 1 | 1,200 | 1 | 1 |
| FDS | 8 | 24 | 50x | 6.25x |

Table 3: Comparison of FDS to a public cloud for reading a single day of stock market data and generating one order book per stock symbol found.

A 6× speedup might actually under-state the improvement from FDS. On the cloud computing system, the VM's single core was underutilized. A multi-core VM would have been unlikely to significantly improve the speed since the bottleneck was I/O.

An interesting performance bottleneck emerged. Because this task had always been I/O-bound, the input was stored after zlib compression and decompressed on the fly. This effectively increased I/O throughput by the compression factor of 7×. In FDS, input tracts arrived so quickly that decompression was the bottleneck. 8 MB compressed tracts were, on average, arriving every 70 ms per 1 Gbps NIC. Zlib decompression took 218 ms, implying 3 cores were needed per NIC, but the machine had only 8 cores. Configuring zlib to favor speed rather than compression reduced decompression time to only 188 ms. Switching to a compression library optimized for decompression speed (XPress), compression ratio was reduced to 3:1 but tract decompression required only 62 ms. FDS transformed cointegration from an I/O-bound task to a compute-bound task.

## 6.3 Serving an index of the web

Search portals such as Bing use an index of the web to provide answers to search queries. In 2011, Bing was evaluating alternative architectures for serving the tail (i.e., infrequent) queries from the index. We ported one system to use FDS and measured its performance.

The tail index serving pipeline is composed, roughly, of two stages. First, for each term found in a user's search query, a list of the documents that contain it are retrieved from disk. The document list comes annotated with a static rank of each document's relevance with respect to that single search term. Second, these document lists are merged, ranked, and sent to a secondary ranker which computes each document's relevance to the total query based on document contents and other information. These document lists were "term-sharded," meaning each machine in the cluster was responsible for a subset of the terms found in all crawled web documents. Each machine spread its document lists across four local disks. The code had been carefully optimized.

Our effort to use FDS focused initially on a feasibility experiment that perturbed as little as possible in the index serving pipeline. Each rack of machines was converted

into an FDS cluster. The index serving pipeline, also running on the same machines, was changed to use FDS as its "local" store instead of local physical disks. This had the effect of striping all shards' data across all disks in the rack.

We ran in-house performance regression tests that issued thousands of queries to the ranker pipeline and measured the number of queries per second (QPS) served while keeping 95% of service times below the maximum specified by the SLA. We measured performance using two configurations. The first consisted of an unmodified index serving pipeline on 40 machines, each of which had 4 disks and a 10 Gb/s network connection to a single switch. The second configuration used only 12 machines but ran using an FDS cluster with 48 tractservers.

The FDS version showed a dramatic QPS improvement of 2× despite using 1/3 the machines. While the median latency was essentially the same in both versions, the FDS version reduced the 95th percentile latency by 2.4×. We attribute the difference to better statistical multiplexing of disks. Queries require document lists that vary in size by several orders of magnitude. Using local disks, long reads delay other queries behind them. FDS, in contrast, was better at spreading document lists more uniformly across all available disks, making it less likely that single machine would fall behind.

The index serving pipeline proved to be a good match for our weak consistency model. While serving the index, the document lists are read-only. When the index is updated, each term's blob is written by a single writer, and the front-end does not try to read the new blob until the update is complete.

## 7 Related Work

We believe that FDS is the first high performance blob storage system designed for datacenter scale with a flat storage model.

The Google File System has a centralized master that keeps all metadata in memory [14]. This approach is limiting because as the contents of the store grow, the metadata server becomes a centralized scaling and performance bottleneck. In a recent interview, Google architects described the GFS metadata server as a limiting factor in terms of scale and performance [22]. Additionally, a desire to reduce the size of a chunk from 64 MB was limited by the proportional increase in the number of chunks in the system.

FDS uses deterministic placement to eliminate the scaling limitations of current blob store metadata servers. The tract locator table's size is determined by the number of machines in a cluster, rather than the size of its contents. Further, the TLT enables FDS to use an unlimited number of small, 8 MB tracts to stripe data across the cluster. Finally, FDS fully distributes the blob metadata into the cluster using metadata tracts, which further minimizes the need to centrally administer the store.

xFS [6] also proposed distributing file system metadata among storage nodes through distributed metadata managers. A replicated manager map determined which manager was responsible for the location of a particular file. FDS distributes its metadata among storage nodes through the metadata tract and uses deterministic placement to eliminate the need for an additional manager service to respond to requests for the location of individual files within the storage system.

DHTs like Chord [31] use techniques such as consistent hashing [20] to eliminating the need for centralized coordination when locating data. However, under churn, a request within a DHT might be routed to several different servers before finding an up-to-date location for data. FDS also uses hashing to implement deterministic placement, but the TLT directs clients to data without ambiguity. Failures spur the generation of a new table, providing an up-to-date location to clients. FDS takes advantage of deterministic placement to minimize the load on the FDS metadata server while relying on a small amount of state to ensure that the location of data is determined within a single network hop.

RAMCloud [26] provides fast recovery from failures by distributing data, in the form of log segments, across many machines and recovering those segments in parallel. However, RAMCloud recovers all data to main memory, while FDS implements fast failure recovery back to stable storage. Further, RAMCloud distributes data to disk purely for reasons of fault tolerance, while FDS replication is used both for fault tolerance and availability. Panasas [35] uses RAID 5 to stripe files, rather than blocks, across many servers to accelerate RAID recovery, while FDS ensures that all disks participate in recovery by striping tracts among all disks in the cluster.

Distributed file systems like Frangiapani [32], GPFS [28], AFS [18] and NFS [29] export a remote storage model across a shared, hierarchical name-space. These systems must contend with strong consistency guarantees, and the vagaries of remote, shared access to a POSIX compliant API. By focusing only on fast access to blob storage, FDS provides weak consistency guarantees with very high performance.

Among many others, systems such as Swift [11], Zebra [17], GPFS [28], Panasas [35], and Petal [21] stripe files, blocks, or logs across file servers to improve read and write throughput for traditional hierarchical file systems. FDS follows in the footsteps of these systems by using the tract locator table to guarantee a uniform distribution of disk accesses regardless of the pattern of requests issued by applications.

TritonSort [27] demonstrated the power of a balanced approach to storage and computation by breaking several

of the world records in sorting. FDS demonstrates that this approach can be extended to a general purpose, locality oblivious system to break the sorting record without loss of performance.

PortLand [24] and VL2 [15] make it economically feasible to build datacenter-scale full bisection bandwidth networks. Other full bisection networks exist, such as Infiniband [23], but at a cost and scale that limited them to supercomputing and HPC environments.

Finally, River [8] used a distributed queue to dynamically adjust the assignment of work to applications nodes at run time in data flow computations. Similarly, FDS applications use dynamic work allocation to choose which node will consume a tract of data at runtime to adjust for performance faults and the varying performance capabilities within a heterogeneous cluster.

## 8 Conclusion

We have presented Flat Datacenter Storage, a datacenter-scale blob storage system that exposes the full bandwidth of its disks to all processors uniformly. It largely obviates the need for locality without sacrificing performance. Individual processes can read and write at near their NIC's rate—2 GBps or more. Aggregate client bandwidth scales nearly linearly and reaches 50% of the theoretical bandwidth of the underlying disks. Bandwidth capacity scales nearly linearly as disks are added.

This has a number of important consequences. First, recovery from failed disks can be done in seconds rather than hours; we recover 655 GB in a 1,000 disk cluster in only 33.5 s. Small recovery windows increase durability by decreasing the likelihood of complete data loss.

Second, FDS has implications for the structure of software. By exposing a cluster's full I/O bandwidth without locality constraints, FDS deconflates high I/O performance from data-parallel programming models such as MapReduce. Programmers can pick the most natural model for expressing computation without sacrificing performance.

Third, FDS has implications for the way clusters are built. Today's big-data clusters are often built with "one size fits all" machines that assume all applications have a similar balance of CPU to disk bandwidth requirements. With FDS, I/O and compute resources can be purchased separately, each independently upgradable depending on which resource is in shortage.

Finally, systems like FDS may pave the way for new kinds of applications. Large matrix operations, sorts, distributed joins, and all-to-all comparisons were largely off-limits to programmers working at datacenter scales: if it couldn't be done within a rack, it couldn't be done quickly. FDS makes these applications practical—potentially even enabling new kinds of science.

## 9 Acknowledgments

We thank our shepherd, Steve Gribble, and the anonymous reviewers for helpful comments and feedback. John Douceur, Jason Flinn and Eddie Kohler also provided insightful comments on early drafts. We are indebted to Barry Bond, who ported the cointegration application to FDS and has provided extensive guidance on performance-tuning Windows Server. Dave Maltz built our first CLOS networks and taught us how to build our own. Johnson Apacible, Rich Draves, and Reuben Olinsky were part of the sort record team. Trevor Eberl, Jamie Lee, Oleg Losinets and Lucas Williamson provided systems support. Galen Hunt provided a continuous stream of optimism and general encouragement. Li Lu helped to integrate FDS with the Bing index serving pipeline. We also thank Jim Larus for agreeing to fund our initial 14-machine cluster on nothing more than a whiteboard and a promise, allowing this work to flourish.

## References

[1] Apache Hadoop. `http://hadoop.apache.org`.

[2] MinuteSort Benchmark. `http://sortbenchmark.org`.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, pages 281–296. USENIX Association, 2010.

[4] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, editors, *SIGCOMM*, pages 63–74. ACM, 2010.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '09)*, October 2010.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, Feb. 1996.

[7] J. Apacible, R. Draves, J. Elson, J. Fan, O. Hofmann, J. Howell, E. Nightingale, R. Olinksy, and Y. Suzue. MinuteSort with Flat Datacenter Storage. Technical report, Microsoft Research, http://sortbenchmark.org/FlatDatacenterStorage2012.pdf, 2012.

[8] R. H. Arpaci-Dusseau. Run-time adaptation in River. *ACM Trans. Comput. Syst.*, 21(1):36–86, Feb. 2003.

[9] D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman, M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, and P. Weinberger. A measure of transaction processing power. *Datamation*, 31(7):112–118, Apr. 1985.

[10] D. Borthakur. The Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[11] L.-F. Cabrera and D. D. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[12] Cisco Systems. Data center: Load balancing Data Center Services, 2004.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *The 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, December 2004.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, New York, NY, USA, 2003. ACM.

[15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, Mar. 2011.

[16] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 57–62, New York, NY, USA, 2008. ACM.

[17] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 29–43, New York, NY, USA, 1993. ACM.

[18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference*, pages 59–72, 2007.

[20] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31:1203–1213, May 1999.

[21] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural support for Programming Languages and Operating Systems*, volume 31, pages 84–92, New York, NY, USA, Sept. 1996. ACM.

[22] M. K. McKusick and S. Quinlan. GFS: Evolution on fast-forward. *acmqueue*, 7(7), August 2009.

[23] Mellanox. Building a Scalable Storage with InfiniBand, 2012.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[25] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. `http://sortbenchmark.org/Yahoo2009.pdf`, 2009.

[26] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 29–41, New York, NY, USA, 2011. ACM.

[27] A. Rasmussen, G. Porter, M. Conley, H. M. andRadhika Niranjan Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[28] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *the Conference on File and Storage Technologies (FAST '02)*, January 2002.

[29] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC 3530, Apr. 2003.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[32] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, New York, NY, USA, 1997. ACM.

[33] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *The 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 91–104, 2004.

[34] V. Vasudevan, H. Shah, A. Phanishayee, E. Krevat, D. Andersen, G. Ganger, and G. Gibson. Solving TCP incast in cluster storage systems. In *The 7th USENIX Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, February 2009.

[35] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small1, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *The 6th USENIX Conference on File and Storage Technologies (FAST '08)*, 2008.

# PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

Joseph E. Gonzalez
*Carnegie Mellon University*
*jegonzal@cs.cmu.edu*

Yucheng Low
*Carnegie Mellon University*
*ylow@cs.cmu.edu*

Haijie Gu
*Carnegie Mellon University*
*haijieg@cs.cmu.edu*

Danny Bickson
*Carnegie Mellon University*
*bickson@cs.cmu.edu*

Carlos Guestrin
*University of Washington*
*guestrin@cs.washington.edu*

## Abstract

Large-scale graph-structured computation is central to tasks ranging from targeted advertising to natural language processing and has led to the development of several graph-parallel abstractions including Pregel and GraphLab. However, the natural graphs commonly found in the real-world have highly skewed power-law degree distributions, which challenge the assumptions made by these abstractions, limiting performance and scalability.

In this paper, we characterize the challenges of computation on natural graphs in the context of existing graph-parallel abstractions. We then introduce the PowerGraph abstraction which exploits the internal structure of graph programs to address these challenges. Leveraging the PowerGraph abstraction we introduce a new approach to distributed graph placement and representation that exploits the structure of power-law graphs. We provide a detailed analysis and experimental evaluation comparing PowerGraph to two popular graph-parallel systems. Finally, we describe three different implementation strategies for PowerGraph and discuss their relative merits with empirical evaluations on large-scale real-world problems demonstrating order of magnitude gains.

## 1 Introduction

The increasing need to reason about large-scale graph-structured data in machine learning and data mining (MLDM) presents a critical challenge. As the sizes of datasets grow, statistical theory suggests that we should apply richer models to eliminate the unwanted bias of simpler models, and extract stronger signals from data. At the same time, the computational and storage complexity of richer models coupled with rapidly growing datasets have exhausted the limits of single machine computation.

The resulting demand has driven the development of new *graph-parallel* abstractions such as Pregel [30] and GraphLab [29] that encode computation as *vertex-programs* which run in parallel and interact along edges in the graph. Graph-parallel abstractions rely on each vertex having a small neighborhood to maximize parallelism and effective partitioning to minimize communication. However, graphs derived from real-world phenomena, like social networks and the web, typically have *power-law* degree distributions, which implies that a small subset of the vertices connects to a large fraction of the graph. Furthermore, power-law graphs are difficult to partition [1, 28] and represent in a distributed environment.

To address the challenges of power-law graph computation, we introduce the PowerGraph abstraction which exploits the structure of vertex-programs and explicitly factors computation over edges instead of vertices. As a consequence, PowerGraph exposes substantially greater parallelism, reduces network communication and storage costs, and provides a new highly effective approach to distributed graph placement. We describe the design of our distributed implementation of PowerGraph and evaluate it on a large EC2 deployment using real-world applications. In particular our key contributions are:

1. An analysis of the challenges of power-law graphs in distributed graph computation and the limitations of existing graph parallel abstractions (Sec. 2 and 3).
2. The PowerGraph abstraction (Sec. 4) which factors individual vertex-programs.
3. A delta caching procedure which allows computation state to be dynamically maintained (Sec. 4.2).
4. A new fast approach to data layout for power-law graphs in distributed environments (Sec. 5).
5. An theoretical characterization of network and storage (Theorem 5.2, Theorem 5.3).
6. A high-performance open-source implementation of the PowerGraph abstraction (Sec. 7).
7. A comprehensive evaluation of three implementations of PowerGraph on a large EC2 deployment using real-world MLDM applications (Sec. 6 and 7).

## 2 Graph-Parallel Abstractions

A **graph-parallel** abstraction consists of a *sparse* graph $G = \{V, E\}$ and a **vertex-program** $Q$ which is executed in parallel on each vertex $v \in V$ and can interact (e.g., through shared-state in GraphLab, or messages in Pregel) with neighboring instances $Q(u)$ where $(u, v) \in E$. In contrast to more general message passing models, graph-parallel abstractions constrain the interaction of vertex-program to a graph structure enabling the optimization of data-layout and communication. We focus our discussion on Pregel and GraphLab as they are representative of existing graph-parallel abstractions.

### 2.1 Pregel

Pregel [30] is a bulk synchronous *message passing* abstraction in which all vertex-programs run simultaneously in a sequence of super-steps. Within a **super-step** each program instance $Q(v)$ receives all messages from the previous super-step and sends messages to its neighbors in the next super-step. A barrier is imposed between super-steps to ensure that all program instances finish processing messages from the previous super-step before proceeding to the next. The program terminates when there are no messages remaining and every program has voted to halt. Pregel introduces commutative associative message **combiners** which are user defined functions that merge messages destined to the same vertex. The following is an example of the PageRank vertex-program implemented in Pregel. The vertex-program receives the single incoming message (after the combiner) which contains the sum of the PageRanks of all in-neighbors. The new PageRank is then computed and sent to its out-neighbors.

```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
     SendMsg(nbr, vertex.val/num_out_nbrs);
```

### 2.2 GraphLab

GraphLab [29] is an *asynchronous distributed shared-memory* abstraction in which vertex-programs have shared access to a distributed graph with data stored on every vertex and edge. Each vertex-program may directly access information on the current vertex, adjacent edges, and adjacent vertices irrespective of edge direction. Vertex-programs can schedule neighboring vertex-programs to be executed in the future. GraphLab ensures serializability by preventing neighboring program instances from running simultaneously. The following is an example of the PageRank vertex-program implemented in GraphLab. The GraphLab vertex-program directly reads neighboring vertex values to compute the sum.

```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
     accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

By eliminating messages, GraphLab isolates the user defined algorithm from the movement of data, allowing the system to choose when and how to move program state. By allowing mutable data to be associated with both vertices *and edges* GraphLab allows the algorithm designer to more precisely distinguish between data shared with all neighbors (vertex data) and data shared with a particular neighbor (edge data).

### 2.3 Characterization

While the implementation of MLDM vertex-programs in GraphLab and Pregel differ in how they collect and disseminate information, they share a common overall structure. To characterize this common structure and differentiate between vertex and edge specific computation we introduce the GAS model of graph computation.

The **GAS** model represents three *conceptual* phases of a vertex-program: **G**ather, **A**pply, and **S**catter. In the **gather** phase, information about adjacent vertices and edges is collected through a generalized sum over the neighborhood of the vertex $u$ on which $Q(u)$ is run:

$$\Sigma \leftarrow \bigoplus_{v \in \mathbf{Nbr}[u]} g\left(D_u, D_{(u,v)}, D_v\right). \tag{2.1}$$

where $D_u$, $D_v$, and $D_{(u,v)}$ are the values (program state and meta-data) for vertices $u$ and $v$ and edge $(u, v)$. The user defined sum $\oplus$ operation must be commutative and associative and can range from a numerical sum to the union of the data on all neighboring vertices and edges.

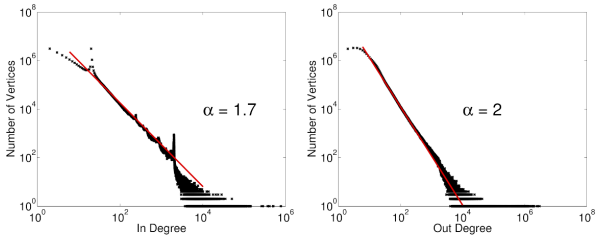
The resulting value $\Sigma$ is used in the **apply** phase to update the value of the central vertex:

$$D_u^{\text{new}} \leftarrow a\left(D_u, \Sigma\right). \tag{2.2}$$

Finally the **scatter** phase uses the new value of the central vertex to update the data on adjacent edges:

$$\forall v \in \mathbf{Nbr}[u]: \quad \left(D_{(u,v)}\right) \leftarrow s\left(D_u^{\text{new}}, D_{(u,v)}, D_v\right). \tag{2.3}$$

The fan-in and fan-out of a vertex-program is determined by the corresponding gather and scatter phases. For instance, in PageRank, the gather phase only operates on in-edges and the scatter phase only operates on out-edges. However, for many MLDM algorithms the graph edges encode ostensibly symmetric relationships, like friendship, in which both the gather and scatter phases touch all edges. In this case the fan-in and fan-out are equal. As we will show in Sec. 3, the ability for graph parallel abstractions to support both high fan-in and fan-out computation is critical for efficient computation on natural graphs.




(a) Twitter In-Degree (b) Twitter Out-Degree

Figure 1: The in and out degree distributions of the Twitter follower network plotted in log-log scale.

GraphLab and Pregel express GAS programs in very different ways. In the Pregel abstraction the gather phase is implemented using message combiners and the apply and scatter phases are expressed in the vertex program. Conversely, GraphLab exposes the entire neighborhood to the vertex-program and allows the user to define the gather and apply phases within their program. The GraphLab abstraction implicitly defines the communication aspects of the gather/scatter phases by ensuring that changes made to the vertex or edge data are automatically visible to adjacent vertices. It is also important to note that GraphLab does not differentiate between edge directions.

## 3 Challenges of Natural Graphs

The sparsity structure of natural graphs presents a unique challenge to efficient distributed graph-parallel computation. One of the hallmark properties of natural graphs is their *skewed* power-law degree distribution[16]: most vertices have relatively few neighbors while a few have many neighbors (e.g., celebrities in a social network). Under a power-law degree distribution the probability that a vertex has degree $d$ is given by:

$$\mathbf{P}(d) \propto d^{-\alpha}, \tag{3.1}$$

where the exponent $\alpha$ is a positive constant that controls the "skewness" of the degree distribution. Higher $\alpha$ implies that the graph has lower density (ratio of edges to vertices), and that the vast majority of vertices are low degree. As $\alpha$ decreases, the graph density and number of high degree vertices increases. Most natural graphs typically have a power-law constant around $\alpha \approx 2$. For example, Faloutsos et al. [16] estimated that the inter-domain graph of the Internet has a power-law constant $\alpha \approx 2.2$. One can visualize the skewed power-law degree distribution by plotting the number of vertices with a given degree in log-log scale. In Fig. 1, we plot the in and out degree distributions of the Twitter follower network demonstrating the characteristic linear power-law form.

While power-law degree distributions are empirically observable, they do not fully characterize the properties of natural graphs. While there has been substantial work (see [27]) in more sophisticated natural graph models, the techniques in this paper focus only on the degree distribution and do not require any other modeling assumptions.

The skewed degree distribution implies that a small fraction of the vertices are adjacent to a large fraction of the edges. For example, one percent of the vertices in the Twitter web-graph are adjacent to nearly half of the edges. This concentration of edges results in a *star-like* motif which presents challenges for existing graph-parallel abstractions:

**Work Balance:** The power-law degree distribution can lead to substantial work imbalance in graph parallel abstractions that treat vertices symmetrically. Since the storage, communication, and computation complexity of the Gather and Scatter phases is linear in the degree, the running time of vertex-programs can vary widely [36].

**Partitioning:** Natural graphs are difficult to partition[26, 28]. Both GraphLab and Pregel depend on graph partitioning to minimize communication and ensure work balance. However, in the case of natural graphs both are forced to resort to hash-based (random) partitioning which has extremely poor locality (Sec. 5).

**Communication:** The skewed degree distribution of natural-graphs leads to communication asymmetry and consequently bottlenecks. In addition, high-degree vertices can force messaging abstractions, such as Pregel, to generate and send many identical messages.

**Storage:** Since graph parallel abstractions must locally store the adjacency information for each vertex, each vertex requires memory linear in its degree. Consequently, high-degree vertices can exceed the memory capacity of a single machine.

**Computation:** While multiple vertex-programs may execute in parallel, existing graph-parallel abstractions do not parallelize *within* individual vertex-programs, limiting their scalability on high-degree vertices.

## 4 PowerGraph Abstraction

To address the challenges of computation on power-law graphs, we introduce PowerGraph, a new graph-parallel abstraction that eliminates the degree dependence of the vertex-program by directly exploiting the GAS decomposition to factor vertex-programs over edges. By lifting the Gather and Scatter phases into the abstraction, PowerGraph is able to retain the natural "think-like-a-vertex" philosophy [30] while distributing the computation of a single vertex-program over the entire cluster.

PowerGraph combines the best features from both Pregel and GraphLab. From GraphLab, PowerGraph borrows the data-graph and shared-memory view of computation eliminating the need for users to architect the movement of information. From Pregel, PowerGraph borrows the commutative, associative gather concept. PowerGraph

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Figure 2: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

**Algorithm 1:** Vertex-Program Execution Semantics

**Input**: Center vertex $u$
**if** *cached accumulator $a_u$ is empty* **then**
  **foreach** *neighbor $v$ in gather_nbrs(u)* **do**
    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$
  **end**
**end**
$D_u \leftarrow \text{apply}(D_u, a_u)$
**foreach** *neighbor $v$ scatter_nbrs(u)* **do**
  $(D_{(u,v)}, \Delta a) \leftarrow \text{scatter}(D_u, D_{(u,v)}, D_v)$
  **if** *$a_v$ and $\Delta a$ are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
  **else** $a_v \leftarrow$ Empty
**end**

supports both the highly-parallel bulk-synchronous Pregel model of computation as well as the computationally efficient asynchronous GraphLab model of computation.

Like GraphLab, the state of a PowerGraph program factors according to a **data-graph** with user defined vertex data $D_v$ and edge data $D_{(u,v)}$. The data stored in the data-graph includes both meta-data (e.g., urls and edge weights) as well as computation state (e.g., the PageRank of vertices). In Sec. 5 we introduce vertex-cuts which allow PowerGraph to efficiently represent and store power-law graphs in a distributed environment. We now describe the PowerGraph *abstraction* and how it can be used to naturally decompose vertex-programs. Then in Sec. 5 through Sec. 7 we discuss how to implement the PowerGraph abstraction in a distributed environment.

## 4.1 GAS Vertex-Programs

Computation in the PowerGraph abstraction is encoded as a state-less **vertex-program** which implements the `GASVertexProgram` interface (Fig. 2) and therefore explicitly factors into the gather, sum, apply, and scatter functions. Each function is invoked in stages by the PowerGraph engine following the semantics in Alg. 1. By factoring the vertex-program, the PowerGraph execution engine can distribute a single vertex-program over multiple machines and move computation to the data.

During the gather phase the `gather` and `sum` functions are used as a *map* and *reduce* to collect information about the neighborhood of the vertex. The `gather` function is invoked in parallel on the edges adjacent to $u$. The particular set of edges is determined by `gather_nbrs` which can be `none`, `in`, `out`, or `all`. The gather function is passed the data on the adjacent vertex and edge and returns a temporary accumulator (a user defined type). The result is combined using the commutative and associative `sum` operation. The final result $a_u$ of the gather phase is passed to the apply phase and cached by PowerGraph.

After the gather phase has completed, the `apply` function takes the final accumulator and computes a new vertex value $D_u$ which is atomically written back to the graph. The size of the accumulator $a_u$ and complexity of the apply function play a central role in determining the network and storage efficiency of the PowerGraph abstraction and should be *sub-linear* and ideally constant in the degree.

During the scatter phase, the `scatter` function is invoked in parallel on the edges adjacent to $u$ producing new edge values $D_{(u,v)}$ which are written back to the data-graph. As with the gather phase, the `scatter_nbrs` determines the particular set of edges on which scatter is invoked. The scatter function returns an optional value $\Delta a$ which is used to dynamically update the cached accumulator $a_v$ for the adjacent vertex (see Sec. 4.2).

In Fig. 3 we implement the PageRank, greedy graph coloring, and single source shortest path algorithms using the PowerGraph abstraction. In PageRank the gather and sum functions collect the total value of the adjacent vertices, the apply function computes the new PageRank, and the scatter function is used to activate adjacent vertex-programs if necessary. In graph coloring the gather and sum functions collect the set of colors on adjacent vertices, the apply function computes a new color, and the scatter function activates adjacent vertices if they violate the coloring constraint. Finally in single source shortest path (SSSP), the gather and sum functions compute the shortest path through each of the neighbors, the apply function returns the new distance, and the scatter function activates affected neighbors.

## 4.2 Delta Caching

In many cases a vertex-program will be triggered in response to a change in a *few* of its neighbors. The gather operation is then repeatedly invoked on *all* neighbors, many of which remain unchanged, thereby wasting computation cycles. For many algorithms [2] it is possible to dynamically maintain the result of the gather phase $a_u$ and skip the gather on subsequent iterations.

The PowerGraph engine maintains a cache of the accumulator $a_u$ from the previous gather phase for each vertex. The scatter function can *optionally* return an additional $\Delta a$ which is atomically added to the cached accumulator $a_v$ of the neighboring vertex $v$ using the `sum` function. If $\Delta a$ is not returned, then the neighbor's cached $a_v$ is cleared,

PageRank

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
  rnew = 0.15 + 0.85 * acc
  D_u.delta = (rnew - D_u.rank)/
              #outNbrs(u)
  D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u,D_(u,v),D_v):
  if(|D_u.delta|>ε) Activate(v)
  return delta
```

Greedy Graph Coloring

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

Single Source Shortest Path (SSSP)

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v + D_(v,u)
sum(a, b): return min(a, b)
apply(D_u, new_dist):
  D_u = new_dist
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // If changed activate neighbor
  if(changed(D_u)) Activate(v)
  if(increased(D_u))
    return NULL
  else return D_u + D_(u,v)
```

Figure 3: The PageRank, graph-coloring, and single source shortest path algorithms implemented in the PowerGraph abstraction. Both the PageRank and single source shortest path algorithms support delta caching in the gather phase.

forcing a complete gather on the subsequent execution of the vertex-program on the vertex $v$. When executing the vertex-program on $v$ the PowerGraph engine uses the cached $a_v$ if available, bypassing the gather phase.

Intuitively, $\Delta a$ acts as an additive correction on-top of the previous gather for that edge. More formally, if the accumulator type forms an **abelian group**: has a commutative and associative sum $(+)$ and an *inverse* $(-)$ operation, then we can define (shortening `gather` to $g$):

$$\Delta a = g(D_u, D^{\text{new}}_{(u,v)}, D^{\text{new}}_v) - g(D_u, D_{(u,v)}, D_v). \qquad (4.1)$$

In the PageRank example (Fig. 3) we take advantage of the abelian nature of the PageRank sum operation. For graph coloring the set union operation is not abelian and so we invalidate the accumulator.

## 4.3 Initiating Future Computation

The PowerGraph engine maintains a set of active vertices on which to eventually execute the vertex-program. The user initiates computation by calling `Activate(v)` or `Activate_all()`. The PowerGraph engine then proceeds to execute the vertex-program on the active vertices until none remain. Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.

Vertices can activate themselves and neighboring vertices. Each function in a vertex-program can only activate vertices visible in the arguments to that function. For example the scatter function invoked on the edge $(u,v)$ can only activate the vertices $u$ and $v$. This restriction is essential to ensure that activation events are generated on machines on which they can be efficiently processed.

The order in which activated vertices are executed is up to the PowerGraph execution engine. The only guarantee is that all activated vertices are eventually executed. This flexibility in scheduling enables PowerGraph programs to be executed both *synchronously* and *asynchronously*, leading to different tradeoffs in algorithm performance, system performance, and determinism.

### 4.3.1 Bulk Synchronous Execution

When run synchronously, the PowerGraph engine executes the gather, apply, and scatter phases in order. Each phase, called a **minor-step**, is run synchronously on all active vertices with a barrier at the end. We define a **super-step** as a complete series of GAS minor-steps. Changes made to the vertex data and edge data are committed at the end of each minor-step and are visible in the subsequent minor-step. Vertices activated in each super-step are executed in the subsequent super-step.

The synchronous execution model ensures a deterministic execution regardless of the number of machines and closely resembles Pregel. However, the frequent barriers and inability to operate on the most recent data can lead to an inefficient distributed execution and slow algorithm convergence. To address these limitations PowerGraph also supports asynchronous execution.

### 4.3.2 Asynchronous Execution

When run asynchronously, the PowerGraph engine executes active vertices as processor and network resources become available. Changes made to the vertex and edge data during the apply and scatter functions are immediately committed to the graph and visible to subsequent computation on neighboring vertices.

By using processor and network resources as they become available and making any changes to the data-graph immediately visible to future computation, an asynchronous execution can more effectively utilize resources and accelerate the convergence of the underlying algorithm. For example, the greedy graph-coloring algorithm in Fig. 3 will not converge when executed synchronously but converges quickly when executed asynchronously. The merits of asynchronous computation have been studied extensively in the context of numerical algorithms [4]. In [18, 19, 29] we demonstrated that asynchronous computation can lead to both theoretical and empirical

gains in algorithm and system performance for a range of important MLDM applications.

Unfortunately, the behavior of the asynchronous execution depends on the number machines and availability of network resources leading to non-determinism that can complicate algorithm design and debugging. Furthermore, for some algorithms, like statistical simulation, the resulting non-determinism, if not carefully controlled, can lead to instability or even divergence [17].

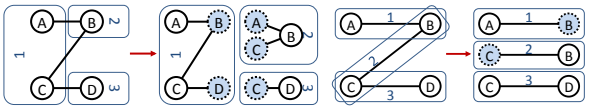
To address these challenges, GraphLab automatically enforces **serializability**: every parallel execution of vertex-programs has a corresponding sequential execution. In [29] it was shown that serializability is sufficient to support a wide range of MLDM algorithms. To achieve serializability, GraphLab prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol which requires *sequentially* grabbing locks on all neighboring vertices. Furthermore, the locking scheme used by GraphLab is unfair to high degree vertices.

PowerGraph retains the strong serializability guarantees of GraphLab while addressing its limitations. We address the problem of sequential locking by introducing a new *parallel* locking protocol (described in Sec. 7.4) which is fair to high degree vertices. In addition, the PowerGraph abstraction exposes substantially more fined grained (edge-level) parallelism allowing the entire cluster to support the execution of individual vertex programs.

## 4.4 Comparison with GraphLab / Pregel

Surprisingly, despite the strong constraints imposed by the PowerGraph abstraction, it is *possible* to emulate both GraphLab and Pregel vertex-programs in PowerGraph. To emulate a GraphLab vertex-program, we use the gather and sum functions to *concatenate* all the data on adjacent vertices and edges and then run the GraphLab program within the apply function. Similarly, to express a Pregel vertex-program, we use the gather and sum functions to combine the inbound messages (stored as edge data) and *concatenate* the list of neighbors needed to compute the outbound messages. The Pregel vertex-program then runs within the apply function generating the set of messages which are passed as vertex data to the scatter function where they are written back to the edges.

In order to address the challenges of natural graphs, the PowerGraph abstraction requires the size of the accumulator and the complexity of the apply function to be sub-linear in the degree. However, directly executing GraphLab and Pregel vertex-programs within the apply function leads the size of the accumulator and the complexity of the apply function to be *linear* in the degree eliminating many of the benefits on natural graphs.




(a) Edge-Cut (b) Vertex-Cut

Figure 4: **(a)** An edge-cut and **(b)** vertex-cut of a graph into three parts. Shaded vertices are ghosts and mirrors respectively.

# 5 Distributed Graph Placement

The PowerGraph abstraction relies on the distributed data-graph to store the computation state and encode the interaction between vertex-programs. The placement of the data-graph structure and data plays a central role in minimizing communication and ensuring work balance.

A common approach to placing a graph on a cluster of $p$ machines is to construct a balanced $p$-way **edge-cut** (e.g., Fig. 4a) in which vertices are evenly assigned to machines and the number of edges spanning machines is minimized. Unfortunately, the tools [23, 31] for constructing balanced edge-cuts perform poorly [1, 28, 26] on power-law graphs. When the graph is difficult to partition, both GraphLab and Pregel resort to hashed (random) vertex placement. While fast and easy to implement, hashed vertex placement cuts most of the edges:

**Theorem 5.1.** *If vertices are randomly assigned to $p$ machines then the expected fraction of edges cut is:*

$$\mathbb{E}\left[\frac{|\mathit{Edges\ Cut}|}{|E|}\right] = 1 - \frac{1}{p}. \tag{5.1}$$

*For a power-law graph with exponent $\alpha$, the expected number of edges cut per-vertex is:*

$$\mathbb{E}\left[\frac{|\mathit{Edges\ Cut}|}{|V|}\right] = \left(1 - \frac{1}{p}\right)\mathbb{E}\left[\mathbf{D}[v]\right] = \left(1 - \frac{1}{p}\right)\frac{\mathbf{h}_{|V|}(\alpha - 1)}{\mathbf{h}_{|V|}(\alpha)}, \tag{5.2}$$

*where the $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.*

*Proof.* An edge is cut if both vertices are randomly assigned to different machines. The probability that both vertices are assigned to different machines is $1 - 1/p$. □

Every cut edge contributes to storage and network overhead since both machines maintain a copy of the adjacency information and in some cases [20], a **ghost** (local copy) of the vertex and edge data. For example in Fig. 4a we construct a three-way edge-cut of a four vertex graph resulting in five ghost vertices and all edge data being replicated. Any changes to vertex and edge data associated with a cut edge must be synchronized across the network. For example, using just two machines, a random cut will cut roughly *half* the edges, requiring $|E|/2$ communication.

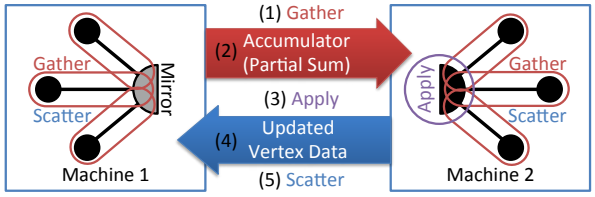



Figure 5: The communication pattern of the PowerGraph abstraction when using a vertex-cut. Gather function runs locally on each machine and then one accumulators is sent from each mirror to the master. The master runs the apply function and then sends the updated vertex data to all mirrors. Finally the scatter phase is run in parallel on mirrors.

## 5.1 Balanced $p$-way Vertex-Cut

By factoring the vertex program along the edges in the graph, The PowerGraph abstraction allows a single vertex-program to span multiple machines. In Fig. 5 a single high degree vertex program has been split across two machines with the gather and scatter functions running in parallel on each machine and accumulator and vertex data being exchanged across the network.

Because the PowerGraph abstraction allows a single vertex-program to span multiple machines, we can improve work balance and reduce communication and storage overhead by evenly *assigning edges* to machines and allowing *vertices to span machines*. Each machine only stores the edge information for the edges assigned to that machine, evenly distributing the massive amounts of edge data. Since each edge is stored exactly once, changes to edge data do not need to be communicated. However, changes to vertex must be copied to all the machines it spans, thus the storage and network overhead depend on the number of machines spanned by each vertex.

We minimize storage and network overhead by limiting the number of machines spanned by each vertex. A balanced $p$-way **vertex-cut** formalizes this objective by assigning each *edge* $e \in E$ to a machine $A(e) \in \{1,\ldots,p\}$. Each vertex then spans the set of machines $A(v) \subseteq \{1,\ldots,p\}$ that contain its adjacent edges. We define the balanced vertex-cut objective:

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{5.3}$$

$$\text{s.t.} \qquad \max_{m} |\{e \in E \mid A(e) = m\}| < \lambda \frac{|E|}{p} \tag{5.4}$$

where the imbalance factor $\lambda \geq 1$ is a small constant. We use the term **replicas** of a vertex $v$ to denote the $|A(v)|$ copies of the vertex $v$: each machine in $A(v)$ has a replica of $v$. Because changes to vertex data are communicated to all replicas, the communication overhead is also given by $|A(v)|$. The objective (Eq. 5.3) therefore minimizes the average number of replicas in the graph and as a consequence the total storage and communication requirements of the PowerGraph engine.

For each vertex $v$ with multiple replicas, one of the replicas is *randomly* nominated as the **master** which maintains the master version of the vertex data. All remaining replicas of $v$ are then **mirrors** and maintain a local cached *read only* copy of the vertex data. (e.g., Fig. 4b). For instance, in Fig. 4b we construct a three-way vertex-cut of a graph yielding only 2 mirrors. Any changes to the vertex data (e.g., the Apply function) must be made to the master which is then immediately replicated to all mirrors.

Vertex-cuts address the major issues associated with edge-cuts in power-law graphs. Percolation theory [3] suggests that power-law graphs have good vertex-cuts. Intuitively, by cutting a small fraction of the very high degree vertices we can quickly shatter a graph. Furthermore, because the balance constraint (Eq. 5.4) ensures that edges are uniformly distributed over machines, we naturally achieve improved work balance even in the presence of very high-degree vertices.

The simplest method to construct a vertex cut is to randomly assign edges to machines. Random (hashed) edge placement is fully data-parallel, achieves nearly perfect balance on large graphs, and can be applied in the streaming setting. In the following theorem, we relate the expected normalized replication factor (Eq. 5.3) to the number of machines and the power-law constant $\alpha$.

**Theorem 5.2** (Randomized Vertex Cuts)**.** *A random vertex-cut on $p$ machines has an expected replication:*

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = \frac{p}{|V|}\sum_{v \in V}\left(1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right). \tag{5.5}$$

*where $\mathbf{D}[v]$ denotes the degree of vertex $v$. For a power-law graph the expected replication (Fig. 6a) is determined entirely by the power-law constant $\alpha$:*

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)} \sum_{d=1}^{|V|-1} \left(\frac{p-1}{p}\right)^{d} d^{-\alpha}, \tag{5.6}$$

*where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.*

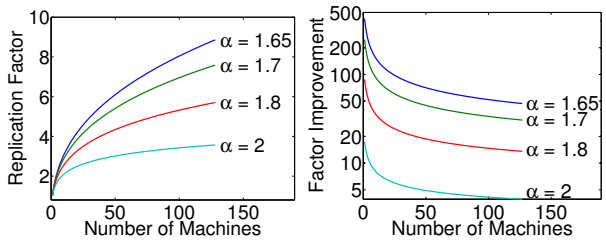
*Proof.* By linearity of expectation:

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = \frac{1}{|V|}\sum_{v \in V} \mathbb{E}\left[|A(v)|\right], \tag{5.7}$$

The expected replication $\mathbb{E}[|A(v)|]$ of a single vertex $v$ can be computed by considering the process of randomly assigning the $\mathbf{D}[v]$ edges adjacent to $v$. Let the indicator $X_i$ denote the event that vertex $v$ has at least one of its edges on machine $i$. The expectation $\mathbb{E}[X_i]$ is then:

$$\mathbb{E}[X_i] = 1 - \mathbf{P}(v \text{ has no edges on machine } i) \tag{5.8}$$

$$= 1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}, \tag{5.9}$$




(a) V-Sep. Bound (b) V-Sep. Improvement

Figure 6: **(a)** Expected replication factor for different power-law constants. **(b)** The ratio of the expected communication and storage cost of random edge cuts to random vertex cuts as a function of the number machines. This graph assumes that edge data and vertex data are the same size.

The expected replication factor for vertex $v$ is then:

$$\mathbb{E}\left[|A(v)|\right] = \sum_{i=1}^{p} \mathbb{E}\left[X_i\right] = p\left(1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right). \quad (5.10)$$

Treating $\mathbf{D}[v]$ as a Zipf random variable:

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = \frac{p}{|V|}\sum_{v\in V}\left(1 - \mathbb{E}\left[\left(\frac{p-1}{p}\right)^{\mathbf{D}[v]}\right]\right), \quad (5.11)$$

and taking the expectation under $\mathbf{P}(d) = d^{-\alpha}/\mathbf{h}_{|V|}(\alpha)$:

$$\mathbb{E}\left[\left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right] = \frac{1}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(1 - \frac{1}{p}\right)^{d} d^{-\alpha}. \quad (5.12)$$

□

While lower $\alpha$ values (more high-degree vertices) imply a higher replication factor (Fig. 6a) the effective gains of vertex-cuts relative to edge cuts (Fig. 6b) actually *increase* with lower $\alpha$. In Fig. 6b we plot the ratio of the expected costs (comm. and storage) of random edge-cuts (Eq. 5.2) to the expected costs of random vertex-cuts (Eq. 5.6) demonstrating order of magnitude gains.

Finally, the vertex cut model is also highly effective for regular graphs since in the event that a good edge-cut can be found it can be converted to a better vertex cut:

**Theorem 5.3.** *For a given an edge-cut with $g$ ghosts, **any** vertex cut along the same partition boundary has strictly fewer than $g$ mirrors.*

*Proof of Theorem 5.3.* Consider the two-way edge cut which cuts the set of edges $E' \in E$ and let $V'$ be the set of vertices in $E'$. The total number of ghosts induced by this edge partition is therefore $|V'|$. If we then select and delete arbitrary vertices from $V'$ along with their adjacent edges until no edges remain, then the set of deleted vertices corresponds to a vertex-cut in the original graph. Since at most $|V'|-1$ vertices may be deleted, there can be at most $|V'|-1$ mirrors. □

| Graph | $\lvert V\rvert$ | $\lvert E\rvert$ |
|---|---|---|
| Twitter [24] | 41M | 1.4B |
| UK [7] | 132.8M | 5.5B |
| Amazon [6, 5] | 0.7M | 5.2M |
| LiveJournal [12] | 5.4M | 79M |
| Hollywood [6, 5] | 2.2M | 229M |

(a) Real world graphs

| $\alpha$ | # Edges |
|---|---|
| 1.8 | 641,383,778 |
| 1.9 | 245,040,680 |
| 2.0 | 102,838,432 |
| 2.1 | 57,134,471 |
| 2.2 | 35,001,696 |

(b) Synthetic Graphs

Table 1: **(a)** A collection of Real world graphs. **(b)** Randomly constructed ten-million vertex power-law graphs with varying $\alpha$. Smaller $\alpha$ produces denser graphs.

## 5.2 Greedy Vertex-Cuts

We can improve upon the randomly constructed vertex-cut by de-randomizing the edge-placement process. The resulting algorithm is a sequential greedy heuristic which places the next edge on the machine that minimizes the conditional expected replication factor. To construct the de-randomization we consider the task of placing the $i+1$ edge after having placed the previous $i$ edges. Using the conditional expectation we define the objective:

$$\arg\min_{k} \mathbb{E}\left[\sum_{v\in V}|A(v)| \,\middle|\, A_i, A(e_{i+1}) = k\right], \quad (5.13)$$

where $A_i$ is the assignment for the previous $i$ edges. Using Theorem 5.2 to evaluate Eq. 5.13 we obtain the following edge placement rules for the edge $(u,v)$:

**Case 1:** If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection.

**Case 2:** If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

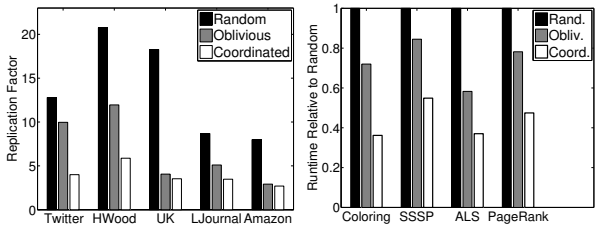
**Case 3:** If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

**Case 4:** If neither vertex has been assigned, then assign the edge to the least loaded machine.

Because the greedy-heuristic is a de-randomization it is guaranteed to obtain an expected replication factor that is no worse than random placement and in practice can be much better. Unlike the randomized algorithm, which is embarrassingly parallel and easily distributed, the greedy algorithm requires coordination between machines. We consider two distributed implementations:

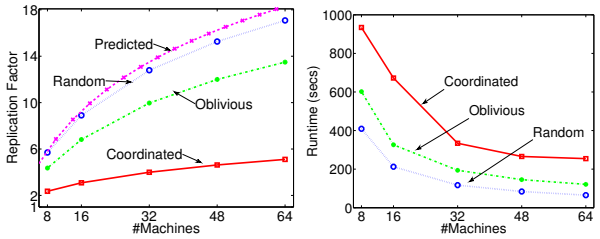
**Coordinated:** maintains the values of $A_i(v)$ in a distributed table. Then each machine runs the greedy heuristic and periodically updates the distributed table. Local caching is used to reduce communication at the expense of accuracy in the estimate of $A_i(v)$.

**Oblivious:** runs the greedy heuristic independently on each machine. Each machine maintains its own estimate of $A_i$ with no additional communication.




(a) Actual Replication (b) Effect of Partitioning

Figure 7: **(a)** The actual replication factor on 32 machines. **(b)** The effect of partitioning on runtime.




(a) Replication Factor (Twitter) (b) Ingress time (Twitter)

Figure 8: **(a,b)** Replication factor and runtime of graph ingress for the Twitter follower network as a function of the number of machines for random, oblivious, and coordinated vertex-cuts.

In Fig. 8a, we compare the replication factor of both heuristics against random vertex cuts on the Twitter follower network. We plot the replication factor as a function of the number of machines (EC2 instances described in Sec. 7) and find that random vertex cuts match the predicted replication given in Theorem 5.2. Furthermore, the greedy heuristics substantially improve upon random placement with an order of magnitude reduction in the replication factor, and therefore communication and storage costs. For a fixed number of machines ($p = 32$), we evaluated (Fig. 7a) the replication factor of the two heuristics on five real-world graphs (Tab. 1a). In all cases the greedy heuristics out-perform random placement, while doubling the load time (Fig. 8b). The Oblivious heuristic achieves compromise by obtaining a relatively low replication factor while only slightly increasing runtime.

# 6 Abstraction Comparison

In this section, we experimentally characterize the dependence on $\alpha$ and the relationship between fan-in and fan-out by using the Pregel, GraphLab, and PowerGraph abstractions to run PageRank on five synthetically constructed power-law graphs. Each graph has ten-million vertices and an $\alpha$ ranging from 1.8 to 2.2. The graphs were constructed by randomly sampling the out-degree of each vertex from a Zipf distribution and then adding out-edges such that the in-degree of each vertex is nearly identical. We then inverted each graph to obtain the corresponding power-law fan-in graph. The density of each power-law graph is determined by $\alpha$ and therefore each graph has a different number of edges (see Tab. 1b).

We used the GraphLab v1 C++ implementation from [29] and added instrumentation to track network usage. As of the writing of this paper, public implementations of Pregel (e.g., Giraph) were unable to handle even our smaller synthetic problems due to memory limitations. Consequently, we used Piccolo [32] as a proxy implementation of Pregel since Piccolo naturally expresses the Pregel abstraction and provides an efficient C++ implementation with dynamic load-balancing. Finally, we used our implementation of PowerGraph described in Sec. 7.

All experiments in this section are evaluated on an eight node Linux cluster. Each node consists of two quad-core Intel Xeon E5620 processors with 32 GB of RAM and is connected via 1-GigE Ethernet. All systems were compiled with GCC 4.4. GraphLab and Piccolo used random edge-cuts while PowerGraph used random vertex-cuts.. Results are averaged over 20 iterations.

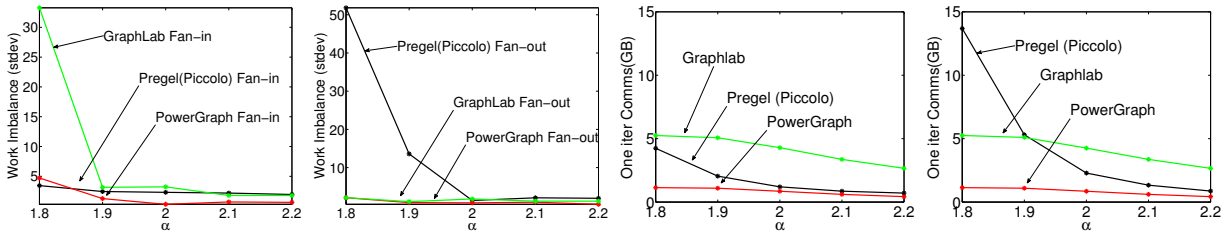
## 6.1 Computation Imbalance

The *sequential* component of the PageRank vertex-program is proportional to out-degree in the Pregel abstraction and in-degree in the GraphLab abstraction. Alternatively, PowerGraph eliminates this sequential dependence by distributing the computation of *individual* vertex-programs over multiple machines. Therefore we expect, highly-skewed (low $\alpha$) power-law graphs to increase work imbalance under the Pregel (fan-in) and GraphLab (fan-out) abstractions but not under the PowerGraph abstraction, which evenly distributed high-degree vertex-programs. To evaluate this hypothesis we ran eight "workers" per system (64 total workers) and recorded the vertex-program time on each worker.

In Fig. 9a and Fig. 9b we plot the *standard deviation* of worker per-iteration runtimes, a measure of work imbalance, for power-law fan-in and fan-out graphs respectively. Higher standard deviation implies greater imbalance. While lower $\alpha$ increases work imbalance for GraphLab (on fan-in) and Pregel (on fan-out), the PowerGraph abstraction is unaffected in either edge direction.

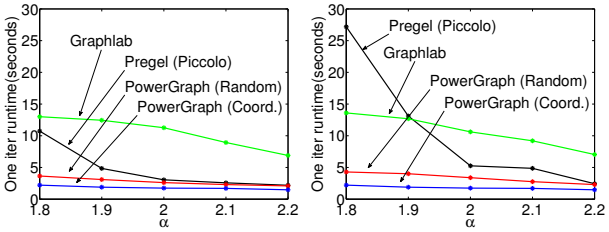
## 6.2 Communication Imbalance

Because GraphLab and Pregel use edge-cuts, their communication volume is proportional to the number of ghosts: the replicated vertex and edge data along the partition boundary. If one message is sent per edge, Pregel's combiners ensure that exactly one network message is transmitted for each ghost. Similarly, at the end of each iteration GraphLab synchronizes each ghost and thus the communication volume is also proportional to the number of ghosts. PowerGraph on the other hand uses vertex-cuts and only synchronizes mirrors after each iteration. The communication volume of a complete iteration is therefore proportional to the number of mirrors induced by the




(a) Power-law Fan-In Balance (b) Power-law Fan-Out Balance (c) Power-law Fan-In Comm. (d) Power-law Fan-Out Comm.

Figure 9: **Synthetic Experiments: Work Imbalance and Communication. (a, b)** Standard deviation of worker computation time across 8 distributed workers for each abstraction on power-law fan-in and fan-out graphs. **(b, c)** Bytes communicated per iteration for each abstraction on power-law fan-in and fan-out graphs.




(a) Power-law Fan-In Runtime (b) Power-law Fan-Out Runtime

Figure 10: **Synthetic Experiments Runtime. (a, b)** Per iteration runtime of each abstraction on synthetic power-law graphs.

vertex-cut. As a consequence we expect that PowerGraph will reduce communication volume.

In Fig. 9c and Fig. 9d we plot the bytes communicated per iteration for all three systems under power-law fan-in and fan-out graphs. Because Pregel only sends messages along out-edges, Pregel communicates more on power-law fan-out graphs than on power-law fan-in graphs.

On the other hand, GraphLab and PowerGraph's communication volume is invariant to power-law fan-in and fan-out since neither considers edge direction during data-synchronization. However, PowerGraph communicates significantly less than GraphLab which is a direct result of the efficacy of vertex cuts. Finally, PowerGraph's total communication increases only marginally on the denser graphs and is the lowest overall.

## 6.3 Runtime Comparison

PowerGraph significantly out-performs GraphLab and Pregel on low $\alpha$ graphs. In Fig. 10a and Fig. 10b we plot the per iteration runtime for each abstraction. In both cases the overall runtime performance closely matches the communication overhead ( Fig. 9c and Fig. 9d) while the computation imbalance (Fig. 9a and Fig. 9b) appears to have little effect. The limited effect of imbalance is due to the relatively lightweight nature of the PageRank computation and we expect more complex algorithms (e.g., statistical inference) to be more susceptible to imbalance. However, when greedy (coordinated) partitioning is used we see an additional 25% to 50% improvement in runtime.

# 7 Implementation and Evaluation

In this section, we describe and evaluate our implementation of the PowerGraph system. All experiments are performed on a 64 node cluster of Amazon EC2 `cc1.4xlarge` Linux instances. Each instance has two quad core Intel Xeon X5570 processors with 23GB of RAM, and is connected via 10 GigE Ethernet. PowerGraph was written in C++ and compiled with GCC 4.5.

We implemented three variations of the PowerGraph abstraction. To demonstrate their relative implementation complexity, we provide the line counts, excluding common support code:

**Bulk Synchronous (Sync):** A fully synchronous implementation of PowerGraph as described in Sec. 4.3.1. [600 lines]

**Asynchronous (Async):** An asynchronous implementation of PowerGraph which allows arbitrary interleaving of vertex-programs Sec. 4.3.2. [900 lines]

**Asynchronous Serializable (Async+S):** An asynchronous implementation of PowerGraph which guarantees serializability of *all* vertex-programs (equivalent to "edge consistency" in GraphLab). [1600 lines]

In all cases the system is entirely symmetric with no single coordinating instance or scheduler. Each instances is given the list of other machines and start by reading a unique subset of the graph data files from HDFS. TCP connections are opened with other machines as needed to build the distributed graph and run the engine.

## 7.1 Graph Loading and Placement

The graph structure and data are loaded from a collection of text files stored in a distributed file-system (HDFS) by all instances in parallel. Each machine loads a separate subset of files (determined by hashing) and applies one of the three distributed graph partitioning algorithms to place the data *as it is loaded*. As a consequence partitioning is accomplished in parallel and data is immediately placed in its final location. Unless specified, all experiments were performed using the oblivious algorithm. Once computation is complete, the final vertex and edge data are saved back to the distributed file-system in parallel.

In Fig. 7b, we evaluate the performance of a collection of algorithms varying the partitioning procedure. Our simple partitioning heuristics are able to improve performance significantly across *all algorithms*, decreasing runtime and memory utilization. Furthermore, the runtime scales *linearly* with the replication factor: halving the replication factor approximately halves runtime.

## 7.2 Synchronous Engine (Sync)

Our synchronous implementation closely follows the description in Sec. 4.3.1. Each machine runs a single multi-threaded instance to maximally utilize the multi-core architecture. We rely on background communication to achieve computation/communication interleaving. The synchronous engine's fully deterministic execution makes it easy to reason about programmatically and minimizes effort needed for tuning and performance optimizations.

In Fig. 11a and Fig. 11b we plot the runtime and total communication of one iteration of PageRank on the Twitter follower network for each partitioning method. To provide a point of comparison (Tab. 2), the Spark [37] framework computes one iteration of PageRank on the same graph in 97.4s on a 50 node-100 core cluster [35]. PowerGraph is therefore between 3-8x faster than Spark on a comparable number of cores. On the full cluster of 512 cores, we can compute one iteration in 3.6s.

The greedy partitioning heuristics improves both performance and scalability of the engine at the cost of increased load-time. The load time for random, oblivious, and coordinated placement were 59, 105, and 239 seconds respectively. While greedy partitioning heuristics increased load-time by up to a factor of four, they still improve overall runtime if more than 20 iterations of PageRank are performed. In Fig. 11c we plot the runtime of each iteration of PageRank on the Twitter follower network. Delta caching improves performance by avoiding unnecessary gather computation, decreasing total runtime by 45%. Finally, in Fig. 11d we evaluate weak-scaling: ability to scale while keeping the problem size per processor constant. We run SSSP (Fig. 3) on synthetic power-law graphs ($\alpha = 2$), with ten-million vertices per machine. Our implementation demonstrates nearly optimal weak-scaling and requires only 65s to solve a 6.4B edge graph.

## 7.3 Asynchronous Engine (Async)

We implemented the asynchronous PowerGraph execution model (Sec. 4.3.2) using a simple state machine for each vertex which can be either: `INACTIVE`, `GATHER`, `APPLY` or `SCATTER`. Once activated, a vertex enters the gathering state and is placed in a *local* scheduler which assigns cores to active vertices allowing many vertex-programs to run simultaneously thereby hiding communication latency. While arbitrary interleaving of vertex programs is permitted, we avoid data races by ensuring that individual gather, apply, and scatter calls have exclusive access to their arguments.

We evaluate the performance of the Async engine by running PageRank on the Twitter follower network. In Fig. 12a, we plot throughput (number of vertex-program operations per second) against the number of machines. Throughput increases moderately with both the number of machines as well as improved partitioning. We evaluate the gains associated with delta caching (Sec. 4.2) by measuring throughput as a function of time (Fig. 12b) with caching enabled and with caching disabled. Caching allows the algorithm to converge faster with fewer operations. Surprisingly, when caching is disabled, the throughput increases over time. Further analysis reveals that the computation gradually focuses on high-degree vertices, increasing the computation/communication ratio.

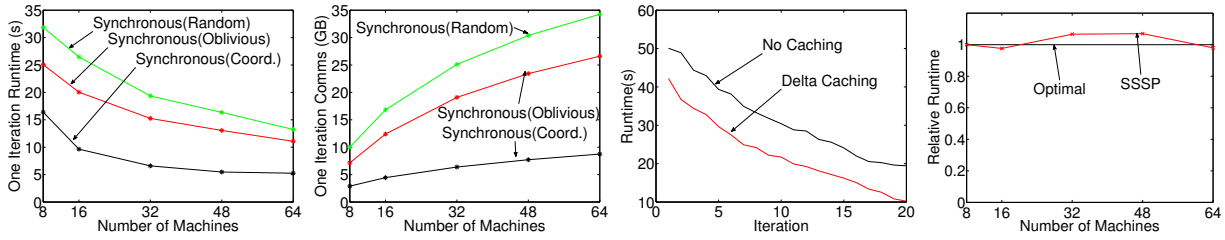
We evaluate the graph coloring vertex-program (Fig. 3) which cannot be run synchronously since all vertices would change to the same color on every iteration. Graph coloring is a proxy for many MLDM algorithms [17]. In Fig. 12c we evaluate weak-scaling on synthetic power-law graphs ($\alpha = 2$) with five-million vertices per machine and find that the Async engine performs nearly optimally. The slight increase in runtime may be attributed to an increase in the number of colors due to increasing graph size.

## 7.4 Async. Serializable Engine (Async+S)

The Async engine is useful for a broad range of tasks, providing high throughput and performance. However, unlike the synchronous engine, the asynchronous engine is difficult to reason about programmatically. We therefore extended the Async engine to enforce serializability.

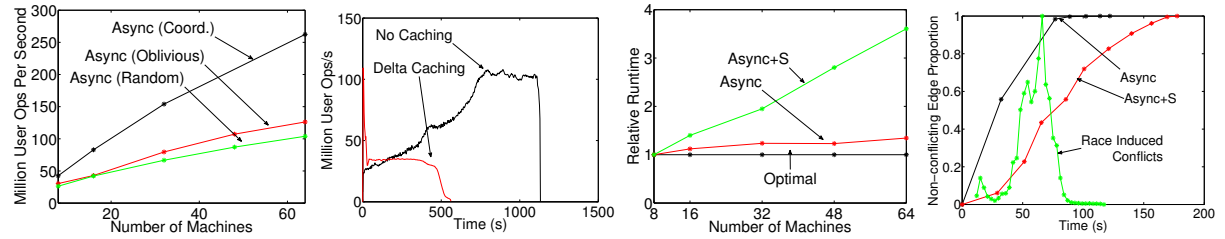
The Async+S engine ensures serializability by preventing adjacent vertex-programs from running simultaneously. Ensuring serializability for graph-parallel computation is equivalent to solving the dining philosophers problem where each vertex is a philosopher, and each edge is a fork. GraphLab [29] implements Dijkstra's solution [14] where forks are acquired *sequentially* according to a total ordering. Instead, we implement the Chandy-Misra solution [10] which acquires all forks simultaneously, permitting a high degree of parallelism. We extend the Chandy-Misra solution to the vertex-cut setting by enabling each vertex replica to request only forks for local edges and using a simple consensus protocol to establish when all replicas have succeeded.

We evaluate the scalability and computational efficiency of the Async+S engine on the graph coloring task. We observe in Fig. 12c that the amount of achieved parallelism does not increase linearly with the number of vertices. Because the density (i.e., contention) of power-law graphs increases super-linearly with the number of




(a) Twitter PageRank Runtime (b) Twitter PageRank Comms (c) Twitter PageRank Delta Cache (d) SSSP Weak Scaling

Figure 11: **Synchronous Experiments (a,b)** Synchronous PageRank Scaling on Twitter graph. **(c)** The PageRank per iteration runtime on the Twitter graph with and without delta caching. **(d)** Weak scaling of SSSP on synthetic graphs.



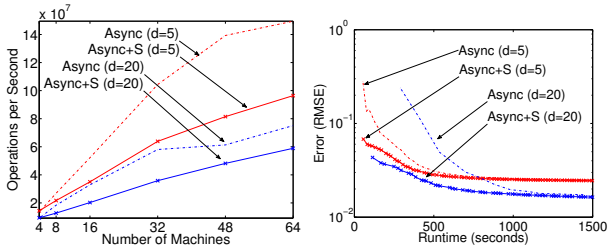

(a) Twitter PageRank Throughput (b) Twitter PageRank Delta Cache (c) Coloring Weak Scaling (d) Coloring Conflict Rate

Figure 12: **Asynchronous Experiments (a)** Number of user operations (gather/apply/scatter) issued per second by Dynamic PageRank as # machines is increased. **(b)** Total number of user ops with and without caching plotted against time. **(c)** Weak scaling of the graph coloring task using the Async engine and the Async+S engine **(d)** Proportion of non-conflicting edges across time on a 8 machine, 40M vertex instance of the problem. The green line is the rate of conflicting edges introduced by the lack of consistency (peak 236K edges per second) in the Async engine. When the Async+S engine is used no conflicting edges are ever introduced.

vertices, we do not expect the amount of serializable parallelism to increase linearly.

In Fig. 12d, we plot the proportion of edges that satisfy the coloring condition (both vertices have different colors) for both the Async and the Async+S engines. While the Async engine quickly satisfies the coloring condition for most edges, the remaining 1% take 34% of the runtime. We attribute this behavior to frequent races on tightly connected vertices. Alternatively, the Async+S engine performs more uniformly. If we examine the total number of user operations we find that the Async engine does more than *twice* the work of the Async+S engine.

Finally, we evaluate the Async and the Async+S engines on a popular machine learning algorithm: Alternating Least Squares (ALS). The ALS algorithm has a number of variations which allow it to be used in a wide range of applications including user personalization [38] and document semantic analysis [21]. We apply ALS to the Wikipedia term-document graph consisting of 11M vertices and 315M edges to extract a mixture of topics representation for each document and term. The number of topics $d$ is a free parameter that determines the computational complexity $O(d^3)$ of each vertex-program. In Fig. 13a, we plot the ALS throughput on the Async engine and the Async+S engine. While the throughput of the Async engine is greater, the gap between engines shrinks as $d$ increases and computation dominates the consistency overhead. To demonstrate the importance of serializability, we plot in Fig. 13b the training error, a measure of solution quality, for both engines. We observe that while the Async engine has greater throughput, the Async+S engine *converges* faster.




(a) ALS Throughput (b) ALS Convergence

Figure 13: **(a)** The throughput of ALS measured in millions of User Operations per second. **(b)** Training error (lower is better) as a function of running time for ALS application.

The complexity of the Async+S engine is justified by the necessity for serializability in many applications (e.g., ALS). Furthermore, serializability adds predictability to the nondeterministic asynchronous execution. For example, even graph coloring may not terminate on dense graphs unless serializability is ensured.

## 7.5 Fault Tolerance

Like GraphLab and Pregel, PowerGraph achieves fault-tolerance by saving a snapshot of the data-graph. The synchronous PowerGraph engine constructs the snapshot between super-steps and the asynchronous engine suspends

| **PageRank** | Runtime | $\|V\|$ | $\|E\|$ | System |
|---|---|---|---|---|
| Hadoop [22] | 198s | – | 1.1B | 50x8 |
| Spark [37] | 97.4s | 40M | 1.5B | 50x2 |
| Twister [15] | 36s | 50M | 1.4B | 64x4 |
| *PowerGraph (Sync)* | 3.6s | 40M | 1.5B | 64x8 |

| **Triangle Count** | Runtime | $\|V\|$ | $\|E\|$ | System |
|---|---|---|---|---|
| Hadoop [36] | 423m | 40M | 1.4B | 1636x? |
| *PowerGraph (Sync)* | 1.5m | 40M | 1.4B | 64x16 |

| **LDA** | Tok/sec | Topics | System |
|---|---|---|---|
| *Smola et al.* [34] | 150M | 1000 | 100x8 |
| *PowerGraph (Async)* | 110M | 1000 | 64x16 |

Table 2: Relative performance of PageRank, triangle counting, and LDA on similar graphs. PageRank runtime is measured per iteration. Both PageRank and triangle counting were run on the Twitter follower network and LDA was run on Wikipedia. The systems are reported as number of nodes by number of cores.

execution to construct the snapshot. An asynchronous snapshot using GraphLab's snapshot algorithm [29] can also be implemented. The checkpoint overhead, typically a few seconds for the largest graphs we considered, is small relative to the running time of each application.

### 7.6 MLDM Applications

In Tab. 2 we provide comparisons of the PowerGraph system with published results on similar data for PageRank, Triangle Counting [36], and collapsed Gibbs sampling for the LDA model [34]. The PowerGraph implementations of PageRank and Triangle counting are one to two orders of magnitude faster than published results. For LDA, the state-of-the-art solution is a heavily optimized system designed for this specific task by Smola et al. [34]. In contrast, PowerGraph is able to achieve comparable performance using only 200 lines of user code.

## 8 Related Work

The vertex-cut approach to distributed graph placement is related to work [9, 13] in hypergraph partitioning. In particular, a vertex-cut problem can be cast as a hypergraph-cut problem by converting each edge to a vertex, and each vertex to a hyper-edge. However, existing hypergraph partitioning can be very time intensive. While our cut objective is similar to the "communication volume" objective, the streaming vertex cut setting described in this paper is novel. Stanton et al, in [35] developed several heuristics for the streaming edge-cuts but do not consider the vertex-cut problem.

Several [8, 22] have proposed generalized sparse matrix vector multiplication as a basis for graph-parallel computation. These abstractions operate on commutative associative semi-rings and therefore also have generalized gather and sum operations. However, they do not support the more general apply and scatter operations, as well as mutable edge-data and are based on a strictly synchronous model in which all computation is run in every iteration.

While we discuss Pregel and GraphLab in detail, there are other similar graph-parallel abstractions. Closely related to Pregel is BPGL [20] which implements a synchronous traveler model. Alternatively, Kineograph [11] presents a graph-parallel framework for time-evolving graphs which mixes features from both GraphLab and Piccolo. Pujol et al. [33] present a distributed graph database but do not explicitly consider the power-law structure. Finally, [25] presents GraphChi: an efficient single-machine disk-based implementation of the GraphLab abstraction. Impressively, it is able to significantly out-perform large Hadoop deployments on many graph problems while using only a single machine: performing one iteration of PageRank on the Twitter Graph in only 158s (PowerGraph: 3.6s). The techniques described in GraphChi can be used to add out-of-core storage to PowerGraph.

## 9 Conclusions and Future Work

The need to reason about large-scale graph-structured data has driven the development of new graph-parallel abstractions such as GraphLab and Pregel. However graphs derived from real-world phenomena often exhibit power-law degree distributions, which are difficult to partition and can lead to work imbalance and substantially increased communication and storage.

To address these challenges, we introduced the PowerGraph abstraction which exploits the **G**ather-**A**pply-**S**catter model of computation to factor vertex-programs over edges, splitting high-degree vertices and exposing greater parallelism in natural graphs. We then introduced vertex-cuts and a collection of fast greedy heuristics to substantially reduce the storage and communication costs of large distributed power-law graphs. We theoretically related the power-law constant to the communication and storage requirements of the PowerGraph system and empirically evaluate our analysis by comparing against GraphLab and Pregel. Finally, we evaluate the PowerGraph system on several large-scale problems using a 64 node EC2 cluster and demonstrating the scalability and efficiency and in many cases order of magnitude gains over published results.

We are actively using PowerGraph to explore new large-scale machine learning algorithms. We are beginning to study how vertex replication and data-dependencies can be used to support fault-tolerance without checkpointing. In addition, we are exploring ways to support time-evolving graph structures. Finally, we believe that many of the core ideas in the PowerGraph abstraction can have a significant impact in the design and implementation of graph-parallel systems beyond PowerGraph.

**Acknowledgments**

This work is supported by the ONR Young Investigator Program grant N00014-08-1-0752, the ARO under MURI W911NF0810242, the ONR PECASE-N00014-10-1-0672, the National Science Foundation grant IIS-0803333 as well as the Intel Science and Technology Center for Cloud Computing. Joseph Gonzalez is supported by the Graduate Research Fellowship from the NSF. We would like to thank Alex Smola, Aapo Kyrola, Lidong Zhou, and the reviewers for their insightful guidance.

## References

[1] ABOU-RJEILI, A., AND KARYPIS, G. Multilevel algorithms for partitioning power-law graphs. In *IPDPS* (2006).

[2] AHMED, A., ALY, M., GONZALEZ, J., NARAYANAMURTHY, S., AND SMOLA, A. J. Scalable inference in latent variable models. In *WSDM* (2012), pp. 123–132.

[3] ALBERT, R., JEONG, H., AND BARABÁSI, A. L. Error and attack tolerance of complex networks. In *Nature* (2000), vol. 406, pp. 378—482.

[4] BERTSEKAS, D. P., AND TSITSIKLIS, J. N. *Parallel and distributed computation: numerical methods*. Prentice-Hall, 1989.

[5] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW* (2011), pp. 587–596.

[6] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *WWW* (2004), pp. 595–601.

[7] BORDINO, I., BOLDI, P., DONATO, D., SANTINI, M., AND VIGNA, S. Temporal evolution of the uk web. In *ICDM Workshops* (2008), pp. 909–918.

[8] BULUÇ, A., AND GILBERT, J. R. The combinatorial blas: design, implementation, and applications. *IJHPCA 25*, 4 (2011), 496–509.

[9] CATALYUREK, U., AND AYKANAT, C. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *IRREGULAR* (1996), pp. 75–86.

[10] CHANDY, K. M., AND MISRA, J. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst. 6*, 4 (Oct. 1984), 632–646.

[11] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys* (2012), pp. 85–98.

[12] CHIERICHETTI, F., KUMAR, R., LATTANZI, S., MITZENMACHER, M., PANCONESI, A., AND RAGHAVAN, P. On compressing social networks. In *KDD* (2009), pp. 219–228.

[13] DEVINE, K. D., BOMAN, E. G., HEAPHY, R. T., BISSELING, R. H., AND CATALYUREK, U. V. Parallel hypergraph partitioning for scientific computing. In *IPDPS* (2006).

[14] DIJKSTRA, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1* (1971), 115–138.

[15] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S., QIU, J., AND FOX, G. Twister: A runtime for iterative MapReduce. In *HPDC* (2010), ACM.

[16] FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review 29*, 4 (1999), 251–262.

[17] GONZALEZ, J., LOW, Y., GRETTON, A., AND GUESTRIN, C. Parallel gibbs sampling: From colored fields to thin junction trees. In *AISTATS* (2011), vol. 15, pp. 324–332.

[18] GONZALEZ, J., LOW, Y., AND GUESTRIN, C. Residual splash for optimally parallelizing belief propagation. In *AISTATS* (2009), vol. 5, pp. 177–184.

[19] GONZALEZ, J., LOW, Y., GUESTRIN, C., AND O'HALLARON, D. Distributed parallel inference on large factor graphs. In *UAI* (2009).

[20] GREGOR, D., AND LUMSDAINE, A. The parallel BGL: A generic library for distributed graph computations. *POOSC* (2005).

[21] HOFMANN, T. Probabilistic latent semantic indexing. In *SIGIR* (1999), pp. 50–57.

[22] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM* (2009), pp. 229 –238.

[23] KARYPIS, G., AND KUMAR, V. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput. 48*, 1 (1998), 96–129.

[24] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is twitter, a social network or a news media? In *WWW* (2010), pp. 591–600.

[25] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *OSDI* (2012).

[26] LANG, K. Finding good nearly balanced cuts in power law graphs. Tech. Rep. YRL-2004-036, Yahoo! Research Labs, Nov. 2004.

[27] LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data 1*, 1 (mar 2007).

[28] LESKOVEC, J., LANG, K. J., DASGUPTA, A., , AND MAHONEY, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1 (2008), 29–123.

[29] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB* (2012).

[30] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD* (2010).

[31] PELLEGRINI, F., AND ROMAN, J. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN Europe* (1996), pp. 493–498.

[32] POWER, R., AND LI, J. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI* (2010).

[33] PUJOL, J. M., ERRAMILLI, V., SIGANOS, G., YANG, X., LAOUTARIS, N., CHHABRA, P., AND RODRIGUEZ, P. The little engine(s) that could: scaling online social networks. In *SIGCOMM* (2010), pp. 375–386.

[34] SMOLA, A. J., AND NARAYANAMURTHY, S. An Architecture for Parallel Topic Models. *PVLDB 3*, 1 (2010), 703–710.

[35] STANTON, I., AND KLIOT, G. Streaming graph partitioning for large distributed graphs. Tech. Rep. MSR-TR-2011-121, Microsoft Research, November 2011.

[36] SURI, S., AND VASSILVITSKII, S. Counting triangles and the curse of the last reducer. In *WWW* (2011), pp. 607–614.

[37] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[38] ZHOU, Y., WILKINSON, D., SCHREIBER, R., AND PAN, R. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM* (2008), pp. 337–348.

# GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrola
Carnegie Mellon University
akyrola@cs.cmu.edu

Guy Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Carlos Guestrin
University of Washington
guestrin@cs.washington.edu

## Abstract

Current systems for graph computation require a distributed computing cluster to handle very large real-world problems, such as analysis on social networks or the web graph. While distributed computational resources have become more accessible, developing distributed graph algorithms still remains challenging, especially to non-experts.

In this work, we present GraphChi, a **disk-based system** for computing efficiently on graphs with billions of edges. By using a well-known method to break large graphs into small parts, and a novel **parallel sliding windows** method, GraphChi is able to execute several advanced data mining, graph mining, and machine learning algorithms on very large graphs, using just a single consumer-level computer. We further extend GraphChi to support graphs that **evolve over time**, and demonstrate that, on a single computer, GraphChi can process over one hundred thousand graph updates per second, while simultaneously performing computation. We show, through experiments and theoretical analysis, that GraphChi performs well on both SSDs and rotational hard drives.

By repeating experiments reported for existing distributed systems, we show that, with only fraction of the resources, GraphChi can solve the same problems in very reasonable time. Our work makes large-scale graph computation available to anyone with a modern PC.

## 1 Introduction

Designing scalable systems for analyzing, processing and mining huge real-world graphs has become one of the most timely problems facing systems researchers. For example, social networks, Web graphs, and protein interaction graphs are particularly challenging to handle, because they cannot be readily decomposed into small parts that could be processed in parallel. This lack of data-parallelism renders MapReduce [19] inefficient for computing on such graphs, as has been argued by many researchers (for example, [13, 30, 31]). Consequently, in recent years several graph-based abstractions have been proposed, most notably Pregel [31] and GraphLab [30]. Both use a vertex-centric computation model, in which the user defines a program that is executed locally for each vertex in parallel. In addition, high-performance systems that are based on key-value tables, such as Piccolo [36] and Spark [45], can efficiently represent many graph-parallel algorithms.

Current graph systems are able to scale to graphs of billions of edges by distributing the computation. However, while distributed computional resources are now available easily through the Cloud, efficient large-scale computation on graphs still remains a challenge. To use existing graph frameworks, one is faced with the challenge of partitioning the graph across cluster nodes. Finding efficient graph cuts that minimize communication between nodes, and are also balanced, is a hard problem [27]. More generally, distributed systems and their users must deal with managing a cluster, fault tolerance, and often unpredictable performance. From the perspective of programmers, debugging and optimizing distributed algorithms is hard.

Our frustration with distributed computing provoked us to ask a question: Would it be possible to do advanced graph computation on just a personal computer? Handling graphs with billions of edges in memory would require tens or hundreds of gigabytes of DRAM, currently only available to high-end servers, with steep prices [4]. This leaves us with only one option: to use persistent storage as memory extension. Unfortunately, processing large graphs efficiently from disk is a hard problem, and generic solutions, such as systems that extend main memory by using SSDs, do not perform well.

To address this problem, we propose a novel method, Parallel Sliding Windows (PSW), for processing very large graphs from disk. PSW requires only a very small number of non-sequential accesses to the disk, and thus performs well on both SSDs and traditional hard drives. Surprisingly, unlike most distributed frameworks, PSW naturally implements the **asynchronous** model of computation, which has been shown to be more efficient than synchronous computation for many purposes [7, 29].

We further extend our method to graphs that are continu-

ously evolving. This setting was recently studied by Cheng et. al., who proposed Kineograph [15], a distributed system for processing a continuous in-flow of graph updates, while simultaneously running advanced graph mining algorithms. We implement the same functionality, but using only a single computer, by applying techniques developed by the I/O-efficient algorithm researchers [42].

We further present a complete system, GraphChi, which we used to solve a wide variety of computational problems on extremely large graphs, efficiently on a single consumer-grade computer. In the evolving graph setting, GraphChi is able to ingest over a hundred thousand new edges per second, while simultaneously executing computation.

The outline of our paper is as follows. We introduce the computational model and challenges for the external memory setting in Section 2. The Parallel Sliding Windows method is described in Section 3, and GraphChi system design and implementation is outlined in Section 4. We evaluate GraphChi on very large problems (graphs with billions of edges), using a set of algorithms from graph mining, machine learning, collaborative filtering, and sparse linear algebra (Sections 6 and 7).

Our contributions:

- The Parallel Sliding Windows, a method for processing large graphs from disk (both SSD and hard drive), with theoretical guarantees.
- Extension to evolving graphs, with the ability to ingest efficiently a stream of graph changes, while simultaneously executing computation.
- System design, and evaluation of a C++ implementation of GraphChi. We demonstrate GraphChi's ability to solve such large problems, which were previously only possible to solve by cluster computing. Complete source-code for the system and applications is released in open source: `http://graphchi.org`.

# 2 Disk-based Graph Computation

In this section, we start by describing the computational setting of our work, and continue by arguing why straightforward solutions are not sufficient.

## 2.1 Computational Model

We now briefly introduce the vertex-centric model of computation, explored by GraphLab [30] and Pregel [31]. A problem is encoded as a directed (sparse) graph, $G = (V, E)$. We associate a value with each vertex $v \in V$, and each edge $e = (source, destination) \in E$. We assume that the vertices are labeled from 1 to $|V|$. Given a directed edge $e = (u, v)$, we refer to $e$ as vertex $v$'s **in-edge**, and as vertex $u$'s **out-edge**.

**Algorithm 1**: Typical vertex **update-function**

```
1 Update(vertex) begin
2     x[] ← read values of in- and out-edges of vertex ;
3     vertex.value ← f(x[]) ;
4     foreach edge of vertex do
5         edge.value ← g(vertex.value, edge.value);
6     end
7 end
```

To perform computation on the graph, programmer specifies an **update-function(v)**, which can access and modify the value of a vertex and its incident edges. The update-function is executed for each of the vertices, iteratively, until a termination condition is satisfied.

Algorithm 1 shows the high-level structure of a typical update-function. It first computes some value $f(x[])$ based on the values of the edges, and assigns $f(x[])$ (perhaps after a transformation) as the new value of the vertex. Finally, the edges will be assigned new values based on the new vertex value and the previous value of the edge.

As shown by many authors [15, 29, 30, 31], the vertex-centric model can express a wide range of problems, for example, from the domains of graph mining, data mining, machine learning, and sparse linear algebra.

Most existing frameworks execute update functions in lock-step, and implement the **Bulk-Synchronous Parallel** (BSP) model [41], which defines that update-functions can only observe values from the previous iteration. BSP is often preferred in distributed systems as it is simple to implement, and allows maximum level of parallelism during the computation. However, after each iteration, a costly synchronization step is required and system needs to store two versions of all values (value of previous iteration and the new value).

Recently, many researchers have studied the **asynchronous** model of computation. In this setting, an update-function is able to to use the *most recent* values of the edges and the vertices. In addition, the ordering (scheduling) of updates can be dynamic. Asynchronous computation accelerates convergence of many numerical algorithms; in some cases BSP fails to converge at all [7, 30]. The Parallel Sliding Windows method, which is the topic of this work, implements the asynchronous[1] model and exposes updated values immediately to subsequent computation. Our implementation, GraphChi, also supports dynamic **se-**

[1] In the context of iterative solvers for linear systems, asynchronous computation is called the Gauss-Seidel method.

**lective scheduling**, allowing update-functions and graph modifications to *enlist* vertices to be updated[2].

#### 2.1.1 Computational Constraints

We state the memory requirements informally. We assume a computer with limited memory (DRAM) capacity:

1. The graph structure, edge values, and vertex values do not fit into memory. In practice, we assume the amount of memory to be only a small fraction of the memory required for storing the complete graph.
2. There is enough memory to contain the edges and their associated values of any *single* vertex in the graph.

To illustrate that it is often infeasible to even store just vertex values in memory, consider the *yahoo-web* graph with 1.7 billion vertices [44]. Associating a floating point value for each vertex would require almost 7 GB of memory, too much for many current PCs (spring 2012). While we expect the memory capacity of personal computers to grow in the future, the datasets are expected to grow quickly as well.

## 2.2 Standard Sparse Graph Formats

The system by Pearce et al. [34] uses *compressed sparse row* (CSR) storage format to store the graph on disk, which is equivalent to storing the graph as adjacency sets: the out-edges of each vertex are stored consecutively in the file. In addition, indices to the adjacency sets for each vertex are stored. Thus, CSR allows for fast loading of *out-edges* of a vertex from the disk.

However, in the vertex-centric model we also need to access the *in-edges* of a vertex. This is very inefficient under CSR: in-edges of a vertex can be arbitrarily located in the adjacency file, and a full scan would be required for retrieving in-edges for any given vertex. This problem can be solved by representing the graph simultaneously in the *compressed sparse column* (CSC) format. CSC format is simply CSR for the transposed graph, and thus allows fast sequential access to the in-edges for vertices. In this solution, each edge is stored twice.

## 2.3 Random Access Problem

Unfortunately, simply storing the graph simultaneously in CSR and CSC does not enable efficient modification of the edge values. To see this, consider an edge $e = (v, w)$, with value $x$. Let now an update of vertex $v$ change its value to $x'$. Later, when vertex $w$ is updated, it should observe its in-edge $e$ with value $x'$. Thus, either 1) when the set of in-edges of $w$ are read, the new value $x'$ must be read from the the set of out-edges of $v$ (stored under CSR); or 2) the modification $x \Rightarrow x'$ has to be written to the in-edge list (under CSC) of vertex $w$. The first solution incurs a *random read*, and latter a *random write*. If we assume, realistically, that most of the edges are modified in a pass over the graph, either $O(|E|)$ of random reads or $O(|E|)$ random writes would be performed – a huge number on large graphs.

In many algorithms, the value of a vertex only depends on its neighbors' values. In that case, if the computer has enough memory to store all the vertex values, this problem is not relevant, and the system by Pearce et al. [34] is sufficient (on an SSD). On the other hand, if the vertex values would be stored on disk, we would encounter the same random access problem when accessing values of the neighbors.

#### 2.3.1 Review of Possible Solutions

Prior to presenting our solution to the problem, we discuss some alternative strategies and why they are not sufficient.

*SSD as a memory extension.* SSD provides relatively good random read and sequential write performance, and many researchers have proposed using SSD as an extension to the main memory. SSDAlloc [4] presents the current state-of-the-art of these solutions. It enables transparent usage of SSD as heap space, and uses innovative methods to implement object-level caching to increase sequentiality of writes. Unfortunately, for the huge graphs we study, the number of very small objects (vertices or edges) is extremely large, and in most cases, the amounts of writes and reads made by a graph algorithm are roughly equal, rendering caching inefficient. SSDAlloc is able to serve some tens of thousands of random reads or writes per second [4], which is insufficient, as GraphChi can access millions of edges per second.

*Exploiting locality.* If related edges appear close to each other on the disk, the amount of random disk access could be reduced. Indeed, many real-world graphs have a substantial amount of inherent locality. For example, webpages are clustered under domains, and people have more connections in social networks inside their geographical region than outside it [27]. Unfortunately, the locality of real-world graphs is limited, because the number of edges crossing local clusters is also large [27]. As real-world graphs have typically a very skewed vertex degree distribution, it would make sense to cache high-degree vertices (such as important websites) in memory, and process the

[2] BSP can be applied with GraphChi in the asynchronous model by storing two versions of each value.

rest of the graph from disk.

In the early phase of our project, we explored this option, but found it difficult to find a good cache policy to sufficiently reduce disk access. Ultimately, we rejected this approach for two reasons. First, the performance would be highly unpredictable, as it would depend on structural properties of the input graph. Second, optimizing graphs for locality is costly, and sometimes impossible, if a graph is supplied without metadata required to efficiently cluster it. General graph partitioners are not currently an option, since even the state-of-the-art graph partitioner, METIS [25], requires hundreds of gigabytes of memory to work with graphs of billions of edges.

*Graph compression.* Compact representation of real-world graphs is a well-studied problem, the best algorithms can store web-graphs in only 4 bits/edge (see [8, 12, 17, 23]). Unfortunately, while the graph *structure* can often be compressed and stored in memory, we also associate data with each of the edges and vertices, which can take significantly more space than the graph itself.

*Bulk-Synchronous Processing.* For a synchronous system, the random access problem can be solved by writing updated edges into a scratch file, which is then sorted (using disk-sort), and used to generate input graph for next iteration. For algorithms that modify only the vertices, not edges, such as Pagerank, a similar solution has been used [14]. However, it cannot be efficiently used to perform asynchronous computation.

# 3 Parallel Sliding Windows

This section describes the Parallel Sliding Windows (PSW) method (Algorithm 2). PSW can process a graph with mutable edge values efficiently from disk, with only a small number of non-sequential disk accesses, while supporting the asynchronous model of computation. PSW processes graphs in three stages: it 1) loads a subgraph from disk; 2) updates the vertices and edges; and 3) writes the updated values to disk. These stages are explained in detail below, with a concrete example. We then present an extension to graphs that evolve over time, and analyze the I/O costs of the PSW method.

## 3.1 Loading the Graph

Under the PSW method, the vertices $V$ of graph $G = (V, E)$ are split into $P$ disjoint **intervals**. For each interval, we associate a **shard**, which stores all the edges that have *destination* in the interval. Edges are stored in the order of their *source* (Figure 1). Intervals are chosen to balance the number of edges in each shard; the number of intervals, $P$, is chosen so that any one shard can be loaded completely into memory. Similar data layout for sparse graphs was used previously, for example, to implement I/O efficient Pagerank and SpMV [5, 21].

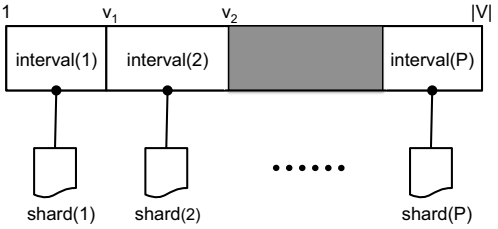



Figure 1: The vertices of graph $(V, E)$ are divided into $P$ intervals. Each interval is associated with a shard, which stores all edges that have destination vertex in that interval.

PSW does graph computation in **execution intervals**, by processing vertices one interval at a time. To create the subgraph for the vertices in interval $p$, their edges (with their associated values) must be loaded from disk. First, **Shard(p)**, which contains the *in-edges* for the vertices in interval(p), is loaded fully into memory. We call thus shard(p) the **memory-shard**. Second, because the edges are ordered by their source, the *out-edges* for the vertices are stored in consecutive chunks in the other shards, requiring additional $P - 1$ block reads. Importantly, edges for interval(p+1) are stored immediately after the edges for interval(p). Intuitively, when PSW moves from an interval to the next, it *slides* a **window** over each of the shards. We call the other shards the **sliding shards**. Note, that if the degree distribution of a graph is not uniform, the window length is variable. In total, PSW requires only $P$ sequential disk reads to process each interval. A high-level illustration of the process is given in Figure 2, and the pseudo-code of the subgraph loading is provided in Algorithm 3.

## 3.2 Parallel Updates

After the subgraph for interval $p$ has been fully loaded from disk, PSW executes the user-defined **update-function** for each vertex *in parallel*. As update-functions can modify the edge values, to prevent adjacent vertices from accessing edges concurrently (race conditions), we enforce *external determinism*, which guarantees that each execution of PSW produces exactly the same result. This guarantee is straightforward to implement: vertices that have edges with both end-points in the same interval are flagged as *critical*, and are updated in sequential order. Non-critical vertices do not share edges with other vertices in the interval, and can be updated safely in parallel. Note, that the update of a critical vertex will observe changes in edges done by

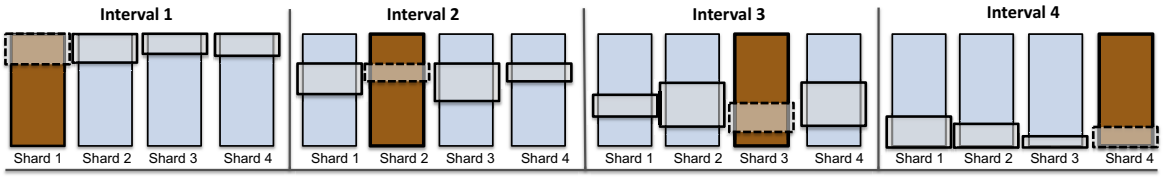



Figure 2: Visualization of the stages of one iteration of the Parallel Sliding Windows method. In this example, vertices are divided into four intervals, each associated with a shard. The computation proceeds by constructing a subgraph of vertices one interval a time. In-edges for the vertices are read from the **memory-shard** (in dark color) while out-edges are read from each of the **sliding shards**. The current **sliding window** is pictured on top of each shard.

**Algorithm 2**: Parallel Sliding Windows (PSW)

```
1  foreach iteration do
2      shards[] ← InitializeShards(P)
3      for interval ← 1 to P do
4          /* Load subgraph for interval, using Alg. 3. Note,
           that the edge values are stored as pointers to the
           loaded file blocks. */
5          subgraph ← LoadSubgraph(interval)
6          parallel foreach vertex ∈ subgraph.vertex do
7              /* Execute user-defined update function,
8               which can modify the values of the edges */
9              UDF_updateVertex(vertex)
10         end
11         /* Update memory-shard to disk */
12         shards[interval].UpdateFully()
13         /* Update sliding windows on disk */ for
           s ∈ 1, .., P, s ≠ interval do
14             shards[s].UpdateLastWindowToDisk()
15         end
16     end
17 end
```

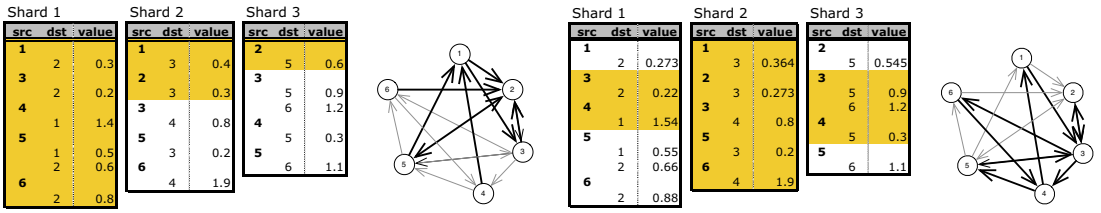
**Algorithm 3**: Function LoadSubGraph(p)

```
   Input : Interval index number p
   Result: Subgraph of vertices in the interval p
1  /* Initialization */
2  a ← interval[p].start
3  b ← interval[p].end
4  G ← InitializeSubgraph(a, b)
5  /* Load edges in memory-shard. */
6  edgesM ← shard[p].readFully()
7  /* Evolving graphs: Add edges from buffers. */
8  edgesM ← edgesM ∪ shard[p].edgebuffer[1..P]
9  foreach e ∈ edgesM do
10     /* Note: edge values are stored as pointers. */
11     G.vertex[edge.dest].addInEdge(e.source, &e.val)
12     if e.source ∈ [a, b] then
13         G.vertex[edge.source].addOutEdge(e.dest, &e.val)
14     end
15 end
16 /* Load out-edges in sliding shards. */
17 for s ∈ 1, .., P, s ≠ p do
18     edgesS ← shard[s].readNextWindow(a, b)
19     /* Evolving graphs: Add edges from shard's buffer p */
20     edgesS ← edgesS ∪ shard[s].edgebuffer[p]
21     foreach e ∈ edgesS do
22         G.vertex[e.src].addOutEdge(e.dest, &e.val)
23     end
24 end
25 return G
```

preceding updates, adhering to the *asynchronous* model of computation. This solution, of course, limits the amount of effective parallelism. For some algorithms, consistency is not critical (for example, see [29]), and we allow the user to enable fully parallel updates.

## 3.3 Updating Graph to Disk

Finally, the updated edge values need to be written to disk and be visible to the next execution interval. PSW can do this efficiently: The edges are loaded from disk in large blocks, which are cached in memory. When the subgraph for an interval is created, the edges are referenced as pointers to the cached blocks; modifications to the edge values directly modify the data blocks themselves. After finishing the updates for the execution interval, PSW writes the modified blocks back to disk, replacing the old data. The memory-shard is completely rewritten, while only the active **sliding window** of each sliding shard is rewritten to disk (see Algorithm 2). When PSW moves to the next interval, it reads the new values from disk, thus implementing the asynchronous model. The number of non-sequential disk writes for a execution interval is $P$, exactly same as the number of reads. Note, if an algorithm only updates edges in one direction, PSW only writes the modified blocks to disk.




(a) Execution interval (vertices 1-2) (b) Execution interval (vertices 1-2) (c) Execution interval (vertices 3-4) (d) Execution interval (vertices 3-4)

Figure 3: Illustration of the operation of the PSW method on a toy graph (See the text for description).

## 3.4 Example

We now describe a simple example, consisting of two execution intervals, based on Figure 3. In this example, we have a graph of six vertices, which have been divided into three equal intervals: 1–2, 3–4, and 5–6. Figure 3a shows the initial contents of the three shards. PSW begins by executing interval 1, and loads the subgraph containing of edges drawn in bold in Figure 3c. The first shard is the memory-shard, and it is loaded fully. Memory-shard contains all in-edges for vertices 1 and 2, and a subset of the out-edges. Shards 2 and 3 are the sliding shards, and the windows start from the beginning of the shards. Shard 2 contains two out-edges of vertices 1 and 2; shard 3 has only one. Loaded blocks are shaded in Figure 3a. After loading the graph into memory, PSW runs the update-function for vertices 1 and 2. After executing the updates, the modified blocks are written to disk; updated values can be seen in Figure 3b.

PSW then moves to the second interval, with vertices 3 and 4. Figure 3d shows the corresponding edges in bold, and Figure 3b shows the loaded blocks in shaded color. Now shard 2 is the memory-shard. For shard 3, we can see that the blocks for the second interval appear immediately after the blocks loaded in the first. Thus, PSW just "slides" a window forward in the shard.

## 3.5 Evolving Graphs

We now modify the PSW model to support changes in the graph *structure*. Particularly, we allow adding edges to the graph efficiently, by implementing a simplified version of I/O efficient buffer trees [2].

Because a shard stores edges sorted by the source, we can divide the shard into $P$ logical parts: part $j$ contains edges with source in the interval $j$. We associate an in-memory **edge-buffer(p, j)** for each logical part $j$, of shard $p$. When an edge is added to the graph, it is first added to the corresponding edge-buffer (Figure 4). When an interval of vertices is loaded from disk, the edges in the edge-buffers are added to the in-memory graph (Alg. 2).

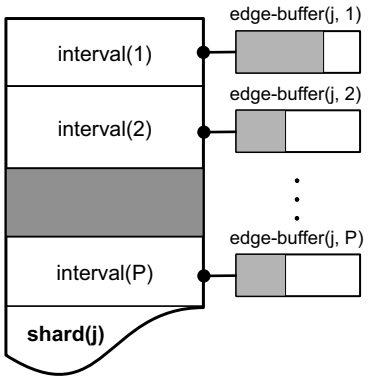



Figure 4: A shard can be split into $P$ logical parts corresponding to the vertex intervals. Each part is associated with an in-memory edge-buffer, which stores the inserted edges that have not yet been merged into the shard.

After each iteration, if the number of edges stored in edge-buffers exceeds a predefined limit, PSW will write the buffered edges to disk. Each shard, that has more buffered edges than a shard-specific limit, is recreated on disk by merging the buffered edges with the edges stored on the disk. The merge requires one sequential read and write. However, if the merged shard becomes too large to fit in memory, it is *split* into two shards with approximately equal number of edges. Splitting a shard requires two sequential writes.

PSW can also support removal of edges: removed edges are flagged and ignored, and permanently deleted when the corresponding shard is rewritten to disk.

Finally, we need to consider consistency. It would be complicated for programmers to write update-functions that support vertices that can change during the computation. Therefore, if an addition or deletion of an edge would affect a vertex in current execution interval, it is added to the graph only after the execution interval has finished.

### 3.6 Analysis of the I/O Costs

We analyze the I/O efficiency of PSW in the I/O model by Aggarwal and Vitter [1]. In this model, cost of an algorithm is the number of block transfers from disk to main memory. The complexity is parametrized by the size of block transfer, $B$, stated in the unit of the *edge object* (which includes the associated value). An upper bound on the number of block transfers can be analyzed by considering the total size of data accessed divided by $B$, and then adding to this the number of *non-sequential* seeks. The total data size is $|E|$ edge objects, as every edge is stored once. To simplify the analysis, we assume that $|E|$ is a multiple of $B$, and shards have equal sizes $\frac{|E|}{P}$. We will now see that $Q_B(E)$, the I/O cost of PSW, is almost linear in $|E|/B$, which is optimal because all edges need to be accessed:

Each edge is accessed twice (once in each direction) during one full pass over the graph. If both endpoints of an edge belong to the same vertex interval, the edge is read only once from disk; otherwise, it is read twice. If the update-function modifies edges in both directions, the number of writes is exactly the same; if in only one direction, the number of writes is half as many. In addition, in the worst (common) case, PSW requires $P$ non-sequential disk seeks to load the edges from the $P-1$ sliding shards for an execution interval. Thus, the total number of non-sequential seeks for a full iteration has a cost of $\Theta(P^2)$ (the number is not exact, because the size of the sliding windows are generally not multiples of $B$).

Assuming that there is sufficient memory to store one memory-shard and out-edges for an execution interval a time, we can now bound the I/O complexity of PSW:

$$\frac{2|E|}{B} \leq Q_B(E) \leq \frac{4|E|}{B} + \Theta(P^2)$$

As the number of non-sequential disk seeks is only $\Theta(P^2)$, PSW performs well also on rotational hard drives.

### 3.7 Remarks

The PSW method imposes some limitations on the computation. Particularly, PSW cannot efficiently support dynamic ordering, such as priority ordering, of computation [29, 34]. Similarly, graph traversals or vertex queries are not efficient in the model, because loading the neighborhood of a single vertex requires scanning a complete memory-shard.

Often the user has a plenty of memory, but not quite enough to store the whole graph in RAM. Basic PSW would not utilize all the available memory efficiently, because the amount of bytes transferred from disk is independent of the available RAM. To improve performance, system can *pin* a set of shards to memory, while the rest are processed from disk.

## 4 System Design & Implementation

This section describes selected details of our implementation of the Parallel Sliding Windows method, GraphChi. The C++ implementation has circa 8,000 lines of code.

### 4.1 Shard Data Format

Designing an efficient format for storing the shards is paramount for good performance. We designed a compact format, which is fast to generate and read, and exploits the sparsity of the graph. In addition, we separate the graph structure from the associated edge values on disk. This is important, because only the edge data is mutated during computation, and the graph structure can be often efficiently compressed. Our data format is as follows:

- The **adjacency shard** stores, implicitly, an edge array for each vertex, in order. Edge array of a vertex starts with a variable-sized length word, followed by the list of neighbors. If a vertex has no edges in this shard, zero length byte is followed by the number of subsequent vertices with no edges in this shard.
- The **edge data shard** is a flat array of edge values, in user-defined type. Values must be of constant size[3].

The current compact format for storing adjacency files is quite simple, and we plan to evaluate more efficient formats in the future. It is possible to further compress the adjacency shards using generic compression software. We did not implement this, because of added complexity and only modest expected improvement in performance.

#### 4.1.1 Preprocessing

GraphChi includes a program, Sharder, for creating shards from standard graph file formats. Preprocessing is I/O efficient, and can be done with limited memory (Table 1).

1. Sharder counts the in-degree (number of in-edges) for each of the vertices, requiring one pass over the input file. The degrees for consecutive vertices can be combined to save memory. To finish, Sharder computes the *prefix sum* [10] over the degree array, and divides vertices into $P$ **intervals** with approximately the same number of in-edges.
2. On the second pass, Sharder writes each edge to a temporary scratch file of the owning shard.
3. Sharder processes each scratch file in turn: edges are sorted and the shard is written in compact format.

[3]The model can support variable length values by splitting the shards into smaller blocks which can efficiently be shrunk or expanded. For simplicity, we assume constant size edge values in this paper.

4. Finally, Sharder computes a binary **degree file** containing in- and out-degree for each vertex, which is needed for the efficient operation of GraphChi, as described below.

The number of shards $P$ is chosen so that the biggest shard is at most one fourth of the available memory, leaving enough memory for storing the necessary pointers of the subgraph, file buffers, and other auxiliary data structures. The total I/O cost of the preprocessing is $\frac{5|E|}{B} + \frac{|V|}{B}$.

## 4.2 Main Execution

We now describe how GraphChi implements the PSW method for loading, updating, and writing the graph. Figure 5 shows the processing phases as a flow chart.

### 4.2.1 Efficient Subgraph Construction

The first prototypes of GraphChi used STL vectors to store the list of edges for each vertex. Performance profiling showed that a significant amount of time was used in resizing and reallocating the edge arrays. Therefore, to eliminate dynamic allocation, GraphChi calculates the memory needs exactly prior to an execution interval. This optimization is implemented by using the **degreefile**, which was created at the end of preprocessing and stores the in- and out-degrees for each vertex as a flat array. Prior to initializing a subgraph, GraphChi computes a *prefix-sum* of the degrees, giving the exact indices for edge arrays for every vertex, and the exact array size that needs to be allocated. Compared to using dynamic arrays, our solution improved running time by approximately 30%.

*Vertex values:* In our computational model, each vertex has an associated value. We again exploit the fact that the system considers vertices in sequential order. GraphChi stores vertex values in a single file as flat array of user-defined type. The system writes and reads the vertex values once per iteration, with I/O cost of $2\lceil |V|/B \rceil$.

*Multithreading:* GraphChi has been designed to overlap disk operations and in-memory computation as much as possible. Loading the graph from disk is done by concurrent threads, and writes are performed in the background.

### 4.2.2 Sub-intervals

The $P$ intervals are chosen as to create shards of roughly same size. However, it is not guaranteed that the number of edges in each subgraph is balanced. Real-world graphs typically have very skewed in-degree distribution; a vertex interval may have a large number of vertices, with very low average in-degree, but high out-degree, and thus a full subgraph of a interval may be too large to load in memory.

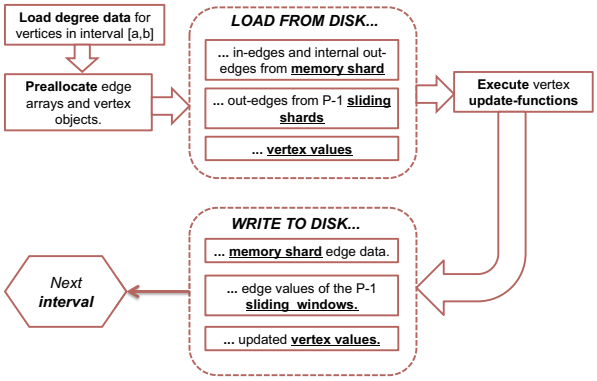



Figure 5: **Main execution flow.** Sequence of operations for processing one execution interval with GraphChi.

We solve this problem by dividing execution intervals into **sub-intervals**. As the system already loads the degree of every vertex, we can use this information to compute the exact memory requirement for a range of vertices, and divide the original intervals to sub-intervals of appropriate size. Sub-intervals are preferred to simply re-defining the intervals, because it allows same shard files to be used with different amounts of memory. Because sub-intervals share the same memory-shard, I/O costs are not affected.

### 4.2.3 Evolving Graphs

We outlined the implementation in previous section. The same execution engine is used for dynamic and static graphs, but we need to be careful in maintaining auxiliary data structures. First, GraphChi needs to keep track of the changing vertex degrees and modify the degreefile accordingly. Second, the degreefile and vertex data file need to grow when the number of vertices increases, and the vertex intervals must be maintained to match the splitting and expansion of shards. Adding support for evolving graphs was surprisingly simple, and required less than 1000 lines of code (15% of the total).

## 4.3 Selective Scheduling

Often computation converges faster on same parts of a graph than in others, and it is desirable to focus computation only where it is needed. GraphChi supports **selective scheduling**: an update can flag a neighboring vertex to be updated, typically if edge value changes significantly. In the evolving graph setting, selective scheduling can be used to implement *incremental computation*: when an edge is created, its source or destination vertex is added to the schedule [15].

GraphChi implements selective scheduling by representing the current schedule as a bit-array (we assume enough memory to store $|V|/8$ bytes for the schedule). A simple

optimization to the PSW method can now be used: On the first iteration, it creates a sparse index for each shard, which contains the file indices of each sub-interval. Using the index, GraphChi can skip unscheduled vertices.

# 5 Programming Model

Programs written for GraphChi are similar to those written for Pregel [31] or GraphLab [29], with the following main differences. Pregel is based on the messaging model, while GraphChi programs directly modify the values in the edges of the graph; GraphLab allows programs to directly read and modify the values of neighbor vertices, which is not allowed by GraphChi, unless there is enough RAM to store all vertex values in memory. We now discuss the programming model in detail, with a running example.

**Running Example:** As a running example, we use a simple GraphChi implementation of the PageRank [32] algorithm. The vertex update-function is simple: at each update, compute a weighted sum of the ranks of in-neighbors (vertices with an edge directed to the vertex). Incomplete pseudo-code is shown in Algorithm 4 (definitions of the two internal functions are model-specific, and discussed below).The program computes by executing the update function for each vertex in turn for a predefined number of iterations.[4]

**Algorithm 4**: Pseudo-code of the vertex update-function for weighted PageRank.

```
1 typedef: VertexType float
2 Update(vertex) begin
3 |   var sum ← 0
4 |   for e in vertex.inEdges() do
5 |   |   sum += e.weight * neighborRank(e)
6 |   end
7 |   vertex.setValue(0.15 + 0.85 * sum)
8 |   broadcast(vertex)
9 end
```

**Standard Programming Model:** In the standard setting for GraphChi, we assume that there is not enough RAM to store the values of vertices. In the case of PageRank, the vertex values are floating point numbers corresponding to the rank (Line 1 of Algorithm 4).

The update-function needs to read the values of its neighbors, so the only solution is to *broadcast* vertex values via the edges. That is, after an update, the new rank of a vertex is written to the out-edges of the vertex. When neighboring vertex is updated, it can access the vertex rank by reading the adjacent edge's value, see Algorithm 5.

[4]Note that this implementation is not optimal, we discuss a more efficient version in the next section

**Algorithm 5**: Type definitions, and implementations of neighborRank() and broadcast() in the standard model.

```
1 typedef: EdgeType { float weight, neighbor_rank; }
2 neighborRank(edge) begin
3 |   return edge.weight * edge.neighbor_rank
4 end
5 broadcast(vertex) begin
6 |   for e in vertex.outEdges() do
7 |   |   e.neighbor_rank = vertex.getValue()
8 |   end
9 end
```

If the size of the vertex value type is small, this model is competitive even if plenty of RAM is available. Therefore, for better portability, it is encouraged to use this form. However, for some applications, such as matrix factorization (see Section 6), the vertex value can be fairly large (tens of bytes), and replicating it to all edges is not efficient. To remedy this situation, GraphChi supports an alternative programming model, discussed next.

**Alternative Model: In-memory Vertices:** It is common that the number of vertices in a problem is relatively small compared to the number of edges, and there is sufficient memory to store the array of vertex values. In this case, an update-function can read neighbor values directly, and there is no need to broadcast vertex values to incident edges (see Algorithm 6).

**Algorithm 6**: Datatypes and implementations of neighborRank() and broadcast() in the alternative model.

```
1 typedef: EdgeType { float weight; }
2 float[] in_mem_vert
3 neighborRank(edge) begin
4 |   return edge.weight * in_mem_vert[edge.vertex_id]
5 end
6 broadcast(vertex) /* No-op */
```

We have found this model particularly useful in several *collaborative filtering* applications, where the number of vertices is typically several orders of magnitude smaller than the number of edges, and each vertex must store a vector of floating point values. The ability to access directly vertex values requires us to consider consistency issues. Fortunately, as GraphChi sequentializes updates of vertices that share an edge, read-write races are avoided assuming that the update-function does not modify other vertices.

# 6 Applications

We implemented and evaluated a wide range of applications, in order to demonstrate that GraphChi can be used for problems in many domains. Despite the restrictive external memory setting, GraphChi retains the expressivity of other graph-based frameworks. The source code for most of the example applications is included in the open-source version of GraphChi.

**SpMV kernels, Pagerank:** Iterative sparse-matrix dense-vector multiply (SpMV) programs are easy to represent in the vertex-centric model. Generalized SpMV algorithms iteratively compute $x^{t+1} = Ax^t = \bigoplus_{i=1}^{n} A_i \otimes x^t$, where $x^t$ represents a vector of size $n$ and $A$ is a $m \times n$ matrix with row-vectors $A_i$. Operators $\oplus$ and $\otimes$ are algorithm-specific: standard addition and multiplication operators yields standard matrix-vector multiply. Represented as a graph, each edge $(u, v)$ represents non-empty matrix cell $A(u, v)$ and vertex $v$ the vector cell $x(v)$.

We wrote a special programming interface for SpMV applications, enabling important optimizations: Instead of writing an update-function, the programmer implements the $\oplus$ and $\otimes$ operators. When executing the program, GraphChi can bypass the construction of the subgraph, and directly apply the operators when edges are loaded, with improved performance of approx. $25\%$. We implemented Pagerank [32] as iterated matrix-vector multiply.

**Graph Mining:** We implemented three algorithms for analyzing graph structure: Connected Components, Community Detection, and Triangle Counting. The first two algorithms are based on *label propagation* [47]. On first iteration, each vertex writes its id ("label") to its edges. On subsequent iterations, vertex chooses a new label based on the labels of its neighbors. For Connected Components, vertex chooses the minimum label; for Community Detection, the most frequent label is chosen [28]. A neighbor is scheduled only if a label in a connecting edge changes, which we implement by using selective scheduling. Finally, sets of vertices with equal labels are interpreted as connected components or communities, respectively.

The goal of Triangle Counting is to count the number of edge triangles incident to each vertex. This problem is used in social network analysis for analyzing the graph connectivity properties [43]. Triangle Counting requires computing intersections of the adjacency lists of neighboring vertices. To do this efficiently, we first created a graph with vertices sorted by their degree (using a modified preprocessing step). We then run GraphChi for $P$ iterations: on each iteration, adjacency list of a selected interval of vertices is stored in memory, and the adjacency lists of vertices with smaller degrees are compared to the selected vertices by the update function.

**Collaborative Filtering:** Collaborative filtering is used, for example, to recommend products based on purchases of other users with similar interests. Many powerful methods for collaborative filtering are based on low-rank matrix factorization. The basic idea is to approximate a large sparse matrix $R$ by the product of two smaller matrices: $R \approx U \times V'$.

We implemented the Alternating Least Squares (ALS) algorithm [46], by adapting a GraphLab implementation [30]. We used ALS to solve the Netflix movie rating prediction problem [6]: in this model, the graph is bipartite, with each user and movie represented by a vertex, connected by an edge storing the rating (edges correspond to the non-zeros of matrix $R$). The algorithm computes a D-dimensional *latent vector* for each movie and user, corresponding to the rows of $U$ and $V$. A vertex update solves a regularized least-squares system, with neighbors' latent factors as input. If there is enough RAM, we can store the latent factors in memory; otherwise, each vertex replicates its factor to its edges. The latter requires more disk space, and is slower, but is not limited by the amount of RAM, and can be used for solving very large problems.

**Probabilistic Graphical Models:** Probabilistic Graphical Models are used in Machine Learning for many structured problems. The problem is encoded as a graph, with a vertex for each random variable. Edges connect related variables and store a *factor* encoding the dependencies. Exact inference on such models is intractable, so approximate methods are required in practice. Belief Propagation (BP) [35], is a powerful method based on iterative message passing between vertices. The goal here is to estimate the probabilities of variables ("beliefs").

For this work, we adapted a special BP algorithm proposed by Kang et. al. [22], which we call WebGraph-BP. The purpose of this application is to execute BP on a graph of webpages to determine whether a page is "good" or "bad". For example, phishing sites are regarded as bad and educational sites as good. The problem is bootstrapped by declaring a seed set of good and bad websites. The model defines binary probability distribution of adjacent webpages and after convergence, each webpage – represented by a vertex – has an associated belief of its quality. Representing Webgraph-BP in GraphChi is straightforward, the details of the algorithm can be found elsewhere [22].

# 7 Experimental Evaluation

We evaluated GraphChi using the applications described in previous section and analyzed its performance on a selection of large graphs (Table 1).

## 7.1 Test setup

Most of the experiments were performed on a Apple Mac Mini computer ("Mac Mini"), with dual-core 2.5 GHz Intel i5 processor, 8 GB of main memory and a standard 256GB SSD drive (price $1,683 (Jan, 2012)). In addition, the computer had a 750 GB, 7200 rpm hard drive. We ran standard Mac OS X Lion, with factory settings. Filesystem caching was disabled to make executions with small and large input graphs comparable . For experiments with multiple hard drives we used an older 8-core server with four AMD Opteron 8384 processors, 64GB of RAM, running Linux ("AMD Server").

| Graph name | Vertices | Edges | P | Preproc. |
|---|---|---|---|---|
| live-journal [3] | 4.8M | 69M | 3 | 0.5 min |
| netflix [6] | 0.5M | 99M | 20 | 1 min |
| domain [44] | 26M | 0.37B | 20 | 2 min |
| twitter-2010 [26] | 42M | 1.5B | 20 | 10 min |
| uk-2007-05 [11] | 106M | 3.7B | 40 | 31 min |
| uk-union [11] | 133M | 5.4B | 50 | 33 min |
| yahoo-web [44] | 1.4B | 6.6B | 50 | 37 min |

Table 1: Experiment graphs. Preprocessing (conversion to shards) was done on Mac Mini.

## 7.2 Comparison to Other Systems

We are not aware of any other system that would be able to compute on such large graphs as GraphChi on a single computer (with reasonable performance). To get flavor of the performance of GraphChi, we compare it to several existing *distributed* systems and the shared-memory GraphLab [29], based mostly on results we found from recent literature[5]. Our comparisons are listed in Table 2.

Although disk-based, GraphChi runs three iterations of Pagerank on the *domain* graph in 132 seconds, only roughly 50% slower than the shared-memory GraphLab (on AMD Server)[6]. Similar relative performance was obtained for ALS matrix factorization, if vertex values are stored in-memory. Replicating the latent factors to edges increases the running time by five-fold.

A recently published paper [38] reports that Spark [45], running on a cluster of 50 machines (100 CPUs) [45] runs five iterations of Pagerank on the *twitter-2010* in 486.6 seconds. GraphChi solves the same problem in less than double of the time (790 seconds), with only 2 CPUs. Note that Spark is implemented in Scala, while GraphChi is native C++ (an early Scala/Java-version of GraphChi runs 2-3x slower than the C++ version). Stanford GPS [37] is a new implementation of Pregel, with compelling performance. On a cluster of 30 machines, GPS can run 100 iterations of Pagerank (using random partitioning) in 144 minutes, approximately four times faster than GraphChi on the Mac Mini. Piccolo [36] is reported to execute one iteration of synchronous Pagerank on a graph with 18.5B edges in 70 secs, running on a 100-machine EC2 cluster. The graph is not available, so we extrapolated our results for the *uk-union* graph (which has same ratio of edges to vertices), and estimated that GraphChi would solve the same problem in 26 minutes. Note, that both Spark and Piccolo execute Pagerank synchronously, while GraphChi uses asynchronous computation, with relatively faster convergence [7].

GraphChi is able to solve the WebGraph-BP on *yahoo-web* in 25 mins, almost as fast as Pegasus [24], a Hadoop-based [7] graph mining library, distributed over 100 nodes (Yahoo M-45). GraphChi counts the triangles of the *twitter-2010* graph in less then 90 minutes, while a Hadoop-based algorithm uses over 1,600 workers to solve the same problem in over 400 minutes [39]. These results highlight the inefficiency of MapReduce for graph problems. Recently, Chu et al. proposed an I/O efficient algorithm for triangle counting [18]. Their method can list the triangles of a graph with 106 mil. vertices and 1.9B edges in 40 minutes. Unfortunately, we were unable to repeat their experiment due to unavailability of the graph.

Finally, we include comparisons to PowerGraph [20], which was published simultaneously with this work (PowerGraph and GraphChi are projects of the same research team). PowerGraph is a distributed version of GraphLab [29], which employs a novel vertex-partitioning model and a new Gather-Apply-Scatter (GAS) programming model allowing it to compute on graphs with power-law degree distribution extremely efficiently. On a cluster of 64 machines in the Amazon EC2 cloud, PowerGraph can execute one iteration of PageRank on the *twitter-2010* graph in less than 5 seconds (GraphChi: 158 s), and solves the triangle counting problem in 1.5 minutes (GraphChi: 60 mins). Clearly, ignoring graph loading, PowerGraph can execute graph computations on a large cluster many times faster than GraphChi on a single machine. It is interesting to consider also the relative performance: with 256 times the cores (or 64 times the machines), PowerGraph can solve

[5] The results we found do not consider the time it takes to load the graph from disk, or to transfer it over a network to a cluster.

[6] For GraphLab we used their reference implementation of Pagerank. Code was downloaded April 16, 2012.

[7] `http://hadoop.apache.org/`

| Application & Graph | Iter. | Comparative result | GraphChi (Mac Mini) | Ref |
|---|---|---|---|---|
| Pagerank & domain | 3 | GraphLab[30] on AMD server (8 CPUs) **87 s** | **132 s** | - |
| Pagerank & twitter-2010 | 5 | Spark [45] with 50 nodes (100 CPUs): **486.6 s** | **790 s** | [38] |
| Pagerank & V=105M, E=3.7B | 100 | Stanford GPS, 30 EC2 nodes (60 virt. cores), **144 min** | approx. **581 min** | [37] |
| Pagerank & V=1.0B, E=18.5B | 1 | Piccolo, 100 EC2 instances (200 cores) **70 s** | approx. **26 min** | [36] |
| Webgraph-BP & yahoo-web | 1 | Pegasus (Hadoop) on 100 machines: **22 min** | **27 min** | [22] |
| ALS & netflix-mm, D=20 | 10 | GraphLab on AMD server: **4.7 min** | **9.8 min** (in-mem)<br>**40 min** (edge-repl.) | [30] |
| Triangle-count & twitter-2010 | - | Hadoop, 1636 nodes: **423 min** | **60 min** | [39] |
| Pagerank & twitter-2010 | 1 | PowerGraph, 64 x 8 cores: **3.6 s** | **158 s** | [20] |
| Triange-count & twitter- 2010 | - | PowerGraph, 64 x 8 cores: **1.5 min** | **60 min** | [20] |

Table 2: **Comparative performance.** Table shows a selection of recent running time reports from the literature.

the problems 30 to 45 times faster than GraphChi.

While acknowledging the caveats of system comparisons, this evaluation demonstrates that GraphChi provides sufficient performance for many practical purposes. Remarkably, GraphChi can solve as large problems as reported for any of the distributed systems we reviewed, but with fraction of the resources.

## 7.3 Scalability and Performance

Here, we demonstrate that GraphChi can handle large graphs with robust performance. Figure 7 shows the normalized performance of the system on three applications, with all of our test graphs (Table 1). The x-axis shows the number of edges of the graph. Performance is measured as *throughput*, the number of edges processed in second. Throughput is impacted by the internal structure of a graph (see Section 3.6), which explains why GraphChi performs slower on the largest graph, *yahoo-web*, than on the next largest graphs, *uk-union* and *uk-2007-5*, which have been optimized for locality. Consistent with the I/O bounds derived in Section 3.6, the ratio between the fastest and slowest result is less than two. For the three algorithms, GraphChi can process 5-20 million edges/sec on the Mac Mini.

The performance curve for SSD and hard drive have similar shape, but GraphChi performs twice as fast on an SSD. This suggests that the performance even on a hard drive is adequate for many purposes, and can be improved by using multiple hard drives, as shown in Figure 8a. In this test, we modified the I/O-layer of GraphChi to *stripe* files across disks. We installed three 2TB disks into the AMD server and used stripe-size of 10 MB. Our solution is similar to the RAID level 0 [33]. At best, we could get a total of 2x speedup with three drives.

Figure 8b shows the effect of block size on performance of GraphChi on SSDs and HDs. With very small blocks, the observed that OS overhead becomes large, affecting also

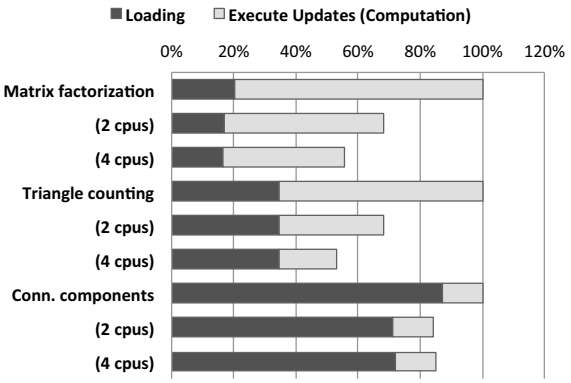
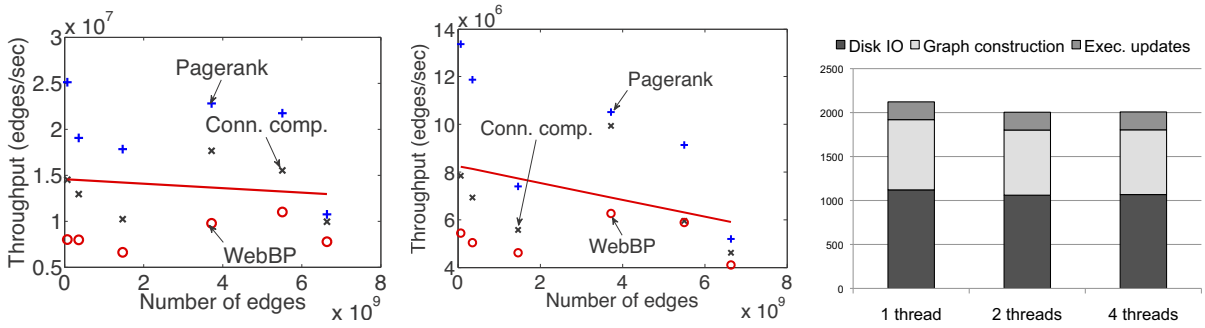



Figure 6: Relative runtime when varying the number of threads used used by GraphChi. Experiment was done on a MacBook Pro (mid-2012) with four cores.

the SSD. GraphChi on the SSD achieves peak performance with blocks of about 1 MB. With hard drives, even bigger block sizes can improve performance; however, the block size is limited by the available memory. Figure 8c shows how the choice of $P$ affects performance. As the number of non-sequential seeks is quadratic in $P$, if the $P$ is in the order of dozens, there is little real effect on performance.

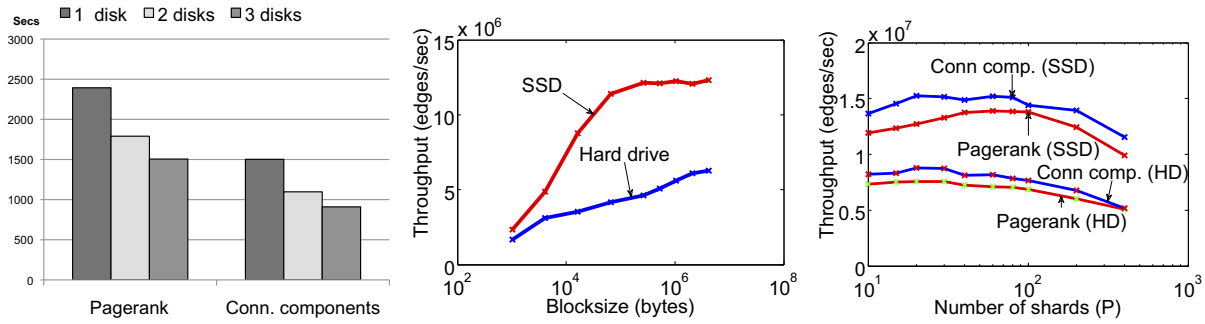
| Application | SSD | In-mem | Ratio |
|---|---|---|---|
| Connected components | 45 s | 18 s | 2.5x |
| Community detection | 110 s | 46 s | 2.4x |
| Matrix fact. (D=5, 5 iter) | 114 s | 65 s | 1.8x |
| Matrix fact. (D=20, 5 iter.) | 560 s | 500 s | 1.1x |

Table 3: Relative performance of an in-memory version of GraphChi compared to the default SSD-based implementation on a selected set of applications, on a Mac Mini. Timings include the time to load the input from disk and write the output into a file.




(a) Performance: SSD (b) Performance : Hard drive (c) Runtime breakdown

Figure 7: (a,b) Computational throughput of GraphChi on the experiment graphs (x-axis is the number of edges) on SSD and hard drive (higher is better), without selective scheduling, on three different algorithms. The trend-line is a least-squares fit to the average throughput of the applications. GraphChi performance remains good as the input graphs grow, demonstrating the scalability of the design. Notice different scales on the y-axis. . (c) Breakdown of the processing phases for the Connected Components algorithm (3 iterations, *uk-union* graph; Mac Mini, SSD).



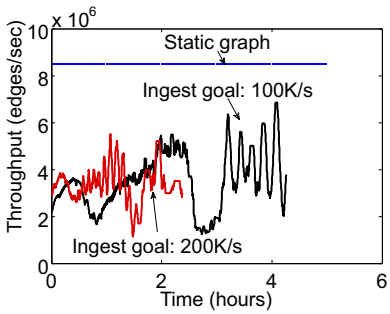

(a) Multiple hard drives (b) Disk block size (c) Number of shards

Figure 8: (a) Runtime of 3 iterations on the *uk-union* graph, when data is striped across 2 or 3 hard drives (AMD server). (b) Impact of the block size used for disk I/O (x-axis is in log-scale). (c) The number of shards has little impact on performance, unless $P$ is very large.

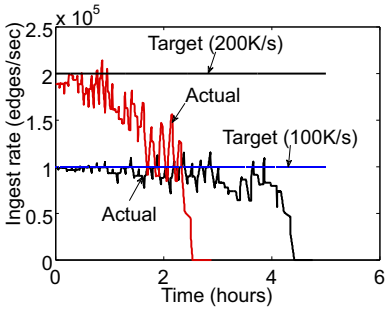
Next, we studied the bottlenecks of GraphChi. Figure 7c shows the break-down of time used for I/O, graph construction and actual updates with Mac Mini (SSD) when running the Connected Components algorithm.We disabled asynchronous I/O for the test, and actual combined running time is slightly less than shown in the plot. The test was repeated by using 1, 2 and 4 threads for shard processing and I/O. Unfortunately, the performance is only slightly improved by parallel operation. We profiled the execution, and found out that GraphChi is able to nearly saturate the SSD with only one CPU, and achieves combined read/write bandwidth of 350 MB/s. GraphChi's performance is limited by the I/O bandwidth. More benefit from parallelism can be gained if the computation itself is demanding, as shown in Figure 6. This experiment was made with a mid-2012 model MacBook Pro with a four-core Intel i7 CPU.

We further analyzed the relative performance of the disk-based GraphChi to a modified in-memory version of GraphChi. Table 3 shows that on tasks that are computationally intensive, such as matrix factorization, the disk overhead (SSD) is small, while on light tasks such as computing connected components, the total running time can be over two times longer. In this experiment, we compared the total time to execute a task, from loading the graph from disk to writing the results into a file. For the top two experiments, the *live-journal* graph was used, and the last two experiments used the *netflix* graph. The larger graphs did not fit into RAM.

**Evolving Graphs:** We evaluated the performance of GraphChi on a constantly growing graph. We inserted edges from the *twitter-2010* graph, with rates of 100K and 200K edges in second, while simultaneously running Pagerank. Edges were loaded from the hard drive, GraphChi operated on the SSD. Figure 9a shows the throughput over




(a) Evolving Graph: Throughput




(b) Ingest rate

Figure 9: (a,b) Evolving graphs: Performance when *twitter-2010* graph is ingested with a cap of 100K or 200K edges/sec, while simultaneously computing Pagerank.

time. The throughput varies as the result of periodic flushing of edge-buffers to disk, and the bumps in throughput, just after half-way of execution, are explained by a a series of shard *splits*. Throughput in the evolving graph case is roughly 50% compared to normal execution on the full graph. GraphChi currently favors computation over ingest rate, which explains the decreasing actual ingest rate over time shown in Figure 9b. A rate of 100K edges/sec can be sustained for a several hours, but with 200K edges/sec, edge buffers fill up quickly, and GraphChi needs to flush the updates to disk too frequently, and cannot sustain the ingestion rate. These experiments demonstrate that GraphChi is able to handle a very quickly growing graph on just one computer.

# 8 Related Work

Pearce et al. [34] proposed an asynchronous system for graph traversals on external and semi-external memory. Their solution stores the graph structure on disk using the *compressed sparse row* format, and unlike GraphChi, does not allow changes to the graph. Vertex values are stored in memory, and computation is scheduled using concurrent work queues. Their system is designed for graph traversals, while GraphChi is designed for general large-scale graph computation and has lower memory requirements.

A collection of I/O efficient fundamental graph algorithms in the external memory setting was proposed by Chiang et. al. [16]. Their method is based on simulating parallel PRAM algorithms, and requires a series of disk sorts, and would not be efficient for the types of algorithms we consider. For example, the solution to connected components has upper bound I/O cost of $O(sort(|V|))$, while ours has $O(|E|)$. Many real-world graphs are sparse, and it is unclear which bound is better in practice. A similar approach was recently used by Blelloch et. al. for I/O efficient Set Covering algorithms [9].

Optimal bounds for I/O efficient SpMV algorithms was derived recently by Bender [5]. Similar methods were earlier used by Haveliwala [21] and Chen et. al. [14]. GraphChi and the PSW method extend this work by allowing asynchronous computation and mutation of the underlying matrix (graph), thus representing a larger set of applications. Toledo [40] contains a comprehensive survey of (mostly historical) algorithms for out-of-core numerical linear algebra, and discusses also methods for sparse matrices. For most external memory algorithms in literature, implementations are not available.

Finally, *graph databases* allow for storing and querying graphs on disk. They do not, however, provide powerful computational capabilities.

# 9 Conclusions

General frameworks such as MapReduce deliver disappointing performance when applied to real-world graphs, leading to the development of specialized frameworks for computing on graphs. In this work, we proposed a new method, Parallel Sliding Windows (PSW), for the external memory setting, which exploits properties of sparse graphs for efficient processing from disk. We showed by theoretical analysis, that PSW requires only a small number of sequential disk block transfers, allowing it to perform well on both SSDs and traditional hard disks.

We then presented and evaluated our reference implementation, GraphChi, and demonstrated that on a consumer PC, it can efficiently solve problems that were previously only accessible to large-scale cluster computing. In addition, we showed that GraphChi relatively (per-node basis) outperforms other existing systems, making it an attractive choice for parallelizing multiple computations on a cluster.

**Acknowledgments**

We thank Joey Gonzalez, Yucheng Low, Jay Gu, Joseph Bradley, Danny Bickson, Phillip B. Gibbons, Eriko Nurvitadhi, Julian Shun, the anonymous reviewers and our shepherd Prof. Arpaci-Dusseau for feedback and helpful discussions. Funded by ONR PECASE N000141010672, Intel Science & Technology Center on Embedded Computing, ARO MURI W911NF0810242.

## References


[1] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. *Algorithms and Data Structures*, pages 334–345, 1995.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. The 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'06. ACM, 2006.

[4] A. Badam and V. S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In *Proc. of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 16–16, Boston, MA, 2011. USENIX Association.

[5] M. Bender, G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory of Computing Systems*, 47(4):934–962, 2010.

[6] J. Bennett and S. Lanning. The netflix prize. In *Proc. of the KDD Cup Workshop 2007*, pages 3–6, San Jose, CA, Aug. 2007. ACM.

[7] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[8] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *In Proc. of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2003.

[9] G. Blelloch, H. Simhadri, and K. Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proc. of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 82–90, 2012.

[10] G. E. Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, 1990.

[11] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[12] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proc. of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[13] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1123–1126, Indianapolis, Indiana, USA, 2010. ACM.

[14] Y. Chen, Q. Gan, and T. Suel. I/O-efficient techniques for computing pagerank. In *Proc. of the eleventh international conference on Information and knowledge management*, pages 549–557, McLean, Virginia, USA, 2002. ACM.

[15] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proc. of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 85–98, Bern, Switzerland, 2012. ACM.

[16] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 139–149, Philadelphia, PA, 1995. Society for Industrial and Applied Mathematics.

[17] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, Paris, France, April 2009. ACM.

[18] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *In Proc. of the 17th ACM SIGKDD international conf. on Knowledge discovery and data mining*, pages 672–680, 2011.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of the 6th USENIX conference on Operating systems design and implementation*, OSDI'04, pages 10–10, San Francisco, CA, 2004. USENIX.

[20] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX conference on Operating systems design and implementation*, OSDI'12, Hollywood, CA, 2012.

[21] T. Haveliwala. Efficient computation of pagerank. Technical report, Stanford University, 1999.

[22] U. Kang, D. Chau, and C. Faloutsos. Inference of beliefs on billion-scale graphs. In *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*, Washington, D.C., 2010.

[23] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *11th International Conference on Data Mining (ICDM'11)*, pages 300–309, Vancouver, Canada, 2011.

[24] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. ICDM '09. IEEE Computer Society, 2009.

[25] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[26] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[27] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[28] X. Liu and T. Murata. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Stat. Mechanics and its Applications*, 389(7):1493–1500, 2010.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, CA, July 2010.

[30] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. SIGMOD 10: Proc. of the 2010 international conference on Management of data, Indianapolis, IN, 2010.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[33] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, Chicago, IL, 1988.

[34] R. Pearce, M. Gokhale, and N. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SuperComputing*, 2010.

[35] J. Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982.

[36] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proc. of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, 2010.

[37] S. Salihoglu and J. Widom. GPS: a graph processing system. Technical report, Stanford University, 2012.

[38] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. Technical report, Microsoft Research, 2012.

[39] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *In Proc. of the 20th international conference on World wide web*, pages 607–614, Lyon, France, 2011. ACM.

[40] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999.

[41] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[42] J. Vitter. *External Memory Algorithms*. ESA, 1998.

[43] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.

[44] Yahoo WebScope. Yahoo! altavista web page hyperlink connectivity graph, circa 2002, 2012. `http://webscope.sandbox.yahoo.com/`.

[45] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[46] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *In Proc. of the 4th international conference on Algorithmic Aspects in Information and Management*, AAIM '08, pages 337–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Carnegie Mellon University, 2002.

# Hails: Protecting Data Privacy in Untrusted Web Applications

Daniel B. Giffin, Amit Levy, Deian Stefan
David Terei, David Mazières, John C. Mitchell
Stanford

Alejandro Russo
Chalmers

## Abstract

Modern extensible web platforms like Facebook and Yammer depend on third-party software to offer a rich experience to their users. Unfortunately, users running a third-party "app" have little control over what it does with their private data. Today's platforms offer only ad-hoc constraints on app behavior, leaving users an unfortunate trade-off between convenience and privacy. A principled approach to code confinement could allow the integration of untrusted code while enforcing flexible, end-to-end policies on data access. This paper presents a new web framework, Hails, that adds mandatory access control and a declarative policy language to the familiar MVC architecture. We demonstrate the flexibility of Hails through `GitStar.com`, a code-hosting website that enforces robust privacy policies on user data even while allowing untrusted apps to deliver extended features to users.

## 1 Introduction

Extensible web platforms that run third-party *apps* in a restricted manner represent a new way of developing and deploying software. Facebook, for example, has popularized this model for social networking and personal data, while Yammer provides a similar platform geared toward enterprises. The functionality available to users of such sites is no longer the product of a single entity, but the combination of a potentially trustworthy platform running code provided by less-trusted third parties.

Many apps are only useful when they are able to manipulate sensitive user data—personal information such as financial or medical details, or non-public social relationships—but once access to this data has been granted, there is no holistic mechanism to constrain what the app may do with it. For example, the Wall Street Journal reported that some of Facebook's most popular apps, including Zynga's FarmVille game, had been transmitting users' account identifiers (sufficient for obtaining personal information) to dozens of advertisers and online tracking companies [38].

In this conventional model, a user sets privacy settings regarding specific apps, or classes of apps. However, users who wish to benefit from the functionality of an app are forced to guess what risk is posed by granting an app access to sensitive information: the platform cannot provide any mechanistic guarantee that the app will not, for example, mine private messages for ad keywords or credit card numbers and export this information to a system run by the app's developer.

Even if they are aware of how an app behaves, users are generally poorly equipped to understand the consequences of data exfiltration. In fact, a wide range of sophisticated third-party tracking mechanisms are available for collecting and correlating user information, many based only on scant user data [27].

In order to protect the interests of its users, the operator of a conventional web platform is burdened with implementing a complicated security system. These systems are usually ad-hoc, relying on access control lists, human audits of app code, and optimistic trust in various software authors. Moreover, each platform provides a solution different from the other.

To address these problems, we have developed an alternate approach for confining untrusted apps. We demonstrate the system by describing `GitStar.com`, a social code hosting website inspired by GitHub. GitStar takes a new approach to the app model: we host third-party apps in an environment designed to protect data. Rather than ask users whether to disclose their data to certain apps, we support policies that restrict information flow into and out of apps, allowing them to give up communication privileges in exchange for access to user data.

GitStar is built on a new web framework called Hails. While other frameworks are geared towards monolithic web sites, Hails is explicitly designed for building web *platforms*, where it is expected that a site will comprise many mutually-distrustful components written by various entities.

Hails is distinguished by two design principles. First, access policies should be specified declaratively alongside data schemas, rather than strewn throughout the codebase as guards around each point of access. Second, access

policies should be mandatory even once code has obtained access to data.

The first principle leads to an architecture we call model–policy–view–controller (MPVC), an extension to the popular model–view–controller (MVC) pattern. In MVC, models represent a program's persistent data structures. A view is a presentation layer for the end user. Finally, controllers decide how to handle and respond to particular requests. The MVC paradigm does not give access policy a first-class role, making it easy for programmers to overlook checks and allow vulnerabilities [34]. By contrast, MPVC explicitly associates every model with a policy governing how the associated data may be used.

The second principle, that data access policies should be mandatory, means that policies must follow data throughout the system. Hails uses a form of mandatory access control (MAC) to enforce end-to-end policies on data as it passes through software components with different privileges. While MAC has traditionally been used for high-security and military operating systems, it can be applied effectively to the untrusted-app model when combined with a notion of decentralized privileges such as that introduced by the decentralized label model [32].

The MAC regime allows a complex system to be implemented by a reconfigurable assemblage of software components that do not necessarily trust each other. For example, when a user browses a software repository on GitStar, a code-viewing component formats files of source code for convenient viewing. Even if this component is flawed or malicious, the access policy attached to the data and enforced by MAC will prevent it from displaying a file to users without permission to see it, or transmitting a private file to the component's author. Thus, the central GitStar component can make repository contents available to any other component, and users can safely choose third-party viewers based solely on the features they deliver rather than on the trustworthiness of their authors.

A criticism of past MAC systems has been the perceived difficulty for application programmers to understand the security model. Hails offers a new design point in this space by introducing MAC to the popular MVC pattern and binding access control policy to the model component in MPVC. Because GitStar is a public site in production use by more than just its developers, we are able to report on the experiences of third-party app authors. While our sample is yet small, our experience suggests MAC security does not impede application development within an MPVC framework.

The remainder of this paper describes Hails, GitStar, and several add-on components built for GitStar. We discuss design patterns used in building Hails applications. We then evaluate our system, provide a discussion, survey related work, and conclude.

# 2 Design

The Hails MPVC architecture differs from traditional MVC frameworks such as Rails and Django by making security concerns explicit. An MVC framework has no inherent notion of security policy. The effective policy results from an ad-hoc collection of checks strewn throughout the application. By contrast, MPVC gives security policies a first-class role. Developers specify policies in a domain-specific language (DSL) alongside the data model. Relying primarily on language-level security, the framework then enforces these policies system-wide, regardless of the correctness or intentions of untrusted code.

MPVC applications are built from mutually distrustful components. These components fall into two categories: *MPs*, comprising model and policy logic, and *VCs*, comprising view and controller logic. An MP provides an API through which other components can access a particular database, subject to its associated policies.

MPs and VCs are explicitly segregated. An MP cannot interact directly with a user, while a VC cannot access a database without invoking the corresponding MP. Our language-level confinement mechanism enforces MAC, guaranteeing that a data-model's policy is respected throughout the system. For example, if an MP specifies that "only a user's friends may see his email address," then a VC (or other MP) reading a user's email address loses the ability to communicate over the network except to the user's friends (who are allowed to see that email address).

Figure 1 illustrates the interaction between different application components in the context of GitStar. Two MPs are depicted: GitStar, which manages projects and `git` data; and Follower, which manages a directional relationship between users. Three VCs are shown invoking these modules: a source-code viewer, a `git`-based wiki, and a bookmarking tool. Each VC provides a distinct interface to the same data. The Code Viewer presents syntax-highlighted source code and the results of static analysis tools such as splint [19]. Using the same MP, the wiki VC interprets text files using markdown to transform articles into HTML. Finally, the bookmarking VC leverages both MPs to give users quick access to projects owned by other users whom they follow.

Because an application's components are mutually distrustful, MPVC also leads to greater extensibility. Any of the VCs depicted in Figure 1 could be developed after the fact by someone other than the author of the MPs. Anyone who doesn't like GitStar's syntax highlighting is

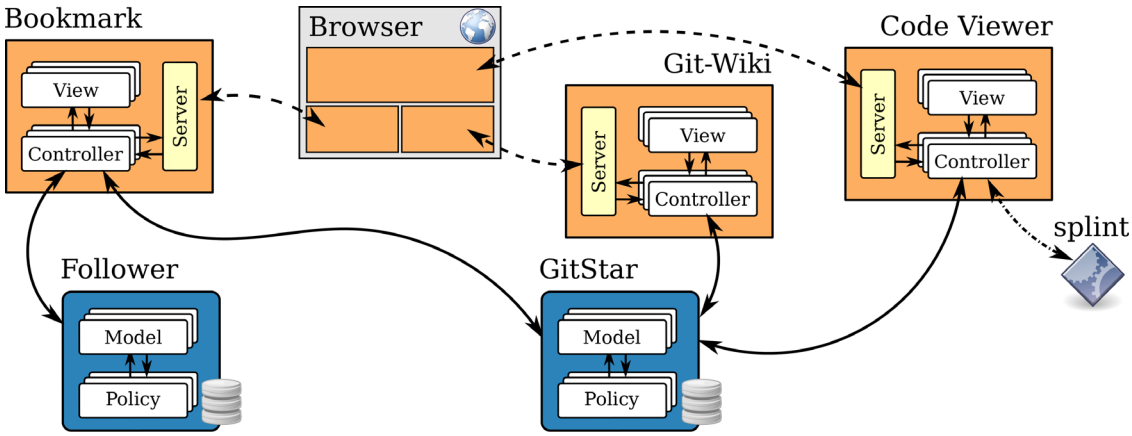



**Figure 1:** Hails platform with three VCs and two MPs. Dashed lines denote HTTP communication; solid lines denote local function calls; dashed-dotted lines denote communication with OS processes. MPs and VCs are confined at the programming language level; OS processes are jailed and only communicate with invoking VCs; the Browser is restricted to communicating with the target VCs.

free to run a different code viewer. No special privileges are required to access an MP's API, because Hails's MAC security continues to restrict what code can do with data even after gaining access to the data.

## 2.1 Principals and privileges

Hails specifies policy in terms of *principals* who are allowed to read or write data. There are four types of principal. Users are principals, identified by user-names (e.g., `alice`). Remote web sites that an app may communicate with are principals, identified by URL (e.g., `http://maps.google.com:80/`). Each VC has a unique principal, by convention starting with prefix "`@`", and each MP has a unique principal starting "`_`" (e.g., `@Bookmark` and `_GitStar` for the components in Figure 1).

An example policy an MP may want to enforce is "user `alice`'s mailing address can be read only by `alice` or by `http://maps.google.com:80/`." Such a policy would allow a VC to present `alice` her own address (when she views her profile) or to fetch a google map of her address and present it to her, but not to disclose the address or map to anyone else. For maximum flexibility, read and write permissions can each be expressed using arbitrary conjunctions and disjunctions of principals. Enforcing such policies requires knowing what principals an app represents locally and what principals it is communicating with remotely.

Remote principals are ascertained as one would expect. Hails uses a standard cookie-based authentication facility; a browser presenting a valid session cookie represents the logged-in user's principal. When VCs or MPs initiate outgoing requests to URLs, Hails considers the remote server to act on behalf of the URL principal of the web site.

Within the confines of Hails, code itself can act on behalf of principals. The trusted Hails runtime supports unforgeable objects called *privileges* with which code can assert the authority of principals. Hails passes appropriate privilege objects to MPs and VCs upon dynamically loading their code. For example, the GitStar MP is granted the `_GitStar` privilege. When a user wishes to use GitStar to manager her data, the policy on the data in question must specify `_GitStar` as a reader and writer so as to give GitStar permission to read the data and write it to its database should it chose to exercise its `_GitStar` privileges.

## 2.2 Labels and confinement

Hails associates a security policy with every piece of data in the system, specifying which principals can read and write the data. Such policies are known as *labels*. The particular labels used by Hails are called *DC labels*. We described and formalized DC labels in a separate paper [39], so limit our discussion to a brief overview of their format and use in MAC. We refer readers to the full DC labels paper for more details.

A DC label is a pair of positive boolean formulas over principals: a *secrecy* formula, specifying who can read the data, and an *integrity* formula, specifying who can write it. For example, a file labeled $\langle \texttt{alice} \vee \texttt{bob}, \texttt{alice} \rangle$ specifies that `alice` or `bob` can read from the file and only `alice` can write to the file. Such a label could be used by the Code Viewer of Figure 1 when fetching `alice`'s source code. The label allows the VC to present the source code to the project participants, `alice` and `bob`, but not disseminate it to others.

The trusted runtime checks that remote principals satisfy any relevant labels before permitting communication.

For instance, data labeled $\langle \texttt{alice} \vee \texttt{bob}, \texttt{alice} \rangle$ cannot be sent to a browser whose only principal is `charlie`. The actual checks performed involve verifying logical implications. Data labeled $\langle S, I \rangle$ can be sent to a principal (or combination of principals) $p$ only when $p \Longrightarrow S$. Conversely, remote principal $p$ can write data labeled $\langle S, I \rangle$ only when $p \Longrightarrow I$. Given these checks, $\langle \text{TRUE}, \text{TRUE} \rangle$ labels data readable and writable by any remote principal, i.e., the data is public, while $p = \text{TRUE}$ means a remote party is acting on behalf of no principals.

The same checks would be required for local data access if code had unrestricted network access. Hails could only allow code to access data it had explicit privileges to read. For example, code without the `alice` privilege should not be able to read data labeled $\langle \texttt{alice}, \text{TRUE} \rangle$ if it could subsequently send the data anywhere over the network. However, Hails offers a different possibility: code without privileges can read data labeled $\langle \texttt{alice}, \text{TRUE} \rangle$ so long as it first gives up the ability to communicate with remote principals other than `alice`. Such communication restrictions are the essence of MAC.

To keep track of communication restrictions, the runtime associates a *current label* with each thread. The utility of the current label stems from the transitivity of a partial order called "*can flow to*." We say a label $L_1 = \langle S_1, I_1 \rangle$ *can flow to* another label $L_2 = \langle S_2, I_2 \rangle$ when $S_2 \Longrightarrow S_1$ and $I_1 \Longrightarrow I_2$—in other words, any principals $p$ allowed to read data labeled $L_2$ can also read data labeled $L_1$ (because $p \Longrightarrow S_2 \Longrightarrow S_1$) and any principals allowed to write data labeled $L_1$ can also write data labeled $L_2$ (because $p \Longrightarrow I_1 \Longrightarrow I_2$).

A thread can read a local data object only if the object's label can flow to the current label; it can write an object only when the current label can flow to the object's. Data sent over the network is always protected by the current label. (Data may originate in a labeled file or database record but always enters the network via a thread with a current label.) The transitivity of the *can flow to* relation ensures no amount of shuffling data through objects can result in sending the data to unauthorized principals.

A thread may adjust the current label to read otherwise prohibited data, only if the old value can flow to the new value. We refer to this as *raising* the current label. Allowing the current label to change without affecting security requires very carefully designed interfaces. Otherwise, labels themselves could leak information. In addition, threads could potentially leak information by not terminating (so called "termination channels") or by changing the order of observable events (so called "internal timing channels"). GitStar is the first production system to address these threats at the language level. We refer interested readers to [41] for the details and security proof of our solution.

A final point is that Hails prevents the current label from accumulating restrictions that would ultimately prevent the VC from communicating back to the user's browser. In MAC parlance, a VC's *clearance* is set according to the user making the request, and serves as an upper bound on the current label. Thus, an attempt to read data that could never be sent back to the browser will fail, confining observation to a "need-to-know" pattern.

## 2.3 Model-Policy (MP)

Hails applications rely on MPs to define the application's data model and security policies. An MP is a library with access to a dedicated database. The MP specifies what sort of data may be stored in the database and what access-control policies should be applied to it. Though MPs may contain arbitrary code, we provide and encourage the use of a DSL, described in Section 2.3.1, for specifying data policies in a concise manner.

The Hails database system is similar to and built atop MongoDB [7]. A Hails *database* consists of a set of *collections*, each storing a set of *documents*. In turn, each document contains a set of *fields*, or named values. Some fields are configured as *keys*, which are indexed and identify the document in its collection. All other fields are non-indexed *elements*.

An MP restricts access to the different database layers using labels. A static label is associated with every database, restricting who can access the collections in the database and, at a coarse level, who can read from and write to the database. Similarly, a static label is associated with a collection, restricting who can read and write documents in the collection. The collection label additionally serves the role of protecting the keys that identify documents—a computation that can read from a collection can also read all the key values.

### 2.3.1 Automatic, fine-grained labeling

In many web applications, dynamic fine-grained policies on documents and fields are desired. Consider the user model shown in Figure 2: each document contains fields corresponding to a user-name, email address, and list of friends. In this scenario, the Follower MP may configure user-names as keys in order to allow VCs to search for `alice`'s profile. Additionally, the MP may specify database and collection labels that restrict access to documents at a coarse grained level. However, these static labels are not sufficient to enforce fine grained dynamic policies such as "only `alice` may modify her profile information" and "only her friends (`bob`, `joe`, etc.) may see her email address."

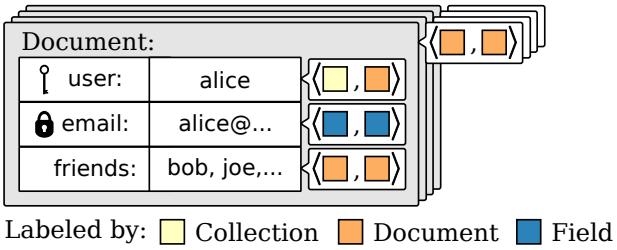



**Figure 2:** Hails user documents. Each document is indexed by a key (`user`-name) and contains the user's `email` address and list of `friends`. Documents and `email` fields are dynamically labeled using a data-dependent policy; the secrecy of the `user` key and is protected by the static collection label, the document label protects its integrity. The "unlabeled" `friends` fields are protected by their corresponding document labels.

Hails introduces a novel approach to specifying document and field policies by assigning labels to documents and fields as a function of the document contents itself.[1] This approach is based on the observation that, in many web applications, the authoritative source for who should access data resides in the data itself. For example, in Figure 2, the user-name and friends field values can be used to specify the document and field policies mentioned above: `alice`'s document is labeled $\langle \text{TRUE}, \texttt{alice} \vee \texttt{\_Follower} \rangle$, while the email field value is labeled $\langle \texttt{alice} \vee \texttt{bob} \vee \texttt{joe} \vee \cdots \vee \texttt{\_Follower}, \text{TRUE} \rangle$. The document label guarantees that only `alice` or the MP can modify any of the constituent fields. The label on the email-address field additionally guarantees that only `alice`, the MP, or her friends can read her address.

Hails's data-dependent "automatic labeling" simplifies reasoning about security policies and localizes label logic to a small amount of source code. Figure 3 shows the implementation of the Follower users policy, as described above, using our DSL. Specifying static labels on the database and collections is simply done by setting the respective `readers` and `writers` in the `database` and `collection` sections. Similarly, setting a document or field label is done using a function from the document itself to a pair of `readers` and `writers`.

### 2.3.2 Database access and policy application

MP policies are applied on every database insert. When a thread attempts to insert a document into an MP collection, the Hails runtime first checks that that the thread can read and write to the database and collection, by comparing the thread's current label with that of the database and collection. Subsequently, the field- and document-labeling policy functions are applied to the document and fields. If the policy application succeeds—it may fail if the thread cannot label data as requested—the Hails runtime removes all the labels on the document and performs the write.

[1] These labeling functions are pure: they cannot perform side effects and must always return the same value for the same input.

```
database $ do
  -- Set database label:
  access $ do
    readers ==> anybody
    writers ==> anybody

-- Set policy for new "users" collection:
collection "users" $ do
  -- Set collection label:
  access $ do
    readers ==> anybody
    writers ==> anybody
  -- Declare user field as a key:
  field "user" key
  -- Set document label, given document doc:
  document $ λdoc -> do
    readers ==> anybody
    writers ==> ("user" `from` doc) \/ _Follower
  -- Set email field label, given document doc:
  field "email" $ labeled $ λdoc -> do
    readers ==> ("user" `from` doc)
                \/ fromList ("friends" `from` doc)
                \/ _Follower
    writers ==> anybody
```

**Figure 3:** DSL-specification of the Follower users policy. Here, `anybody` corresponds to the boolean formula TRUE; `fromList` converts a list of principals to a disjunction of principals; and, `"x" `from` doc` retrieves the value of field `x` from document `doc`. The `database` and `collection` labels are static. Field `user` is configured as a key. Finally, each document and `email` field is labeled according to a function from the document itself to a set of `readers` and `writers`.

Hails also allows threads to insert already-labeled documents (e.g., documents retrieved from another MP or directly from the user). As before, when inserting a labeled document, the MP database and collection must be readable and writable at the current label. Different from above, the thread does not need to apply the policy functions; instead, the Hails runtime verifies that the labels on fields and the document agree with those specified by the MP. Finally, if the check succeeds, the Hails runtime strips the labels and performs the write.

Application components, including VCs, can fetch elements from an MP's database collection by specifying a query predicate. Predicates are restricted to solely involve indexed keys (or be TRUE). Similar to insert, when performing a fetch, the runtime first checks that that the thread can read from the database and collection. Next, the documents matching the predicate are retrieved from

the database. Finally, the field- and document-labeling policy functions are applied to each document and field; the resultant labeled documents are returned to the invoking thread.

Hails supports additional database operations, including update and delete. These operations are similar to those of MongoDB [7], though Hails enforces the MP's policies whenever its database is accessed. Since the restrictions on most operations are similar to those of insert and fetch, we do not describe them further.

## 2.4 View-Controller (VC)

VCs interact with users. Specifically, controllers handle user requests, and views present interfaces to the user. However, VCs do not define database-backed models. Instead, a controller invokes one or more MPs when it needs to store or retrieve user data. This data can also be passed on to views when rendering user interfaces.

Each VC is a standalone process, linked against the MP libraries it depends on to provide a data model. The VC author solely provides a definition for a *main* controller, which is a function from an HTTP request to an HTTP response. This function may perform side-effects: it may access a database-backed model by invoking an MP, read files from the labeled filesystem, etc. Hails uses language-level confinement to prevent the VC and MPs it invokes from modifying or leaking data in violation of access permissions. Additionally, since each VC is a process, OS-level isolation and resource management mechanisms can be leveraged to enforce additional platform-specific policies.

At the heart of every VC is the Hails HTTP server. The server, a privileged part of the trusted computing base (TCB), receives HTTP requests and invokes the main VC controller to process them. When a request is from an authenticated user, the server sets the `X-Hails-User` header to the user-name and attests to the request's contents for the benefit of MPs that care about request provenance and integrity. In turn, the main controller processes the supplied request, by potentially calling into MPs to interact with persistent state, and finally returns an HTTP response. The server returns the provided response to the browser on the condition that it depend only on data the user is permitted to observe.

In carrying out their duties, many VCs rely on communication with external web sites. Hence, Hails applications have access to an HTTP client. Before establishing a connection, and on each read or write, the HTTP client checks that the current label of the invoking thread is compatible with the remote server principal. In practice, this means VCs can only communicate with external hosts when they have not read any sensitive data or they have only read data *explicitly* labeled for the external server.

Additionally, VCs may need to run arbitrary programs. For example, as highlighted in Figure 1, GitStar's Code Viewer relies on splint, a standalone C program, to flag possible coding errors. Addressing this need, Hails provides a mechanism for spawning confined Linux processes with no network access, no visibility of other processes, and no writable file system shared by other processes. Each such processes is governed by a fixed label, namely the VC's current label at the time the program was spawned. In turn, labeled file handles can be used to communicate with the process, subject to the restrictions imposed by the current thread's label.

## 2.5 Life-cycle of an application

In this section, we use GitStar's deployment model to illustrate the life-cycle of a Hails application from development, through deployment, to a user-request.

### 2.5.1 Application development and deployment

A third-party application developer may introduce a new data model to the GitStar platform by writing an MP. For example, the Follower MP shown earlier specifies a data-model for storing a relation between users, as well as a policy specifying who is able to read, create and modify those relationships. Once written, the developer uploads the library code to the GitStar servers where it is compiled and installed. The platform administrator generates a unique privilege for the new MP and associates it with a specific database in a globally-accessible configuration file. Subsequently, any Hails code may import the MP, which when invoked, will be loaded with its privilege and database-access.

The third-party developer may build a user interface to the newly-created model by writing a VC controller. As with MPs, developers upload their VC code to the GitStar servers where it is compiled and linked against any MPs it depends on. Thereafter, a program called `hails`, which contains the Hails runtime and HTTP server, is used to dynamically load the main VC controller and service user requests on a dedicated TCP port.

While in this example both the VC and MP were implemented by a single developer, third-party developers can implement applications consisting solely of a VC that interacts with MPs created by others. In fact, in GitStar, most applications are simply VCs that use the GitStar MP to manage projects and retrieve `git` objects. For example, the `git`-based wiki application, as shown in Figure 1, is simply a VC that displays formatted text from a particular branch of a `git` repository.

### 2.5.2 An example user request

When an end-user request is sent to the GitStar platform, an HTTP proxy routes the request to the appropriate VC HTTP server based on the hostname in the request.

The Hails server receiving the forwarded request invokes the main controller of the corresponding VC in a newly spawned thread. The controller is executed with the VC's privileges and sanitized request. The HTTP server sanitizes the incoming request by removing headers such as `Cookie`; it also sets the `X-Hails-User` header to the user-name, if the request is from an authenticated user.

The main controller may be a simple request handler that returns a basic HTML page without accessing any sensitive data (e.g., an index or about page). A more interesting VC may access sensitive user data from an MP database before computing a response. In this case, the VC invokes the MP by performing a database operation such as insert or fetch. The invocation consists of several steps. First, the Hails runtime instantiates the MP with its privilege and establishes a connection to the associated database, as specified in the global configuration file. Then, the MP executes the database operations supplied by the VC, and, in coordination with the Hails runtime, labels the data according to its policies. While some database operations are not sensitive (e.g., accessing a public `git` repository in GitStar), many involve private information. In such cases, the database operation will also "raise" the current label of the VC, and thereby affect all its future communication.

When a VC produces an HTTP response, the runtime checks that the current label, which reflects all data accesses or other sensitive operations, is still compatible with the end-user's browser. For example, if `alice` has sent a request to the Code Viewer asking for code from a private repository, the response produced by Code Viewer will only be forwarded by the Hails server if the final label of Code Viewer can flow to $\langle$`alice`, TRUE$\rangle$

On the client side, the Hails browser extension, detailed in Section 3.3, restricts all incoming responses and outgoing requests according to the response label. For example, if the Code Viewer returns a response labeled $\langle$`alice`$\vee$ `http://code.google.com`, TRUE$\rangle$, the rendered page may retrieve scripts for prettifying code from `http://code.google.com`, but not retrieve images from `http://haskell.org`. On the other hand, a publicly labeled response imposes no restrictions on the requests triggered by the page.

## 2.6 Trust assumptions

The Hails runtime, including the confinement mechanism, HTTP server, and libraries are part of the TCB. Parts of the system, namely our labels and confinement mechanism, have been formalized in [30, 39–41]. We remark that different from other work, our language-level concurrent confinement system is sound even in the presence of termination and timing covert channels [41]. However, similar to other MAC systems (e.g., [24]), we assume that the remaining Hails components are correct and that the underlying OS and network are not under the control of an attacker.

By visiting a web page, the MPs invoked by the VC presenting the page are trusted by users to preserve their privacy. This is a consequence of MPs being allowed to manage all aspects of their database. However, one MP cannot declassify data managed by another, and thus users can choose to use trustworthy MPs. Facilitating this choice, Hails makes the MP policies and dependency relationships between VCs and MPs available for inspection.

Since a user can choose to invoke a VC according to the MPs it depends on, VCs are *mostly* untrusted. On the server-side, VCs cannot exfiltrate user data from the database without collusion from an MP the user has trusted. Nevertheless, VCs cannot be considered completely untrusted since they directly interact with users through their browser. Unfortunately, in today's browsers, even with our client-side sandbox, a malicious VC can coerce a user to declassify sensitive data.

# 3 Implementation

Hails employs a combination of language-level, OS-level and browser-level confinement mechanisms spread across all layers of the application stack to achieve its security goals. Most notably, we use a language-level information flow control (IFC) framework to enforce fine-grained policies on VCs and MPs. This section describes this framework, and some of the implementation details of our OS and browser confinement mechanisms.

## 3.1 Language-level confinement

Hails applications are written in Haskell. Haskell is a statically- and strongly-typed, memory-safe language. Crucially, Haskell's type system distinguishes operations involving side-effects (such as potentially data-leaking I/O) from purely-functional computation. As a consequence, for example, compiling a VC's main controller with an appropriately specified type is sufficient to assert that the VC cannot perform arbitrary network communication.

Hails relies on the safety of the Haskell type system when incorporating untrusted code. However, like other languages, Haskell "suffers" from a set of features that allow programmers to perform unsafe, but useful, actions (e.g., type coercion). To address this, we ex-

tended the Glasgow Haskell Compiler (GHC) with `Safe Haskell` [44]. `Safe Haskell`, deployed with GHC as of version 7.2, guarantees type safety by removing the small set of language features that otherwise allow programs to violate the type system and break module boundaries.

With this change, Haskell permits the implementation of language-level dynamic IFC as a library. Accordingly, we implemeted LIO [40], which employs the label-tracking and confinement mechanisms of Section 2.2. Despite sharing many abstractions with OS-level IFC systems, such as HiStar [46] and Flume [17], LIO is more fine-grained (e.g., it allows labels to be associated with values, such as documents and email addresses) and thus better suited for web applications.

We believe the Hails architecture is equally realizable in other languages, though possibly with less backward compatibility. For example, JiF [33], Aeolus [5] and Breeze [15] provide similar confinement guarantees and are also good choices. However, to use existing libraries JiF and Aeolus typically require non-trivial modifications, while Breeze requires porting to a new language. Conversely, about 4,000 modules in Hackage (27%), a popular Haskell source distribution site, are currently safe for Hails applications to import. Of course, the functions that perform arbitrary I/O are not directly useful, and, like in JiF, must be modified to run in LIO. Nevertheless, many core libraries require no modifications. Moreover, we expect the number of safe modules to grow significantly with the next GHC release, which refactors core libraries to remove unsafe functions from general-purpose modules.

## 3.2 OS-level confinement

Hails uses Linux isolation mechanisms to confine processes spawned by VCs. These techniques are not novel, but it is important that they work properly. Using *clone* with the various `CLONE_NEW*` flags, we give each confined process its own mount table and process ID namespace, as well as a new network stack with a new loopback device and no external interfaces. Using a read-only bind-mount and the tmpfs file system, we create a system image in which the only writable directory is an empty `/tmp`. Using cgroups, we restrict the ability to create and use devices and consume resources. With `pivot_root` and `umount`, we hide filesystems outside of the read-only system image. The previous actions all occur in a setuid root wrapper utility, which finally calls `setuid` and drops capabilities before executing the confined process.

## 3.3 Browser-level confinement

VC responses are protected from inappropriate leaks on the client side using a sandbox. The sandbox, implemented as a browser extension for chrome, intercepts all network communication. In turn, all requests triggered by the page are allowed only if they are guaranteed to not leak information.

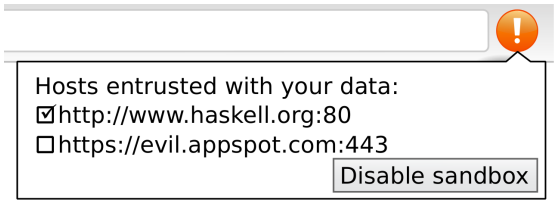



**Figure 4:** Hails client sandbox configuration. Users may (dis)allow communication to explicit hosts when the page label does not permit the flow directly.

The Hails client-side sandbox arbitrates traffic according to the label of the page, which is analogous to a server-side thread label. The Hails HTTP server sends the header `X-Hails-Label` with every VC response containing the initial page label, i.e., the label of response. As previously mentioned, if the page label is public, the sandbox does not impose any restrictions on the external requests triggered by the page. If the page label is not public, the sandbox only allows a request to a remote host if the page label is compatible with the principal implied by the remote host name. For instance, an image will be fetched from `maps.google.com` and a link will be followed to `hackage.haskell.org` if the page label is $\langle$`alice`$\vee$ `http://maps.google.com:80/` $\vee$ `http://hackage.haskell.org:80/`, TRUE$\rangle$. However, an XMLHttpRequest to `evil.appspot.com` will not be allowed. Similarly, if the page was instead labeled $\langle$`alice`, TRUE$\rangle$ the sandbox would reject all requests.

Users may approve otherwise disallowed network communication at the risk of potentially leaking their sensitive data to designated remote hosts. The first time a request to a disallowed host is intercepted, our extension requires the user to intervene. Specifically, the user is alerted and asked to approve network communication to the host in question. Clicking "No" blocks network access to the host for that `iframe` or tab. (The user can still view the contents of the page, except for resources, such as images or style-sheets, from the blocked host.) Conversely, clicking "Yes" allows the page to load normally; however, as illustrated in Figure 4, an icon is used to warn the user of a potential leak. In both cases, the user decision is saved for future requests and may easily be changed, as also highlighted in Figure 4.

The client-side sandbox is the least satisfying aspect of Hails's security, in part because it requires each user to install a new extension. In Section 7 we discuss the limitations of our current extension and future research

directions that could help address leaking sensitive data through the browser. Here, we finally remark that standards proposals such as Mozilla's CSP [42] show that browser vendors are open to incorporating mechanisms that coordinate with web servers to enforce security policies. Addressing data leaks on the server-side first, with systems like Hails, will help compel changes in tomorrow's browsers.

# 4 Applications

We built and deployed `GitStar.com`, a Hails platform centered around source code hosting and project management. We and others have authored a number Hails applications for the GitStar platform. Below we detail some of these applications including the core interface, a code viewer, follower application, wiki and messaging system.

**GitStar** At its core, GitStar includes a basic MP and VC. The MP manages users' SSH public-keys, project membership and project meta-data such as name and description; the VC provides a simple user interface for managing such projects and users.

Since Hails does not have built-in support for `git` or SSH, the GitStar platform includes an SSH server (and `git`'s transport utilities) as an external service. Our modified SSH server queries the GitStar VC when authenticating users and determining access control for repositories. Conversely, the GitStar MP communicates with an HTTP service atop this external `git`-repository server to access `git` objects.

GitStar allows users to create projects to which they can push files via `git`. Projects may be public (anyone can view or checkout repository contents) or private, in which case only specific users identified as *readers* or *collaborators* may access the project. In both cases, only collaborators may *push* contents to the project repository. GitStar provides an interface for managing these settings.

The rest of `GitStar.com` is provided by separately-administered, mutually-distrustful Hails applications, some of which were written by third-party developers. Each application is independently accessible through a unique subdomain of `GitStar.com`. When a user "installs" an application in a project, GitStar creates a link on the project page that embeds an `iframe` pointing to the application. This gives third-party applications a first-class role in extending the user experience.

**Code Viewer** One of the most useful features of source-code hosting sites is the ability to browse a project's code. We have implemented a code-viewing VC that allows users to navigate to different branches in a project's repository, view syntax-highlighted code, etc. Source code markup is done on the client-side using Google's Prettify JavaScript library [14]. Additionally, if the source file is written in C or Haskell, the VC provides the user with an option to see the output of static-analysis tools splint [19] and hlint [29], respectively.

Like all third-party applications, the Code Viewer is untrusted and accesses repository contents through the GitStar MP. When accessing objects in a private repository, the GitStar MP changes the VC's current label to restrict communication to authorized readers of the repository. Note that this may also restrict the VC from subsequently writing to the database.

**`git`-based Wiki** The `git`-based Wiki displays Markdown files from the "wiki" branch of a project repository as formatted HTML. It uses the pandoc library [25] to convert Markdown to HTML. Like the Code Viewer, the wiki VC accesses source files through the GitStar MP, meaning it cannot show private wiki pages to the wrong users. This application leverages functionality originally intended for the Code Viewer for different purposes, demonstrating the power of separating policies from application logic.

**Standalone Wiki** The standalone wiki is similar to the `git`-based Wiki, except that pages are stored directly in a database rather than in files checked into `git`. To accomplish this, the developer wrote both an MP and a VC. The MP stores a mapping between project names and wiki pages. Wiki pages are labeled dynamically to allow project readers and collaborators to read and write wiki pages. This is different from the `git`-based Wiki in that it allows a more relaxed policy: readers can create and modify wiki pages. Moreover, it is a concrete example of one MP that depends on another (namely the GitStar MP).

**Follower** GitHub introduced the notion of "social coding," which combines features from social networks with project collaboration. This requires that a user be able to "follow" other users and projects. GitStar does not provide this feature natively, but a Follower MP has been developed to manage such relationships. Users may now add the "Bookmark" application (implemented as a VC) to their project pages, which allows other users to add the project to their list of followed repositories.

**Messenger** The Messenger application provides a simple private-messaging system for users. Its MP, as implemented by the developer, defines a message model and policies on the messaging data. The policy allows any user to create a message, but restricts the reading of a message to the sender and intended recipient. Interfacing with the Messenger MP, the Messenger VC provides a page where users may compose messages, and a separate page

where they may read incoming messages.

# 5 Design Patterns

In this section, we detail the applicability of some existing security patterns within Hails, and various design patterns that we have identified in the process of building GitStar.

**Privilege separation** Since MPs are trusted by users to protect the confidentiality and integrity of their data, a well-designed MP should be coded defensively. Moreover, an MP should treat all invoking VCs as untrusted, including ones written by the same author.

The easiest way to program defensively is to minimize use of an MP's privileges, i.e., practice separation of privilege [35]. When doing so, invoking VCs will only be able to fetch data that the end user can observe, as opposed to all data when using the MP's privileges. Similarly, this restricts VCs to inserting already-labeled documents, as discussed in Section 2.3.2. This is important as it effectively limits a VC to inserting user-endorsed data, as opposed to almost-arbitrary data when using the MP privilege.

**Trustworthy user input** VC-constructed documents cannot necessarily be trusted to represent user intentions; thus, MPs should not allow VCs to arbitrarily insert data on behalf of the user. Consider, for example, the policy of the Follower MP imposed on user documents, as given in Figure 3. Here, a VC, even one running on behalf of `alice`, should not be allowed to construct and insert the document of Figure 2, without `alice` or the MP endorsing its contents.

Since VCs do not own user privilege and, as discussed above, MPs should not grant their privileges, Hails provides a mechanism for transforming user input data to a labeled document, that retains integrity. Recall that a VC's main controller is invoked, by the Hails server, with a pre-labeled HTTP request; the label on this request has the integrity of the user (e.g., $\langle \text{TRUE}, \texttt{alice} \rangle$). If the VC directly manipulates the request to construct an appropriate document, the integrity will be stripped. Hence, Hails provides a library for transforming a labeled, URL-encoded body (e.g., submitted from an HTML form in the user's browser) into a labeled document, that MPs may expose to VCs. This transformer takes a user-endorsed request and returns an MP-endorsed document that the VC may, in turn, insert into the database.

Users must still trust VCs to construct HTML forms that will reflect their intentions. However, an MP may inspect requests before transforming them to labeled documents. Moreover, policies, such as that of Figure 3, would prevent a VC trusted only by `bob` from modifying `alice`'s data.

**Partial update** The trustworthy user input pattern is suitable for inserting and updating documents in whole; it is not, however, directly applicable to partially updating documents. Returning to the Follower user model of Figure 2, a VC that wishes to present a form for updating the user's email address would have to include all the remaining fields as hidden input variables. Though this would allow the VC to update the email field by effectively inserting a new labeled document, this approach is error prone and not scalable.

Instead, we found that a partial document that contains the newly-updated fields, the document keys, and a token `$hailsDbOp` indicating the operation (`partialUpdate`, in this case) is sufficient for the MP to update an existing document. This partial-document must be endorsed by the user or MP, by, for example, applying the previous pattern. Directly, to carry out the partial update, the MP first verifies that the user is aware of the update by checking the presence of the operation token `$hailsDbOp`. Next, the MP uses the keys to fetch the existing document and merges the newly-updated fields into the document. Finally, the document update is performed, imposing restrictions similar to those of Section 2.3.2.

**Delete** We have found that most applications use a pattern similar to the partial update pattern when deleting documents: a VC invokes an MP with a document containing the target-document's keys and an operation token indicating a delete, i.e., `$hailsDbOp` set to `delete`. As in the partial update, this document must be endorsed by the user or MP by applying the trustworthy input pattern. Directly, the VC may invoke the MP with the labeled document, who, in turn, removes the target document after inspection.

**Privilege delegation** Hails provides a call-gate mechanism, inspired by [46], with which code can authenticate itself to a called function, i.e., prove possession of privileges, without actually granting any privileges to the called function. One use of call gates is to delegate privileges. For instance, an MP can provide a gate that simply returns its own privilege, on the condition that it was called by a particular VC.

While earlier version of GitStar utilized privilege delegation, we now largely avoid it; in many cases, we found modifying the policy to be a better alternative. For instance, the early version of the GitStar VC used the GitStar MP's privilege to look up project readers and collaborators for the SSH server. Now, we simply created a user account for the SSH server and added this principal as a reader in the `project` collection policy. Nevertheless, such refactoring may not always be possible and privilege delegation may prove necessary.

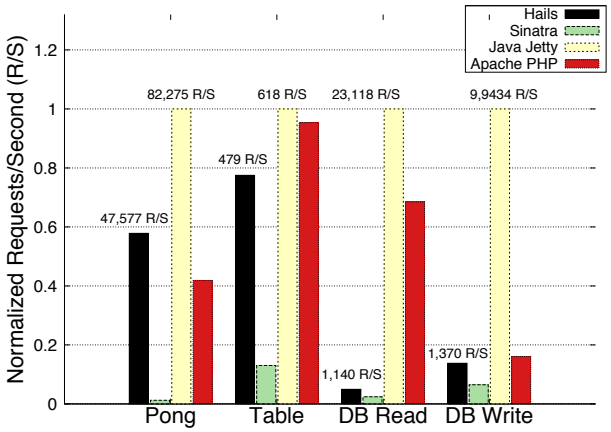



**Figure 5:** Micro-benchmarks of basic web application operations. The measurements are normalized to the Java Jetty throughput. All database operations are on MongoDB.

# 6 Evaluation

We compare the performance of the Hails framework against existing web frameworks, and report on the experience of application authors not involved in the design and implementation of the framework.

## 6.1 Performance Benchmarks

To demonstrate how Hails performs in comparison to other widely-used frameworks, we present the results of four micro-benchmarks that reflect basic operations common to web applications. Figure 5 shows the performance of Hails, compared with:

- ▷ Ruby Sinatra framework [36] on the Unicorn web server. Sinatra is a common application framework for small Ruby applications and APIs (e.g., the GitHub API is written using Sinatra).
- ▷ PHP on the Apache web server with `mod_php`. Apache+PHP is one of the most widely deployed technology for web applications, including WordPress blogs, Wikipedia, and earlier versions of Facebook.
- ▷ Java on the Jetty web server [10]. Jetty is a container for Oracle's Java Servlet specification, and is widely used in production Java web-applications including Twitter's streaming API, Zimbra and Google AppEngine.

We use `httperf` [31] to measure the throughput of each server setup when 100 client connections continuously make requests in a closed-loop—we report the average responses/second. The client and server were executed on separate machines, each with two Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM, connected over a Gigabit local network.

In the Pong benchmark the server simply responds with the text "PONG". This effectively measures the throughput of the web server itself and overhead of the framework. Hails responds to 1.7× fewer requests/second than Jetty. However, the measured throughput of 47,577 requests/second is roughly 28% and 47× higher than Apache+PHP and Sinatra, respectively.

In the Table benchmark, the server dynamically renders an HTML table containing 5,000 entries, effectively measuring the performance of the underlying language. Hails respectively responds to 30% and 23% fewer requests/second than Jetty and Apache+PHP, but 6× more than Sinatra. Hails is clearly less performant than Jetty and Apache+PHP for such workloads, even though Haskell should be faster than PHP at CPU workloads. We believe that this is primarily because Hails does not allow pipelined HTTP responses, so a large response body must be generated in memory and sent in its entirety at once (as opposed to sent in chunks as output is available). Nonetheless, Hails responds to 6× more requests/second than Sinatra.

The DB Read and DB Write benchmarks compare the performance of the read and write database throughput. Specifically, for the DB Read benchmark the server responds with a document stored in the MongoDB, while for the DB Write the server inserts (with MongoDB's `fsync` and `safe` settings on) a new document into a database collection and reports success. Like the Ruby library, the Haskell MongoDB library does not implement a connection pool, so we loose significant parallelism in the DB Read workload when compared to Jetty and Apache+PHP. In the DB Write workload, this effect is obviated since the `fsync` option serializes all writes.

## 6.2 Experience Report

We gathered experience reports from four developers that used Hails to build applications. Their reflections validate some of the design choices we made in Hails, as well as highlight some ways in which we could make Hails applications easier to write.

We conjectured that separating code into MPs and VCs leads to building applications for which it is easier to reason about security. This was validated by the application authors who remarked that although "experienced developers [need to] write the tough [MP] code and present a good interface," when compared to frameworks such as Rails, not having to "sprinkle [security] checks in the controller" made it easier to be sure that "a check was not missing." With Hails, they, instead, "spent time focusing on developing the [VC] functionality." We further found that developers (ourselves included) had a number of mass-assignment bugs in VC code [34]. Different from [20, 34], these bugs did not prove to be vulnerabilities in GitStar—the policies specified by the GitStar MP

trivially prevents attacks wherein one user tries to impersonate another. Though such vulnerabilities can be addressed differently, we found that similar bugs are easy to introduce and a well-specified policy can prevent them from becoming privacy concerns.

Implementing MPs using an earlier version of Hails proved challenging for most of the developers. In particular, while devising policies for a model was generally straightforward, developers felt that the API for actually implementing policy modules was difficult to learn. In fact, inspecting the code of one of the blog applications, we found that the developer had a bug that leaked blog posts regardless of whether the user decided to publish the blog post or not. This bug was a result of inadvertently making posts, as opposed to just post IDs, keys. We believe that this was due in part due to the terse policy-specification API.

Addressing the challenges with specifying policies, we designed the DSL presented in Section 2.3. We found this DSL to make policy specification much simpler. Equally important, developers have found it easier to understand what an MP enforces and thus make a more informed decision when deciding to use the library.

We are actively working on improving the Hails development experience. Compared to other frameworks, Hails needs more "good documentation with recipes." Developers found that the lack of "scaffolding tools for generating boiler-plate code [and] a template framework" impedes the development process. Part of our ongoing work includes building scaffolding tools for both VCs and MPs, adopting a templating language, and creating additional tutorials that illustrate typical application development.

# 7 Discussion and Limitations

In this section, we discuss the ramifications of the design and implementation of Hails and suggest solutions to some of its limitations.

**Browser-level confinement** As previously noted, we cannot expect all users to install the Hails browser extension which provides confinement in the browser. A different approach would be to re-write VC output at the server-side before sending it to the client, neutralizing data-exfiltration risks. Until recently, such content-rewriting was a dangerous proposition. In particular, Google [28], Yahoo [11], Facebook [13], and Microsoft [16] have all developed technology to constrain the effects of third-party web content such as advertisements; but the design of existing browser interfaces made those tools vulnerable to attack [26].

However, ECMAScript 5 Strict mode, now supported by most browsers, makes the prospect of safe re-writing far more tractable. For instance, SES [43], one promising approach with solid theoretical foundations, can now be implemented in about 200 lines of JavaScript. Though SES is not compatible with popular JavaScript libraries such as jQuery, this may well change. In our preliminary experimentation with Caja [28], a system which influenced SES, we successfully sandboxed VC responses in a similar fashion to our browser extension. Hence, if we cannot get traction from the browser vendors with our custom HTTP header, in the future we will experiment with a server-side filter that parses and regenerates HTML (so as to sanitize URLs in `src` and `href` attributes), and enforces JavaScript confinement with SES.

**Query interface** Hails queries are limited to expressions on keys. By separating keys from elements, the decision to permit a query is simple: if a Hails component can read from the database collection, it may perform a key-based query. This limited interface is sufficient for many VCs, which may perform further refinement of query results by inspecting labeled fields in their own execution contexts.

For larger datasets, better performance would result from filtering on all relevant fields in the underlying database system itself. Additionally, this would obviate the need to reason about the security semantics of keys. However, providing this more-general interface to a Hails application would require sensitivity to label policies inside the query engine. Since Hails builds atop MongoDB, which provides a JavaScript interface, we hope to compile policies to code that can implement the necessary label-checking logic.

# 8 Related Work

**Information flow control and web applications** A series of work based on Jif addresses security in web applications. SIF (Servlet Information Flow) is a framework that essentially allows programmers to write their web applications as Servlets in Jif [9]. Swift [8], based on Jif/split [45, 47], compiles Jif-like code for web applications into JavaScript code running on the client-side and Java code running on the server by applying a clever partitioning algorithm. SIF and Swift do not support information flow control with databases or untrusted executables; on the other hand, Hails provides weak security guarantees on the client side.

Ur/Web [6] is a domain specific language for web application development that includes a static information flow analysis called UrFlow. Policies are expressed in the form of SQL queries and while statically enforced, can depend

on dynamic data from the database. Security can also be enforced on the client side in a similar manner to Swift, with Ur/Web compiling to both the server and client. A crucial difference from Hails is that Ur/Web does not aim to support a platform architecture consisting of mutually distrustful applications as Hails does. Moreover, Hails is more amendable to extensions such as executing untrusted binaries or scaling to a distributed setting.

Logical attestation [37] allows specifying a security policy in first-order logic and the system ensures that the policy is obeyed by all server-side components. This system was implemented as a new OS, called Nexus. Hails's DC labels are similar to Nexus' logical attestation, but based on a simpler logic, namely propositional logic. A crucial difference between the Nexus OS [37] and Hails is that we provide very fine grained labeling and a framework for separating data-manipulating code from other application logic at the language level. For a web framework, fine grained policies are desirable; the language-level approach also addresses the limitations of cobufs used in Nexus [37]. Moreover, requiring users to install a new OS as opposed to a library is not always feasible. Nevertheless, their work is very much complimentary: GitStar can potentially use Nexus to execute untrusted executables in an environment that is less restricting that our Linux jail (e.g., it could have network access as directed by Nexus).

The closest related work to Hails is W5 [18]. Similar to Hails, they propose a separation of user data and policies (MPs), from from the application logic (VCs). Moreover, they propose an architecture that, like Hails, uses IFC to address issues with current website architectures. W5's design is structured around OS-level IFC systems. This approach is less flexible in being coarser grained, but, like Nexus, complimentary. A distinguishing factor from W5 is our ability to report on the implementation and evaluation of production system.

**Trust management** Trust Management is an approach to distributed access control and authorization, popularized in [2]. Related work includes [1, 3, 12, 21, 22]. One central idea in trust management, which we follow in the present paper, is to separate policy from other components of the system. However, trust management makes access control decisions based on policy supplied by multiple parties; in contrast, our approach draws on information flow concepts, avoiding the need for access requests and grant/deny decisions.

**Persistent storage** Li and Zdancewic [23] enforce information flow control in PHP programs that interact with a relational database. They statically indicate the types of the input fields and the results of a predetermined number of database queries. In contrast, Hails allows arbitrary queries on keys and automatically infers the security levels of the returned results.

Extending Jif, Fabric [24] is an IFC language that is used to build distributed programs with support for data stores and transactions. Fabric safely stores objects, with exactly one security label, into a persistent storage consisting of a collection of objects. Different from Fabric, Hails store units (documents) can have different security labels for individual elements. Like Fabric, Hails can only fetch documents based on key fields.

BStore [4] separates application and data storage code in a similar fashion to Hails's separation of code into VCs and MPs. Their abstraction is at the file system granularity, enforcing policies by associating labels with files. Our main contribution provides a mechanism for associating labels with finer grained objects—namely Haskell values. We believe that BStore is complimentary since they address similar issues, but on the client side.

# 9 Conclusion

Ad-hoc mechanisms based on access control lists are an awkward fit for modern web frameworks that incorporate third-party software components but must protect user data from inappropriate modification or sharing. By applying confinement mechanisms at the language, OS, and browser levels, Hails allows mutually-untrusted applications to interact safely. Because the framework promotes data-flow policies to first-class status, authors may specify policy concisely in one place and be assured that the desired constraints on confidentiality and integrity are enforced across all components in the system, in a mandatory fashion, whatever their quality or provenance.

As a demonstration of the expressiveness of Hails, we built a production system, GitStar, whose central function of hosting source-control repositories with user-configurable sharing is enriched by various third-party applications for viewing documents and collaborating within and between development projects. Through our active use of this system and the experience of other developers who built VCs and MPs for it, we were able to confirm the ability of the framework to support a modular system of heterogeneously-trusted software components that nevertheless can enforce flexible data-protection policies demanded by real-world users.

# Acknowledgments

We thank Amy Shen, Eric Stratmann, Ashwin Siripurapu, and Enzo Haussecker for sharing their Hails development experience with us. We thank Diego Ongaro, Mike Piatek,

Justine Sherry, Joe Zimmerman, our shepard Jon Howell and the anonymous reviewers for their helpful comments on earlier drafts of this paper. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, and by the Swedish research agency VR and STINT. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

# References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Oct. 1993.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 164–173, 1996.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704 (Informational), Sept. 1999. URL `http://www.ietf.org/rfc/rfc2704.txt`.

[4] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BSTORE. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 1–1, 2010.

[5] W. Cheng, D. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[6] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'10, 2010.

[7] K. Chodorow and M. Dirolf. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[8] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. pages 31–44, Oct. 2007.

[9] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, Aug. 2007.

[10] M. B. Consulting. Jetty webserver, March 2012. `http://jetty.codehaus.org/jetty/`.

[11] D. Crockford. Making JavaScript safe for advertising. `http://adsafe.org/`.

[12] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society Press, May 2002.

[13] Facebook. Fbjs (Facebook JavaScript). `http://developers.facebook.com/docs/fbjs/`.

[14] Google. Google code prettify, September 2012. `http://code.google.com/p/google-code-prettify/`.

[15] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. Exceptionally available dynamic IFC. Submitted to POPL, July 2012.

[16] S. Isaacs. Microsoft web sandbox. `http://www.websandbox.org/`.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles*, October 2007.

[18] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Atlanta, GA, November 2007.

[19] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, August 2001.

[20] L. Latif. Github suffers a Ruby on Rails public key vulnerability, March 2012. `http://www.theinquirer.net/inquirer/news/2157093/github-suffers-ruby-rails-public-key-vulnerability`.

[21] N. Li and J. C. Mitchell. RT: A role-based trust-management framework. In *The Third DARPA Information Survivability Conference and Exposition (DISCEX III)*. IEEE Computer Society Press, Apr. 2003.

[22] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, Feb. 2003.

[23] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*. IEEE Computer Society, 2005.

[24] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, , and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[25] J. MacFarlane. Pandoc:a universal document converter. `http://johnmacfarlane.net/pandoc/`.

[26] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 77–91, 2009.

[27] J. Mayer and J. Mitchell. Third-party web tracking: Policy and technology. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 413–427, 2012.

[28] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`, June 2008.

[29] N. Mitchell. *HLint Manual*. `http://community.haskell.org/~ndm/darcs/hlint/hlint.htm`.

[30] B. Montagu, B. Pierce, R. Pollack, and A. Surée. A theory of information-flow labels. *Draft, July*, 2012.

[31] D. Mosberger and T. Jin. httperf-a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[32] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM symposium on Operating systems principles*, pages 129–142, 1997.

[33] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[34] T. Preston-Werner. Public key security vulnerability and mitigation, March 2012. `https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation`.

[35] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[36] Sinatra. Sinatra, September 2012. `http://www.sinatrarb.com/`.

[37] E. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[38] E. Steel and G. Fowler. Facebook in privacy breach. *The Wall Street Journal*, 18, October 2010.

[39] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Proceedings of the NordSec 2011 Conference*, October 2011.

[40] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th Symposium on Haskell*, pages 95–106, September 2011.

[41] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2012.

[42] B. Sterne, M. Corporation, A. Barg, and G. Inc. Content security policy, May 2012. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.

[43] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript APIs. In *IEEE Symposium on Security and Privacy*, 2011.

[44] D. Terei, S. Marlow, S. P. Jones, , and D. Mazières. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell*, September 2012.

[45] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. Oct. 2001.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[47] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, Washington, DC, USA, 2003. IEEE Computer Society.

# Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels

Alan M. Dunn Michael Z. Lee Suman Jana Sangman Kim Mark Silberstein
Yuanzhong Xu Vitaly Shmatikov Emmett Witchel
*The University of Texas at Austin*

## Abstract

Modern systems keep long memories. As we show in this paper, an adversary who gains access to a Linux system, even one that implements secure deallocation, can recover the contents of applications' windows, audio buffers, and data remaining in device drivers—long after the applications have terminated.

We design and implement Lacuna, a system that allows users to run programs in "private sessions." After the session is over, all memories of its execution are erased. The key abstraction in Lacuna is an *ephemeral channel*, which allows the protected program to talk to peripheral devices while making it possible to delete the memories of this communication from the host. Lacuna can run unmodified applications that use graphics, sound, USB input devices, and the network, with only 20 percentage points of additional CPU utilization.

## 1. Introduction

Computers keep memories of users' activities—whether users want it or not. A political dissident may want to upload text and photos to a social media site, watch a forbidden video, or have a voice-over-IP conversation without leaving incriminating evidence on her laptop. A biomedical researcher may want to read a patient's file or run a data-mining computation on a database of clinical histories and then erase all traces of the sensitive data from his computer. You, the reader, may wish to browse a medical, adult, or some other sensitive website without your machine keeping a record of the visit.

None of the above are possible in modern computers. Traces of users' activities remain in application and OS memory, file systems (through both direct and indirect channels such as OS swap), device drivers, memories of peripheral devices, etc. [7, 12, 17, 56]. Even when applications such as Web browsers explicitly support "private" or "incognito" mode, intended to leave no evidence of users' activities on the host machine, they fail to achieve their objective because traces are kept by system components outside the application's control [1].

Secure memory deallocation (the eager clearing of deallocated memory) [8] and secure file deletion [2, 4, 23] do not completely solve the problem because they do not address the issue of a user's data remaining in long-lived shared servers (including the OS) on that user's machine. We show how to recover sensitive data—including screen images of private documents and SSH sessions—from memory that is not controlled by the application and remains allocated even after the application terminates: memory of the X server, kernel device drivers, and the mixing buffer of the PulseAudio audio server (see § 2). Furthermore, the PaX patch, a common implementation of secure deallocation for Linux [37], does not apply it pervasively and leaves sensitive data, such as buffer cache pages, in memory.

In this paper, we describe the design and implementation of Lacuna, a system that protects privacy by erasing all memories of the user's activities from the host machine. Inspired by the "private mode" in Web browsers, Lacuna enables a "private session" abstraction for the whole system. The user may start multiple private sessions, which run concurrently with each other and with non-private computer activities. Within a private session, the user may browse the Web, read documents, watch video, or listen to audio. Once the private session ends, all evidence, including application memory, keystrokes, file data, and IP addresses of network connections, is destroyed or made unrecoverable.

We use the term **forensic deniability** for the novel privacy property provided by Lacuna: after the program has terminated, an adversary with complete control of the system and ability to threaten or coerce the user, cannot recover any state generated by the program.

Lacuna executes private sessions in a virtual machine (VM) under a modified QEMU-KVM hypervisor on a modified host Linux kernel. Using a VM helps protect applications that consist of many executables communicating via inter-process communication (IPC), e.g., most modern Web browsers.

After the VM is terminated, Lacuna erases its state and all memories of its interaction with the devices. To make the latter task tractable, Lacuna introduces a new system abstraction, **ephemeral channels**. We support ephemeral channels of two types. Encrypted channels encrypt all data and erase the key when the channel is destroyed. Hardware channels transfer data using hardware, leaving no trace in host software—for example, by having a guest OS directly read and write a hardware-virtualized NIC. In both cases, application data is exposed to hundreds of lines of code rather than millions, making secure erasure feasible.

In summary, we make the following contributions:

1. Demonstrate how sensitive data from terminated applications persists in the OS kernel and user-level servers. This motivates forensic deniability as an interesting privacy property that merits system support.
2. Design and implement ephemeral channels, an abstraction that allows a host kernel and hypervisor to erase memories of programs executed within a VM.
3. Evaluate a full-system Lacuna prototype, based on Linux and QEMU-KVM, that supports any Linux or Windows program —including Web browsers, PDF readers, and VoIP clients—and provides forensic deniability for workloads simultaneously accessing the display, audio, USB keyboards, mice, and the network, with minimal performance cost, e.g., 20 percentage points of additional CPU utilization.

## 2. Remembrance of things past

In this section, we describe two new attacks that recover screen and audio outputs of applications *after* they terminate. These outputs remain in allocated buffers at user and kernel level, thus even properly implemented secure deallocation would have not erased them. We also show that a popular implementation of secure deallocation (the Linux PaX patch) does not implement it completely, leaving sensitive application data in system memory caches and compromising forensic deniability.

### 2.1 Display

The following experiments were conducted on a recent version of the Linux graphics stack: X.org X server 1.10.6 (referred to as X below), Nouveau open-source NVIDIA GPU DRI2 module 0.0.16, kernel module 1.0.0, and Linux 3.3.0 with the PaX patch.

***EXA caches in X server.*** Figure 1 is a visualization of a particular data structure found in X's heap after all applications terminated and no open windows remain on the screen. It shows the screen outputs of several applications—an SSH client, a PDF viewer, and a Web browser—that were not invoked concurrently and terminated at different times.

The availability of the entire visual state of a window from a terminated application within the memory of the X server illustrates a general point. Modern systems have deep software stacks that can retain the data of even "secure" programs running on top of them.

In this specific case, the X server allocates memory for its own use[1] as part of the EXA acceleration layer, a standard part of the modern X server architecture used by many open-source GPU drivers. EXA accelerates 2D graphical operations performed during screen updates

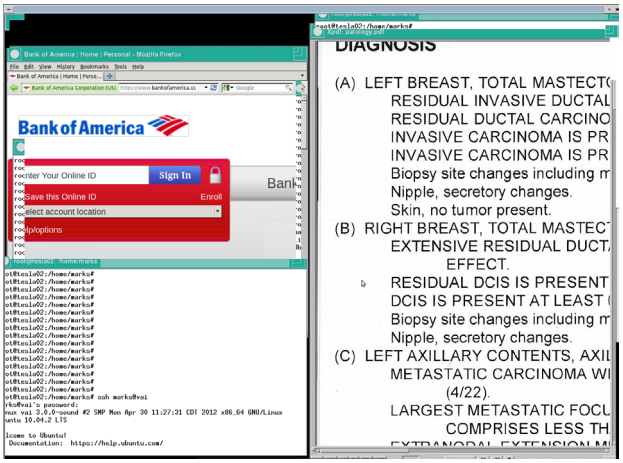



**Figure 1.** The display state of recently used applications cached in the X server after their termination.

when application windows are moved or their visibility changes. EXA uses the memory allocated by the X server as a cache—for example, to cache the bitmap representation of window contents when part of the window is obscured. When an occluding window is relocated, the exposed part of the screen is recovered by fetching the bitmap from the EXA cache instead of redrawing the entire application window (assuming that the window's contents are unchanged). The cache is not invalidated when an application terminates, and is kept allocated until the last X client terminates. Typically, the last X client is an X window manager whose termination coincides with the termination of the X server itself.

The EXA subsystem cache contains desktop contents only for certain window managers which employ 2D acceleration, such as TWM and FVMW2. We also recovered window bitmaps from an X server without any window manager. With Xfce 4 and the Gnome/Unity environments, however, this memory buffer contains only a static desktop wallpaper image. Furthermore, we observed this leak when using the open-source Nouveau graphics driver deployed by all major Linux distributions, but not with the proprietary NVIDIA driver because the latter does not use the EXA buffer.

***TTM DMA driver memory pool.*** Window contents of terminated applications can also be retrieved from kernel memory, in a way that does not depend on X's user-space behavior. We exploit the TTM module, a general memory manager for a Direct Rendering Manager (DRM)[2] subsystem used by most modern open-source GPU drivers in Linux.

The TTM module manages a DMA memory pool for transferring data between the host and GPU memories[3].

[1] `exaPrepareAccessReg_mixed()` allocates memory for each pixel on the screen (file exa/exa_migration_mixed.c, line 203). The pointer to the memory is stored in the `Client` data structure for X's own X client and referenced from the global array of pointers to the `Client` data structures for all active X clients.

[2] `http://dri.freedesktop.org/wiki/DRM`

[3] See drivers/gpu/drm/ttm/ttm_page_alloc_dma.c in the Linux kernel source.

Scanning the pages in this memory pool reveals bitmaps rendered on the screen by previously terminated applications, including the QEMU VM and VNC (used for remote access to graphical desktops).

This technique works for the Gnome/Unity environment (the current Ubuntu default) and is likely independent of the choice of window manager because all of them use the kernel modules. The lifetime of data recovered this way is measured in hours if the system is idle, but it is sensitive to the churn rate of windows on the desktop and applications' behavior. For example, the display contents of a terminated VM remain in memory almost intact after running various desktop applications, such as terminal emulator and word processor, that do relatively little image rendering. Only about half of the contents remain after invoking a new VM instance, but some remnants survive all the way until the DMA memory pool is cleared as a result of the X server's termination or virtual console switch.

We also found a similar leak with the proprietary NVIDIA driver when displaying static images outside the QEMU VM. Its lifetime was limited to about 10 minutes. Without the driver's source code, however, we are unable to identify the exact reasons for the leak.

### 2.2 Audio

Most popular Linux distributions use the PulseAudio server, which provides a uniform interface for advanced audio functions like mixing and resampling. PulseAudio uses shared memory segments of at most 64MB to communicate with applications. These segments are allocated when applications create "PulseAudio streams" by calling `pa_simple_new` and `pa_stream_new`. If an application crashes or exits without freeing its segment via `pa_simple_free` or `pa_stream_free`[4], its audio output remains in PulseAudio's memory. PulseAudio lazily garbage-collects segments whose owners have exited, but only when a new shared segment is mapped.

Sound streams recovered from PulseAudio shared segments after the application terminated are noisy because the PulseAudio client library stores memory management metadata inline with stream contents in the same segment.[5] Nevertheless, we were able to recover up to six seconds of audio generated by Skype (sufficient to reveal sensitive information about the conversation and its participants) and music players like mplayer. In general, duration of the recovered audio depends on the application's and input file's sampling rate.

### 2.3 "Secure" deallocation that isn't

***System caches.*** Not all system memory caches are explicitly freed when no longer in use, thus secure deallocation is not sufficient for forensic deniability. For example, PaX leaves file data read from disk in the system buffer cache because those pages are not freed on program exit. Buffer cache pages compromise forensic deniability even for programs inside a VM. We ran LibreOffice in an Ubuntu 11.10 guest VM on a host without LibreOffice installed, then shut down the VM and dumped the host's physical memory. Examination of the memory image revealed symbol names from the `libi18nisolang1gcc3.so` library, disclosing (with the help of `apt-file`) that LibreOffice had run.

***Network data.*** Contrary to the advice from [8], PaX does not clean `sk_buff` structures which store network packets. In general, PaX does not appear to eagerly erase any `kmem_cache` memory at all, which can completely compromise forensic deniability. For example, we visited websites with Google Chrome in private mode running inside a VM with NAT-mode networking on a PaX-enabled host. After closing Chrome and shutting down the VM, a physical memory dump revealed complete packets with IP, TCP, and HTTP headers.

## 3. Overview

The purpose of Lacuna is to execute applications within **private sessions**, then erase all memories of execution once the session is over. Lacuna runs applications in a VM which confines their inter-process communications. Applications, however, must interact with the user and outside world via peripheral devices. If an application's data leaks into the memories of the kernel or shared, user-level servers on the host, erasing it after the application terminates becomes difficult or even impossible.

A key contribution of Lacuna is the **ephemeral channel** abstraction, depicted in Figure 2. Ephemeral channels connect the VM to hardware or small bits of software so that only the endpoints see the data from private sessions. The bulk of the kernel and user-level server code does not see this data except possibly in encrypted form. Ephemeral channels facilitate secure erasure after a private session is over because the unencrypted data from the session (1) is confined into a few easy-to-inspect paths, and (2) leaves the system only through a few well-defined endpoints located as close as possible to the hardware.

### 3.1 Usability properties

***Run private and non-private applications concurrently.*** Users can perform sensitive tasks within a private session concurrently with non-private tasks. For example, a user can fill out a medical questionnaire or visit her bank while continuing to poll for new email or listening to music from a cloud service.

***Incur extra costs only for private applications.*** Lacuna is "pay as you go." If the user is not concerned about some application (for example, a computer game)

[4] See src/pulse/stream.c and src/pulse/simple.c in the PulseAudio source.

[5] See src/pulsecore/memblock.c in the PulseAudio source.

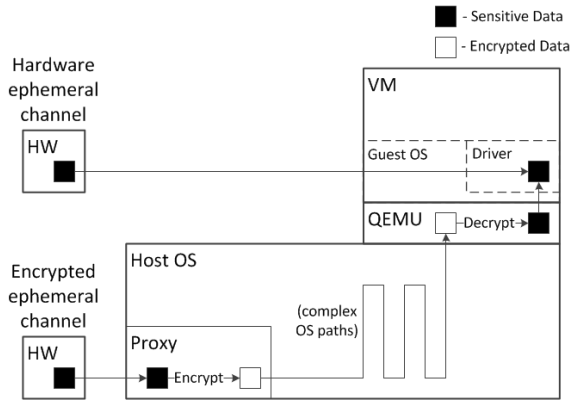



**Figure 2.** Overview of ephemeral channels. Sensitive data flow is shown for both types of channels. Hardware ephemeral channels connect guest system software directly to hardware, while encrypted ephemeral channels connect guest system software to small software proxies on the host or peripheral device. Black boxes represent unencrypted data, white boxes encrypted data.

leaving its data on her computer, the application executes directly on the host OS and Lacuna does not impose a performance overhead.

***Minimize application, VM, and guest OS changes.*** To implement ephemeral channels, Lacuna must change the host OS and the virtual machine manager (VMM), but it supports completely unmodified guest OSes and applications. We were able to run Lacuna with a Microsoft Windows guest and watch streaming video using Internet Explorer. However, in some cases minor modifications to the guest OS yield privacy and performance benefits (e.g., §5.3.3).

***Improve with hardware support but keep legacy compatibility.*** Ephemeral channels benefit from specialized hardware. For example, single root I/O virtualization (SR-IOV) network cards enable hardware ephemeral channels for network packets. However, device support for virtualization is not yet commonplace; SR-IOV is predominantly available in server-class network cards. Lacuna is designed to take advantage of hardware support when it exists, but also works on older systems that lack such support.

***Don't interfere with VM-based security techniques.*** Users can augment the security of an application because of its encapsulation in a VM, and Lacuna will not interfere. For example, a user can wrap a Web browser into a Lacuna VM confined by `iptables` so that it can connect only to the range of IP addresses associated with a particular bank.

***Allow user to revoke protection from certain files.*** For usability, Lacuna lets users save files from a private session into the host system. This revocation of privacy protection requires the user to explicitly identify the file via a trusted dialog box. Such a dialog, which executes under the control and with the privileges of the VMM (not the guest OS), is often called a "powerbox" [45]. Lacuna also supports explicit, user-directed file import from the host into a private session, but hides neither the fact that import took place, nor the imported data.

### 3.2 Privacy properties

Our threat model is similar to the "private mode" in Web browsers, which is familiar to many users and matches their intuitive understanding of what it means for one's computer activities to remain private.

Suppose the user ends a private session at time $T_{user}$, all of its memories are erased by time $T_{clean}$, and the OS reports the process exited at $T_{exit}$ (where $T_{exit} > T_{clean} > T_{user}$). At time $T > T_{exit}$, the computer is seized by a *local attacker* who gains complete control of the entire system, including the OS.

This adversary should not be able to extract any usable evidence of activities conducted in a private session, except (1) the fact that the machine ran a private session at some point in the past (but not which programs were executed during the session), and (2) which devices were used during the session. He should not be able to answer even binary questions ("Did the user watch this video?", "Did she browse that website?", etc.) any better than by random guessing. We refer to this property as **forensic deniability** because it allows the user to plausibly deny any computer activity that she may have engaged in while in a private session.

Forensic deniability must be *coercion-resistant* (this property is sometimes called "rubber-hose resistance"): the user herself should not be able recover any evidence from her private sessions. Lacuna does not persist secrets from one private session to another (e.g., a program in a Lacuna VM cannot save encrypted state to be reused during its next invocation). The attacker controls the host, and if a secret is kept by the user instead—e.g., as a password or in a hardware device—she can be coerced to open the persistent state. To avoid keeping secrets with the user, the contents of the initial VM image are not protected for privacy or integrity.

Lacuna aims to minimize the window from $T_{user}$ (user completes the private session) to $T_{clean}$ (all memories are erased). For example, we rejected any design that requires searching the disk as part of sanitization.

### 3.3 Out-of-scope threats

In keeping with the browsers' "private mode" abstraction, Lacuna is not intended to protect users' privacy against concurrent attackers. If the adversary runs on the host concurrently with a private session (e.g., the host has been compromised by malware *before* the private session terminated), he can observe the user's data in memory and learn everything. We must also assume

that the host operating system is not malicious. A malicious or "pathologically buggy" OS could accidentally persist the contents of memory, expose arbitrary secrets, and not erase them when the private session terminates.

The concept of a "trusted computing base" (TCB) is typically used in contexts where trusted and untrusted components coexist on the same machine. It is not applicable in our threat model, where the attacker gains access to the machine after the private session is over. Before the session terminates, the TCB for Lacuna is the entire system; after $T_{clean}$, the TCB is empty—any software can be malicious. In Section 4.5, we discuss resistance of Lacuna to side-channel attacks.

Lacuna makes its best effort to erase the peripherals' memories, but it cannot prevent them from keeping state that is not erasable via public APIs. For example, Lacuna does not protect against a hypothetical GPU or NIC that logs data in hardware and makes the logs available via an undocumented protocol.

Lacuna does not protect sensitive data stored outside the system. For example, websites may keep evidence of users' visits and reveal it to third parties. An adversary who seizes a router or modem that caches IP addresses or DNS queries may recover traces of network activity even after $T_{clean}$. Note that some local attackers—for example, malware that compromises the machine after the private session is over—do not have access to the state kept in the network. Users concerned about network surveillance can run Tor [53] inside a Lacuna VM.

## 4. Design

This section details the design of Lacuna. In the following, "VMM" refers to Lacuna's virtual machine manager (which is a modified QEMU VMM in our prototype).

### 4.1 Constructing ephemeral channels

A Lacuna VM communicates with peripheral devices via ephemeral channels. Lacuna uses two mechanisms to construct ephemeral channels: encryption and hardware. Table 1 lists all device types supported by our Lacuna prototype and the corresponding channels.

Lacuna takes advantage of recent developments in hardware. Hardware support for efficient virtualization (e.g., nested page tables) allows fast execution of private sessions in a VM, confining most forms of inter-process communications. Lacuna relies on a programmable GPU and obtains great performance benefits from hardware support for encryption (§ 6). Ephemeral channels based on dedicated hardware are only practical with an IOMMU, otherwise a buggy guest kernel could damage the host. Hardware ephemeral channels also benefit from hardware virtualized peripheral devices which are just becoming widely available.

| **Device** | **Endpoints (◁ VMM, ▷ host) of the ephemeral channel** | **HW** |
|---|---|---|
| Display | ◁ Frame buffer<br>▷ CUDA routine on GPU | GPGPU |
| Audio | ◁ Sound card<br>▷ Lacuna software mixer | None |
| Network | ◁ Network card<br>▷ NIC driver | SR-IOV NICs |
| USB input devices | ◁ USB controller<br>▷ USB generic host controller driver | VT-d/ IOMMU |

**Table 1.** Ephemeral channels implemented by Lacuna and the corresponding hardware support, if any.

***Hardware channels.*** A hardware ephemeral channel can use either dedicated hardware, or hardware virtualization support. To assign exclusive control of hardware to the guest kernel, Lacuna uses peripheral component interconnect (PCI) device assignment. Assigned devices are not available to the host, thus host drivers need not be modified. Because the host never handles the data flowing to or from assigned devices (not even in encrypted form), this data leaves no trace in the host. Dedicated hardware sometimes makes sense (e.g., USB controllers), but can be expensive (e.g., multiple network cards), awkward to use (e.g., multiple keyboards), or even impossible (e.g., physical limitations on the number or topology of peripherals).

When available, hardware support for virtualization combines the performance of dedicated hardware with the economy and convenience of dynamic partitioning. For example, a single-root I/O virtualization (SR-IOV) network interface card (NIC) appears to software as multiple NICs, each of which can be directly assigned to a guest. Hardware virtualization is great when available, but is not always an option, thus for some devices—for example, GPUs and audio devices—Lacuna constructs an encrypted channel instead.

***Encrypted channels.*** Encrypted channels use standard key exchange and encryption to establish a trusted channel over an untrusted medium, just like encryption is used to secure network communication. An encrypted channel connects the VMM with a small software proxy for each piece of hardware. Only the VMM process and the proxy handle raw data from the private session, the rest of the system handles only encrypted data. When the VM terminates, the OS zeroes its memory, the proxy zeroes its own memory (if it has one), and the symmetric key that encrypted the data in the channel is deleted. Deleting the key **cryptographically erases** the data, making it unrecoverable [4].

The software proxies are different for each class of devices, but most Lacuna support is relatively device-

independent and can be used for a variety of hardware without the need to port low-level driver changes. For example, Lacuna modifies the generic USB host controller driver to encrypt packets from USB input devices; the hypervisor decrypts the packets just before the virtual keyboard device delivers them to the guest kernel.

One important question is whether we plugged all possible leaks in the software that handles unencrypted data. We believe we did, but that is not the point. The system abstraction of ephemeral channels reduces the auditing burden from impossible to feasible (i.e., about 1,000 lines in our prototype according to Table 3).

### 4.2 Ephemeral channels for specific device types

Our Lacuna prototype provides ephemeral channels for display, audio, USB, and the network.

#### 4.2.1 Display

All accesses by applications to a graphics card in a typical Linux desktop system are controlled by the X server. X processes display requests and sends hardware-specific commands and data to the GPU kernel-mode driver for rendering. Even if a program is running inside a VM, its graphical output is captured by the VMM and rendered as a bitmap in a standard application window on the host.[6] The memory of the host's X server may hold the complete display from the private session, as shown in Section 2.1. The problem of erasing graphical output is thus not confined to a specific driver, but requires in-depth analysis of the code of the X server, which is notorious for its size and complexity.

Lacuna uses an ephemeral channel to remove trust in all user-level servers and kernel-level drivers for display data. The VMM encrypts the virtual frame buffer and sends it directly to GPU memory. GPU memory is exclusively owned by the GPU and is not directly addressable from the CPU. Lacuna thus avoids exposure of the display data to any code running on the CPU such as GPU libraries, the X server, or the host kernel. The VMM then invokes Lacuna's CUDA routine which runs entirely *on the GPU*, decrypts the data in GPU memory, and renders it on the screen via an OpenGL shader. The unencrypted display data is thus present only in VMM memory and Lacuna-controlled GPU memory.

#### 4.2.2 Audio

In Lacuna, audio functionality is split between mixing and everything else (e.g., resampling, equalization, and sound effects). A guest system processes and mixes its own audio, then the VMM virtual sound card encrypts it. To allow multiple VMs to share a single audio device with each other and with non-private processes, the host mixes all of these audio streams. Mixing an unlimited number of audio streams in hardware is not practical and not supported by most sound cards, so Lacuna provides a hardware-agnostic software mixer that runs on the host. The mixer decrypts guest audio just before the final mix is written to the DMA buffer in the host sound card driver. Each VM has an ephemeral channel for audio input and another one for audio output. These channels connect the VMM sound card device with the DMA buffer in the sound card driver.

#### 4.2.3 USB

Lacuna supports a wide variety of USB input devices—including, at a minimum, keyboard and mouse[7]—with ephemeral channels based on either PCI device assignment (a hardware ephemeral channel), or encrypted USB passthrough (an encrypted ephemeral channel).

Many USB input peripherals must communicate with both private and non-private applications, but not at the same time. For example, the user will not be typing into both private and non-private windows simultaneously. Therefore, Lacuna can dynamically switch control of USB devices between the host and the guest.

Using PCI device assignment, Lacuna can assign an entire host USB controller to a VM, thus avoiding any handling of USB data on the host. However, device assignment requires an IOMMU. Furthermore, all devices downstream of the controller (reachable via hubs) are assigned to the same guest, which may be undesirable.

Using encrypted USB passthrough, Lacuna can switch between host drivers and thus let the user toggle the destination of input keystrokes between the private VM and the host. This channel does not require an IOMMU and allows device assignment at a per-port level.

Lacuna minimizes USB-related code modifications by using features common across USB versions and devices. The USB passthrough mode requires no modification to the lowest-level host controller drivers that control specific USB port hardware on motherboards. This mode takes advantage of the output format for USB Human Interface Device (HID) class devices (which include all keyboards and mice) to determine when to return device control to the host, but in general can support any device of this class.

#### 4.2.4 Network

Network support is important for both usability and privacy. Some of the attacks we consider (e.g., malware infecting the host after the private session is over) do not control the network, but can learn private information from IP headers leaked by the VM.

Lacuna creates an ephemeral channel from the host NIC driver to the VMM where it delivers the packet to the virtual network card. This channel can be based on either encryption, or SR-IOV hardware. Encrypted ephemeral channels connect to the host in layer 2: each

[6] Lacuna does not currently support 3D acceleration inside VMs.

[7] Keyboard input can leak via TTY buffers [7].

VM connects to a software tap device, which connects to the NIC via a software bridge. The entire packet, including IP (layer-3) header, is encrypted while it passes through the host. Hardware ephemeral channels based on SR-IOV network cards give a VM direct control over a virtual network PCI device in the card hardware that multiplexes a single network connection.

To minimize the changes to specific device drivers, we encapsulate most routines for MAC registration and encryption/decryption in a generic, device-independent kernel module, `privnet`. This module checks whether a MAC address belongs to some VM and encrypts or decrypts a packet when needed.

### 4.3 Clearing swap

Some users avoid swap. Ubuntu guidelines, however, recommend enabling swap [54] to accommodate memory-hungry programs, support hibernation, prevent program termination in case of unforeseen disaster, and to allow the kernel to manage memory effectively. Lacuna supports swap for greater usability.

Swapped-out memory must be encrypted lest it leaks data from a private session. Existing solutions (dm-crypt in Linux) associate a single, system-wide key with the entire swap. This is unacceptable in our design because when a private session ends, the key used to encrypt this session's swap must be erased. Erasing the key would make any data swapped by a concurrent non-private process undecryptable. There exists a research system [42] that uses multiple rotating keys, but it must swap in any live data upon key rotation, with negative impact on performance.

Lacuna adds metadata so that swap code can recognize pages associated with private sessions. These pages are not shared and only they are encrypted upon swap.

### 4.4 Clearing stack memory

The kernel puts sensitive data in stack-allocated variables that can persist after the function returns [34]. We take advantage of the fact that 64-bit Linux confines a kernel thread's activities to (a) its own kernel stack, and (b) interrupt and exception stacks. When a private VM terminates, Lacuna clears the thread's kernel stack and sends an inter-processor interrupt (IPI) to clear all per-core interrupt and exception stacks.

PaX has a mechanism for zeroing the kernel stack on every return from a system call, but Lacuna does not use this technique because it has a significant performance cost, e.g., a 20% drop in TCP throughput over a loopback connection in one experiment.

### 4.5 Mitigating side channels

In this section, we analyze two classes of side channels, but a comprehensive study of side channels in Linux is well beyond the scope of this paper. Note that a typical side-channel attack assumes that the adversary monitors some aspect of the system concurrently with the protected program's execution. In our threat model, however, the adversary gains access to the system only after the execution terminates. This dramatically reduces the bandwidth of side channels because the adversary observes only a *single value*, as opposed to a sequence of values correlated with the program's execution.

***Statistics.*** Linux keeps various statistics that can potentially compromise forensic deniability. For instance, `/proc/net/dev` keeps the number of bytes transferred by the NIC, while `/proc/interrupts` keeps per-device received interrupt counts. These counters are scattered through kernel code and data structures, making it difficult to design a single mitigation strategy.

Low counts mean that the machine has *not* been used for certain activities. For example, if the number of bytes transferred over the network is low, then the machine has not been used for streaming video. If the number of keyboard and mouse interrupts is low, then the machine has not been used to create a PowerPoint presentation. High counts, on the other hand, may not convey much useful information about activity in a private session because all statistics are aggregates since boot.

***Device metadata.*** Lacuna cannot hide that a particular device was used during a private session, but in-memory data structures that describe device activity can leak additional information. For example, the USB request block contains the length of the USB packet, which may leak the type of the USB device or the type of data transferred (e.g., photos have characteristic sizes).

Lacuna eliminates this side channel by carefully zeroing all metadata fields.

### 4.6 Design alternatives

We survey design alternatives that may appear to—but do not—provide the same guarantees as Lacuna.

***"Just use a virtual machine."*** Running an application in a VM and then erasing the VM's memory when it exits does not provide forensic deniability. As we show in Section 2, programs running in a VM leave traces in the host's data structures, OS swap, and shared user-level servers. Furthermore, saving data from the protected program is essential for usability, but requires a secure dialog (§5.5) that is not a standard feature of VMs.

***"Just use secure deallocation."*** All of our experiments demonstrating recovery of sensitive data after the program terminated were conducted on a Linux system patched with PaX security modifications. One of these modifications is secure deallocation: freed kernel buffers are eagerly scrubbed of their contents. Secure deallocation does not address the problem of sensitive data in shared memory that remains allocated on program exit, including X, PulseAudio, and the kernel.

Additionally, PaX fails to scrub the kernel's numerous memory caches on deallocation, even though this is a known data-lifetime hazard [8, 17]. Ephemeral channels make it easier to implement secure deallocation correctly and comprehensively by limiting the number of memory locations potentially containing unencrypted program data. Rather than eagerly scrubbing freed cache memory, which would harm performance (e.g., over 10% reduction in throughput for our TCP stream to localhost experiment), we manually audit the (few) Lacuna code pathways that require secure deletion to make sure they don't use memory caches. Where memory caches are unavoidable, unencrypted data is either overwritten in place by encrypted data, or (as a last resort) eagerly erased on being freed.

***"Just use hardware."*** Recent research [26, 50] proposed comprehensive virtualization in hardware. These approaches require static partitioning of resources that would be very unattractive for the home user. For example, the number of VMs must be fixed in advance, and a fixed amount of RAM must be dedicated to each VM whether it is used or not. By contrast, Lacuna can run as many concurrent VMs as can be efficiently executed by the underlying hardware (see §6.9 for empirical scalability measurements). Lacuna, too, can take advantage of hardware virtualization where available.

***"Just reboot the machine."*** Rebooting the machine does not guarantee that no traces of application data remain on disk or even in RAM [21]. More importantly, rebooting has an unacceptable impact on usability. For example, few users would be willing to reboot before and after every online banking session.

# 5. Implementation

## 5.1 VMM setup and teardown

Lacuna builds upon the QEMU-KVM hypervisor and a kernel patched with the secure deallocation portion of the PaX patch. Lacuna securely tracks modifications to the initial VM image via an encrypted **diffs file**, which is created when the user starts a private session. To reduce disk I/O, a small amount of image-modification metadata, such as translation tables between sector number and diffs file offset, is kept in VMM memory and never written to the diffs file. The rest of the metadata and all writes to the image are encrypted before they are written to the diffs file. When a session terminates, the key that encrypts the diffs file is deleted and memory containing the VMM address space is zeroed.

In keeping with its threat model, Lacuna does not persist changes to the VM image. Therefore, software updates during a private session (e.g., self-updates to a Web browser) are lost after the session completes.

On teardown, the VMM must erase its image file from the kernel page cache. We add a flag to the `open` system call (`O_PRIVATE`) that tracks all virtual disk images opened by the VMM. On `close`, all private files in the page cache are invalidated and zeroed by PaX.

| Operation | Function |
|---|---|
| init | Sets parameters that describe the cryptographic algorithm to be used (e.g., key size, cipher) |
| set_iv | Sets the initialization vector (IV) for a channel direction |
| send_kex_msg | Sends a key exchange message and receives a response |
| set_activation | Turns a context on or off—this is needed when the use of a device that cannot be multiplexed is toggled between a VM and the host |
| destroy | Zeroes and frees memory associated with a context |
| per_backend | Answers queries specific to a cryptographic context type (e.g., obtains ids for kernel cryptographic contexts) |

**Table 2.** Interface for cryptographic contexts.

## 5.2 Encrypted ephemeral channels

To implement encrypted ephemeral channels, the kernel and programmable devices maintain **cryptographic contexts**, one for each direction of each device's logical communication channel (input from the device or output to the device). Our Lacuna prototype provides kernel and GPU implementations. For symmetric encryption, kernel cryptographic contexts use the Linux kernel's cryptographic routines, while GPU contexts use our own implementation of AES. To establish a shared secret key for each context, Lacuna uses the key exchange portion of TLS 1.1. We ported the relevant parts of the PolarSSL [41] cryptographic library (SHA1, MD5, multi-precision integer support) to run in the kernel.

These contexts are managed from userspace via our `libprivcrypt` library; its interface is shown in Table 2. We modified the QEMU VMM to use `libprivcrypt`. On initialization, the VMM creates cryptographic contexts in the kernel and GPU and establishes shared parameters (algorithm, IV, secret key), allowing it to encrypt data destined to these contexts and decrypt data originating from them. To encrypt and decrypt, `libprivcrypt` uses `libgcrypt` [30] or ported kernel code and Intel's AES-NI hardware encryption support.

When a private session terminates, even abnormally (i.e., from `SIGKILL` or crash), all cryptographic contexts associated with it are zeroed, including those on the GPU. This, along with zeroing of the VMM's memory, ensures that all data that has passed through the ephemeral channels is cryptographically erased.

### 5.3 Ephemeral channels for specific device types

#### 5.3.1 Display

The endpoints of the GPU ephemeral channel are the VMM's frame buffer for an emulated graphics card, which stores the guest's display image as a bitmap, and the GPU. The VMM polls the frame buffer, and, upon each update, encrypts the buffer contents and transfers the encrypted data to GPU memory. Lacuna then invokes its CUDA routine[8] to decrypt the guest's frame buffer in the GPU, maps it onto an OpenGL texture, and renders it on the host's screen with an OpenGL shader. The implementation consists of 10 LOC in the QEMU UI module and SDL library, and an additional QEMU-linked library for rendering encrypted frame buffers, with 691 LOC of CPU code for GPU management and 725 lines of GPU decryption and rendering code.

#### 5.3.2 Audio

Lacuna provides output and input audio channels for each VM and a small (approximately 550 LOC) software mixer that directly interacts with the audio hardware's DMA buffer (§4.2.2). We modified the widely used Intel HD-audio driver to work with the mixer, changing fewer than 50 lines of code. This driver works for both Intel and non-Intel controller chips.[9]

Lacuna can send sound input to multiple VMs. For output (playback), the host kernel keeps a separate buffer for each VM to write raw encrypted audio. Linux's audio drivers provide a callback to update the pointers indicating where the hardware should fetch the samples from or where the application (e.g., PulseAudio) should write the samples. Our mixer takes advantage of this mechanism: upon pointer updates, samples in each encrypted output buffer are decrypted, copied to the DMA buffer between the old and new application pointers, and then zeroed in the encrypted output buffer. The DMA buffer is erased when the VM terminates.

#### 5.3.3 USB

Lacuna's USB passthrough mode encrypts data in USB Report Buffers (URBs) as they are passed to system software from hardware control. Packets destined for the guest and the host may be interspersed, so Lacuna tracks which URBs it should encrypt by associating cryptographic contexts with USB device endpoints. An endpoint is one side of a logical channel between a device and the host controller; communication between a single device and the controller involves multiple endpoints.

We added 118 lines to the `usbcore` driver to encrypt URBs associated with cryptographic contexts as they are returned from hardware-specific host controller drivers. These URBs are decrypted in the VMM's virtual USB host controller before they are passed on to the guest USB subsystem. Our prototype has been tested only with USB 1.1 and 2.0 devices, but should work with USB 3.0. It does not support USB mass storage devices and less common USB device classes (such as USB audio), but adding this support should require a reasonably small effort because our mechanism is largely agnostic to the contents of URBs.

When the user moves her mouse over a private VM's display and presses "Left-Control+Left-Alt", Lacuna engages a user-level USB driver, `devio`, to redirect the keyboard and mouse ports to the VMM.[10] The title bar of the VMM window indicates whether the keyboard and mouse input are redirected through ephemeral channels. When they are not redirected, the Lacuna VMM refuses input to avoid accidental leaks.

The same key combination toggles control of the keyboard and mouse back to the host. The VMM's virtual hardware detects the key combination by understanding the position of modifier key status in data packets common to USB HID devices. With a hardware ephemeral USB channel, detecting the combination requires guest OS modification (119 LOC). With hardware channels, errors that freeze the guest currently leave no way of restoring input to the host, but we believe that this limitation is not intrinsic to our architecture (e.g., the host could run a guest watchdog).

#### 5.3.4 Network

Lacuna VMs are networked in layer 2, enabling encryption of entire layer-3 packets. Each VM is assigned its own MAC address controlled by our `privnet` module, which uses cryptographic contexts to do encryption in Intel's `e1000e` driver with 30 lines of glue code.

Outgoing packets are encrypted by the VMM. The host kernel places them in an `sk_buff`, the Linux network packet data structure. The driver maps each `sk_buff` to a DMA address for the NIC to fetch; right before it tells the NIC to fetch, it queries `privnet` whether the packets in the transmit queue come from a Lacuna VM, and, if so, decrypts them in place. The driver zeroes `sk_buffs` on receipt of a "transmission complete" interrupt. Because decryption takes place right before the packets are written into hardware buffers, packets from a VM cannot be received by the host (and vice versa) at a local address.

For incoming packets, as soon as the driver receives the interrupt informing it that packets are transferred from the NIC to the kernel via DMA, it encrypts the packets destined for the Lacuna VMs. Encryption is

[8] While our implementation uses CUDA and is compatible only with NVIDIA GPUs, similar functionality can be also implemented for AMD GPUs using OpenCL [35].

[9] `http://www.kernel.org/doc/Documentation/sound/alsa/HD-Audio.txt`

[10] The unmodified QEMU already uses this key combination for acquiring exclusive control of the keyboard, but it takes events from the X server and does not provide forensic deniability.

done in place and overwrites the original packets. Decryption takes place in the VMM.

Although the layer-2 (Ethernet) header is not encrypted, its EtherType, an indicator of the layer-3 protocol it is encapsulating, is modified to prevent a checksum failure: a constant is added to it so that the resulting value is not recognized by the Linux kernel during encryption, and subtracted again during decryption. As a side benefit, this bypasses host IP packet processing, improving performance (§6.6).

### 5.4 Encrypted per-process swap

Lacuna adds a new flag, `CLONE_PRIVATE`, to the `clone` system call. When this flag is set, the kernel allocates a private swap context, generates a random key, and protects the swap contents for that kernel thread.

When an anonymous page is evicted from memory, the kernel checks the virtual memory segment metadata (VMA in Linux) to see whether the page is part of a private process. If so, the kernel allocates a scratch page to hold the encrypted data and allocates an entry in a radix tree to track the private swap context. The tree is indexed by the kernel's swap entry so that it can find the context on swap-in. Our implementation re-uses much of the existing swap code path. To help distinguish private pages during normal swap cache clean up, we add an additional bit in the radix tree to indicate when a particular entry may be removed and which entries to purge during process cleanup.

### 5.5 User-controlled revocation of protection

For usability, Lacuna provides a mechanism that allows the user to explicitly revoke protection from a file and save it from a private session to the host, where it may persist beyond the end of the session. This mechanism raises a dialog box ("powerbox") running under the control and with the privileges of the VMM [45]. This dialog enables the user to specify the destination on the host, thus ensuring that all transfers from a private session are explicitly approved by the user.

To implement this mechanism, we made a small modification (74 lines of code) to the Qt framework[11] so that a "Save" dialog box in private VMs presents the user with an additional option to access a file in the host file system. When this button is clicked, Qt makes a hypercall which causes the VMM to open a "File save" dialog that lets the user write the file to the host. Lacuna uses a QEMU virtual serial device to transfer data between private applications and the host.

For importing data into the private session, Lacuna provides command-line programs on the guest and host. The host program writes to a UNIX socket, the VMM reads it and writes into the same virtual serial device, which is read by the guest program. These import utilities are not currently connected to Qt functionality.

[11] `http://qt.nokia.com/`

## 6. Evaluation

We evaluate both the privacy properties and performance of Lacuna. We run all benchmarks except switch latency on a Dell Studio XPS 8100 with a dual-core 3.2 GHz Intel Core i5 CPU, 12 GB of RAM, an NVIDIA GeForce GTX 470, and an Intel Gigabit CT PCI-E NIC, running Ubuntu 10.04 desktop edition. The swap partition is on a 7200 RPM, 250GB hard drive with an 8MB cache. Switch latency to and from the private environment is benchmarked on a Lenovo T510 with a dual-core 2.67 GHz Core i7 CPU and 8GB of RAM, running Ubuntu 12.04 desktop edition. The Lenovo has a Microsoft USB keyboard (vendor/device ID `045e:0730`) and mouse (vendor/device ID `045e:00cb`), as well as an IOMMU, which is required for the PCI assignment-based ephemeral channel. Both machines have AES-NI and use it for all AES encryption except where indicated. Our Lacuna prototype is based on the Linux 3.0.0 host kernel (with a port of the PaX patch's `CONFIG_PAX_MEMORY_SANITIZE` option) and QEMU 0.15.1. The guest VM runs Ubuntu 10.04 desktop edition, with 2 GB RAM and the Linux 3.0.0 kernel to which small modifications were made to support PCI assignment (§5.3.3) and the experiments discussed below.

### 6.1 Validating privacy protection

Following the methodology of [8], we inject 8-byte "tokens" into the display, audio, USB, network, and swap subsystems, then examine physical RAM for these tokens afterwards. Without Lacuna (but with QEMU and PaX), the tokens are present after the applications exit. With Lacuna, no tokens are found after the private session terminates. This experiment is not sufficient to prove forensic deniability, but it demonstrates that Lacuna plugs at least the known leaks.

One subtlety occurred with the video driver. We use the Nouveau open-source driver for the test without the display ephemeral channel and the NVIDIA proprietary driver for the test with the channel, because the NVIDIA driver is required for CUDA execution. To inject tokens, we run a program that displays a static bitmap inside a VM. With the ephemeral channel, no tokens from the bitmap are found after VM termination. Without the channel, we detect the tokens[12] after the VM termination—but not if we use the proprietary driver. This driver does leak data from other applications, but not from QEMU. Without the source code, we are unable to identify the causes for this observed behavior.

[12] The tokens are slightly modified due to the display format conversion in QEMU, which adds a zero after every third byte.

| Subsystem | LOC |
|---|---|
| Graphics | 0 (725 CUDA) |
| Sound | 200 (out), 108 (in) |
| USB | 414 |
| Network | 208 |

**Table 3.** Lines of code (LOC) external to QEMU that handle unencrypted data. Line counts were determined by manual examination of data paths from interrupt handler to encryption using SLOCCount [58].

| | Video | Browser | LibreOffice |
|---|---|---|---|
| QEMU | $32.2 \pm 7.4$ | $25.9 \pm 1.3$ | $8.1 \pm 1.2$ |
| Lacuna | $49.7 \pm 0.3$ ($\Delta$17.5) | $46.2 \pm 1.5$ ($\Delta$20.3) | $21.1 \pm 0.6$ ($\Delta$13.0) |

**Table 4.** CPU utilization (%) for benchmarks with encrypted network, video, and sound channels. The performance of all benchmarks on Lacuna is identical to unmodified QEMU. The increase in CPU utilization is marked with $\Delta$. Averages are calculated over 5 trials with standard deviations as shown.

### 6.2 Measuring data exposure

To estimate the potential exposure of private-session data, Table 3 shows the size of driver code that handles it unencrypted. The graphics data is not exposed at all because it is encrypted by the VMM, which then transfers it directly to the GPU memory and invokes the Lacuna implementation of the CUDA decryption and GL rendering routines on the GPU (implemented in 725 LOC).

### 6.3 Full-system benchmarks

We measure the overhead of Lacuna on a number of full-system tasks: watching a 854 $\times$ 480 video with `mplayer` across the network, browsing the Alexa top 20 websites, and using LibreOffice, a full-featured office suite, to create a document with 2,994 characters and 32 images. We sample CPU utilization at 1 second intervals. To avoid the effects of VM boot and to capture application activity, we omit the first 15 samples and report an average of the remaining samples.

The execution times of the video and LibreOffice benchmarks on Lacuna are within 1% of base QEMU. The performance of the browser benchmark varies due to network conditions, but there is no difference in average execution time. The display—redrawn upon every contents change at the maximum rate of 63 frames/s—is not perceptibly sluggish in any of the benchmarks when using the encrypted GPU channel. Table 4 shows the CPU utilization of the workloads running on Lacuna and on unmodified QEMU.

### 6.4 Clean-up time

The clean-up after a private VM terminates is comprised of five concurrent tasks:

***Clear VM memory.*** Lacuna uses PaX to zero VM memory when the VM process exits and frees its address space. To measure the worst-case window of vulnerability, we run a program in the VM that allocates all 2 GB of available VM memory, then send the VMM a signal to terminate it and measure the time between signal delivery and process exit. Linux does not optimize process exit, often rescheduling a process during its death. In 10 trials, unmodified Linux required $2.1 \pm 0.1$ s to terminate a VM. The worst case we measured for Lacuna (USB passthrough mode with keyboard and mouse) is $2.5 \pm 0.2$ s.

***Clear buffered disk image.*** The Lacuna VMM opens disk image files with a privacy flag so that the kernel can securely deallocate all buffer cache pages for those files when the VMM exits without affecting the page cache contents for concurrent, non-private programs. Only clean pages need to be deallocated and zeroed because a private Lacuna session does not persist the modified disk image. This operation takes $0.111 \pm 0.002$ s in our video benchmark.

***Clear swap cache memory.*** Lacuna securely deallocates freed swap cache pages. A benchmark program allocates 12 GB of memory to force the system to swap, writing out an average of $677.8 \pm 33.4$ MB to the swap partition. However, because the swap cache is used only for transient pages (those that have not completely swapped out or swapped in), the average number of memory pages remaining in the swap cache at program termination is only 50 or so (200KB). Clearing this data takes only $68.9 \pm 44.6$ $\mu$s.

***Clear kernel stacks.*** Lacuna zeroes the VMM's kernel stack, and also notifies and waits for each CPU to zero their interrupt and exception stacks. In our video benchmark, this takes $15.8 \pm 1.15$ $\mu$s.

***Clear GPU memory.*** Lacuna has a GPU memory scrubber which uses the CUDA API to allocate all available GPU memory and overwrites it with zeros. A similar GPU memory scrubbing technique is used in NCSA clusters.[13] Our scrubber zeroed 1.5GB of GPU memory in $0.170 \pm 0.005$ s.

### 6.5 Switch time

Table 5 shows how long it takes to switch into a private session and how the switch time depends on the number of devices and type of the ephemeral channel(s).

A significant portion of the switch time when using encrypted USB passthrough results from disabling the

[13] http://www.ncsa.illinois.edu/AboutUs/Directorates/ISL/software.html

| Channel type | Switch time (s) |
|---|---|
| **USB passthrough** | |
| keyboard only | $1.4 \pm 0.2$ |
| keyboard + mouse | $2.3 \pm 0.2$ |
| **PCI assignment** | |
| keyboard only | $2.4 \pm 0.2$ |
| keyboard + mouse | $3.8 \pm 0.2$ |

**Table 5.** Switch time for different numbers of peripherals and ephemeral channel types (averages over 5 trials).

peripheral USB drivers ($0.8 \pm 0.1$ s for keyboard alone, $1.0 \pm 0.2$ s for keyboard and mouse) to allow `devio` to take control. This time is affected by the number of USB devices that must be disconnected. Interestingly, it is also affected by the complexity of the USB device: keyboards with media keys often show up as two devices on the same interface, which necessitates disconnecting two instances of the peripheral driver.

We noticed an interaction between the guest USB drivers and QEMU that significantly affects switch time. Linux's USB drivers perform two device resets during device initialization. These resets in the guest are particularly costly because each results in QEMU performing an unnecessary (since QEMU has already performed a reset) unbinding of the `devio` driver and the reattachment of the device's initial `usbhid` driver. Eliminating QEMU's action upon these resets cuts this component of switch time by two thirds.

### 6.6 Network performance

We benchmark network performance between a private VM and a gateway connected by a switch: netperf and ping results are in Table 8, scp and netcat in Table 6.

There are several types of netperf tests. TCP_STREAM uses bulk transfer to measure throughput, the other types measure latency. TCP_RR (Request/Response) tests the TCP request/respond rate, not including connection establishment. TCP_CC (Connect/Close) measures how fast the pair of systems can open and close a connection. TCP_CRR (Connect/Request/Response) combines a connection with a request/response transaction. Ping measures round-trip time.

| File size | Transfer time (s) | | |
|---|---|---|---|
| | scp | Ephemeral + netcat | |
| | | AES-NI | Software |
| 400MB | 8.41 | 4.28 | 8.92 |
| 800MB | 14.96 | 8.55 | 17.50 |

**Table 6.** netcat and scp test results.

Neither latency, nor throughput is significantly affected when using AES-NI, except for a dip in throughput for receiving 300 byte packets. For small packets, performance with AES-NI encryption is slightly better than without encryption because encrypted packets bypass some host processing (since they appear to be of an unknown packet type). To verify this explanation, we did an additional experiment where we changed the EtherType of each packet without encrypting the content. We measured over 120Mbps throughput when sending 30-byte packets, which is about a 40% improvement. Software encryption achieves roughly half the throughput of AES-NI.

| | No encryption | AES-NI | PCI assignment |
|---|---|---|---|
| CPU util (%) | 27.7±2.7 | 36.0±1.6 | 14.7±4.2 |

**Table 7.** CPU utilization for tap networking without encryption, with encryption, and using PCI assignment when transferring an 800MB file via netcat. The throughput is 794±3 Mbps for all runs.

We also compare the file transfer time for netcat using an encrypted ephemeral channel and scp without using ephemeral channels (Table 6). File transfer with AES-NI encryption is twice as fast as software-only scp. These results also validate that our software encryption performance is comparable to `scp`.

Table 7 shows the measurements of CPU utilization when transferring an 800MB file using no encryption, AES-NI, and PCI assignment. This benchmark was run on a quad-core 3.6 GHz Dell OptiPlex 980 with 8 GB of RAM and an Intel Gigabit ET NIC.

While all methods have nearly identical throughput, PCI assignment significantly lowers CPU utilization.

### 6.7 Audio latency

To measure output latency from the VM to the sound DMA buffer, we sent a known sequence through the sound channel and measured host timestamps for send and receive. The results are in Table 9, showing that the latency of the encrypted ephemeral audio channel is smaller than that of PulseAudio.

| | Latency (ms) |
|---|---|
| Ephemeral channel | $23.5 \pm 8.6$ |
| PulseAudio | $57.5 \pm 11.3$ |

**Table 9.** Audio latency comparison (averages over 10 trials).

There are counterbalancing effects at play here. The encrypted channel incurs additional computational overhead, but bypasses PulseAudio mixing and shortens the path from the VM to host audio DMA buffer.

### 6.8 Swap performance

Figure 3 compares the performance of plain Linux, Lacuna without encrypted swap, Lacuna with encrypted

| Test type | Netperf throughput (Mbps) | | | | | | Netperf latency$^{-1}$ (Trans./s) | | | Ping round-trip time (ms) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TCP_STREAM send | | | TCP_STREAM recv | | | TCP_RR | TCP_CC | TCP_CRR | | | |
| Packet size | 1400 | 300 | 30 | 1400 | 300 | 30 | 1 | 1 | 1 | 1400 | 300 | 30 |
| QEMU | 788 | 516 | 86 | 827 | 829 | 226 | 5452 | 2530 | 2260 | 0.327 | 0.251 | 0.237 |
| Lacuna | 769 | 419 | 89 | 819 | 820 | 231 | 5312 | 2487 | 2180 | 0.366 | 0.253 | 0.219 |
| HW encryption | 2% | 19% | -4% | 1% | 1% | -2% | 3% | 2% | 4% | 12% | 1% | -8% |
| Lacuna | 373 | 242 | 54 | 373 | 370 | 168 | 5206 | 2264 | 2029 | 0.408 | 0.277 | 0.244 |
| SW encryption | 53% | 53% | 37% | 55% | 55% | 26% | 5% | 11% | 10% | 25% | 10% | 0.3% |

**Table 8.** Netperf and ping test results for unmodified QEMU and Lacuna with hardware-assisted (HW) AES-NI encryption and software (SW) encryption. Reductions in performance are shown as percentages, where negative values indicate better performance than QEMU.

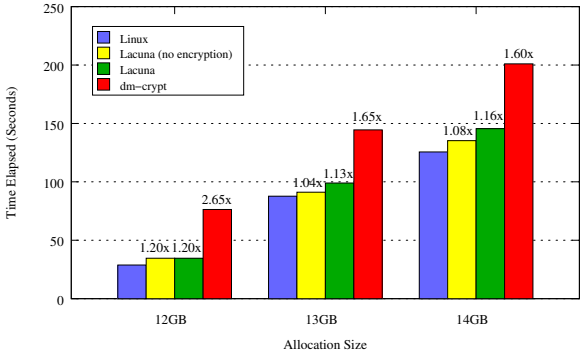



**Figure 3.** Average elapsed time for swap microbenchmarks (lower is better). This benchmark allocates a buffer using malloc, touches each page in pseudorandom order, and reads the pages of the buffer in order to check correctness. The numbers above the bars indicate relative slowdown relative to Linux.

swap, and `dm-crypt`-protected swap. In the first three cases, a non-private process performs similarly to Linux. Our encrypted swap differs from standard swap in two ways whose effects are shown in the graph: it allocates a scratch page and bookkeeping for every private page swapped and encrypts the swapped-out pages.

`dm-crypt` has particularly bad performance in this microbenchmark. We verified that our installation of `dm-crypt` on ext4 adds, on average, 5% overhead when running file-system benchmarks such as IOzone[14]

### 6.9 Scalability

Table 10 shows the performance of multiple concurrent Lacuna VMs, all executing the LibreOffice workload in a private session. The performance overhead of one VM is negligible, but increases with eight concurrent VMs because the CPU is overcommitted. Our attempt to run more than eight VMs produced an unexplained CUDA error. Non-private VMs scale to 24 instances before Linux's out-of-memory killer starts killing them.

[14] `http://www.iozone.org/`.

| Setup | Running Time (s) |
|---|---|
| 1 QEMU VM | $189.3 \pm 0.1$ |
| 1 Lacuna VM | $190.6 \pm 0.1$ $(1.01\times)$ |
| 8 QEMU VMs | $191.6 \pm 0.1$ |
| 8 Lacuna VMs | $277.3 \pm 1.1$ $(1.45\times)$ |

**Table 10.** Time to complete the LibreOffice workload under contention from other VMs (averages over 5 trials).

## 7. Related work

***Lifetime of sensitive data.*** Copies of sensitive data can remain in memory buffers, file storage, database systems, crash reports, etc. long after they are no longer needed by the application [6, 7, 17, 46, 56], or leak through accidentally disclosed kernel memory [22, 34]. To reduce the lifetime of sensitive data, Chow et al. proposed *secure deallocation* of memory buffers [8]. In Section 2, we demonstrate that secure deallocation alone does not achieve forensic deniability. Chow et al. focus on reducing average data lifetime, whereas forensic deniability requires minimizing worst-case data lifetime. A recent position paper [24] identifies the problem of worst-case data lifetime and suggests using information flow and replay to solve it.

CleanOS [52] helps mobile applications protect their secrets from future compromise by encrypting sensitive data on the phone when the application is idle. It does not prevent leaks through the OS and I/O channels.

***Red/green systems.*** Lampson [27] discusses the idea of two separate systems, only one of which ever sees sensitive data (that one is red, the other green). Several systems switch between "secure" and regular modes [5, 32, 47, 55]. They do not provide forensic deniability for the red system and often require all activity on the green system to cease when the red one is active. Pausing the green system can disrupt network connections, e.g., to a cloud music service. Lacuna supports concurrent, finely interleaved private and non-private activities.

***Isolation.*** Xoar [9] and Qubes [43] break up the Xen control VM into security domains to minimize its attack surface and enforce the principle of least privilege; Qubes also facilitates partitioning of user applications. These systems provide an implementation of an inferior VM [40] (aka disposable VM) that isolates untrusted programs in a fast-booting,[15] unprivileged, copy-on-write domain. Although not designed for minimizing data lifetime per se, these systems could be Lacuna's underlying virtualization mechanism instead of QEMU. Lacuna's ephemeral channels can support private sessions regardless whether the underlying hypervisor is monolithic or compartmentalized.

Tahoma [11] and the Illinois Browser OS [51] increase the security of Web applications using a combination of hypervisors and OS abstractions. They do not limit data lifetime within the host system.

Systems with multi-level security (MLS) and, in general, mandatory access control (MAC) can control information flow to prevent information from disclosure. Some MAC systems separate trusted and untrusted keyboard input [25] as Lacuna does. We are not aware of any MAC, MLS, or more modern (e.g., [24, 31, 59]) system that provides deniability against an attacker who compromises the system after a private session is over.

***Encrypted file systems.*** Boneh and Lipton observed that data can be "cryptographically erased" by encrypting it first and then erasing the key [4]. Many cryptographic file systems use encryption to (1) protect the data after the computer has been compromised, and/or (2) delete the data by erasing the key [3, 15, 38, 39, 60]. Recently, encrypted file systems have been proposed for secure deletion of flash memory [28, 29, 44]. Encrypted file systems that derive encryption keys from user passwords are not coercion-resistant. ZIA relies on a hardware token to provide the decryption key when the token is in physical proximity to the machine [10].

In contrast to full-disk encryption, filesystem-level encryption does not provide forensic deniability. For example, the current implementation of the encrypted file system in ChromeOS on a Cr-48 laptop is based on eCryptfs [14] which reveals sizes of individual objects, allowing easy identification of many visited websites in the encrypted browser cache using standard fingerprinting techniques based on HTML object sizes [13, 48].

Provos observed that application data stored in memory may leak out via OS swap and proposed encrypting memory pages when they are swapped out [42]. We use a similar idea in our implementation of encrypted swap.

***Steganographic and deniable file systems.*** Steganographic and deniable file systems aim to hide the existence of certain files [18, 33, 36]. This is a stronger privacy property than forensic deniability. Czeskis et al. showed that the OS and applications can unintentionally reveal the existence of hidden files [12]. Deniable file systems can be used in combination with our system for stronger privacy protection.

***Data remanence.*** There has been much work on data remanence in RAM, magnetic, and solid-state memory [19–21], as well as secure deletion techniques focusing on flash memory [28, 29, 44, 49, 57]. The latter are complementary to our approach.

***Digital rights management (DRM).*** The goal of DRM is to restrict users' control over digital content. Some DRM systems encrypt application data which may reduce its lifetime, but any resulting deniability is incidental. For example, high-bandwidth digital content protection (HDCP) is a cryptographic protocol that prevents content from being displayed on unauthorized devices, but the content is still exposed to the X server and GPU device drivers. DRM is controversial [16], and we believe that solutions for protecting user privacy should not be based on proprietary DRM technologies.

## 8. Conclusion

We presented Lacuna, a system that makes it possible to erase memories of programs' execution from the host. Lacuna runs programs in a special VM and provides "ephemeral channels" through which they can securely communicate with display, audio, and USB input devices, with only 20 percentage points of CPU overhead. Ephemeral channels limit the number of outlets through which program data can leak into the host, prevent unwanted copying of the data, and allow easy erasure. The abstraction presented to the user is a "private session," akin to the "private mode" in modern Web browsers albeit with much stronger privacy guarantees.

**Acknowledgments.** We thank Owen Hofmann for his clever and witty comments. This research was partially supported by the NSF grants CNS-0746888, CNS-0905602, CNS-1017785, a Google research award, the MURI program under AFOSR Grant No. FA9550-08-1-0352, the Andrew and Erna Fince Viterbi Fellowship, and grant R01 LM011028-01 from the National Library of Medicine, National Institutes of Health.

## References

[1] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security*, 2010.

[2] S. Bauer and N. Priyantha. Secure data deletion for Linux file systems. In *USENIX Security*, 2001.

[3] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1994.

[4] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security*, 1996.

[5] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security*, 2009.

[15] 4-5 seconds, per `http://theinvisiblethings.blogspot.com/2010/10/qubes-alpha-3.html`

[6] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *USENIX Security*, 2003.
[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.
[8] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.
[9] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *SOSP*, 2011.
[10] M. Corner and B. Noble. Zero-interaction authentication. In *MOBICOM*, 2004.
[11] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for Web applications. In *S&P*, 2006.
[12] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *HotSec*, 2008.
[13] G. Danezis. Traffic analysis of the HTTP protocol over TLS. `http://research.microsoft.com/en-us/um/people/gdane/papers/TLSanon.pdf`, 2010.
[14] eCryptfs. `https://launchpad.net/ecryptfs`.
[15] The encrypting file system. `http://technet.microsoft.com/en-us/library/cc700811.aspx`.
[16] E. Felten. USACM policy statement on DRM. `https://freedom-to-tinker.com/blog/felten/usacm-policy-statement-drm/`.
[17] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *ACM SIGOPS European Workshop*, 2004.
[18] P. Gasti, G. Ateniese, and M. Blanton. Deniable cloud storage: Sharing files via public-key deniability. In *WPES*, 2010.
[19] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *USENIX Security*, 1996.
[20] P. Gutmann. Data remanence in semiconductor devices. In *USENIX Security*, 2001.
[21] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security*, 2008.
[22] K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosure attacks. In *DSN*, 2007.
[23] N. Joukov, H. Papaxenopoulos, and E. Zadok. Secure deletion myths, issues, and solutions. In *ACM Workshop on Storage Security and Survivability*, 2006.
[24] J. Kannan, G. Altekar, P. Maniatis, and B.-G. Chun. Making programs forget: Enforcing lifetime for sensitive data. In *HotOS*, 2011.
[25] P. A. Karger, M. E. Zurko, D. W. Benin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *S&P*, 1990.
[26] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In *ISCA*, 2010.
[27] B. Lampson. Usable security: How to get it. *Communications of the ACM*, 52(11), Nov. 2009.
[28] B. Lee, K. Son, D. Won, and S. Kim. Secure data deletion for USB flash memory. *Journal of Information Science and Engineering*, 2011.
[29] J. Lee, S. Yi, J. Heo, H. Park, S. Y. Shin, and Y. Cho. An efficient secure deletion scheme for flash file systems. *Journal of Information Science and Engineering*, 2010.
[30] The libgcrypt reference manual. `http://www.gnupg.org/documentation/manuals/gcrypt/`.
[31] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do you know where your data are? Secure data capsules for deployable data protection. In *HotOS*, 2011.
[32] M. Mannan, B. H. Kim, A. Ganjali, and D. Lie. Unicorn: Two-factor attestation for data security. In *CCS*, 2011.
[33] A. McDonald and M. Kuhn. StegFS: A steganographic file system for Linux. In *IH*, 1999.
[34] J. Oberheide and D. Rosenberg. Stackjacking your way to grsecurity/PaX bypass. `http://jon.oberheide.org/files/stackjacking-hes11.pdf`, 2011.
[35] OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.
[36] H. Pang, K.-L. Tan, and X. Zhou. StegFS: A steganographic file system. In *ICDE*, 2003.
[37] Homepage of the PaX team. `http://pax.grsecurity.net`.
[38] R. Perlman. The Ephemerizer: Making data disappear. `http://www.filibeto.org/~aduritz/truetrue/smli_tr-2005-140.pdf`, 2005.
[39] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin. Secure deletion for a versioning file system. In *FAST*, 2005.
[40] M. Piotrowski and A. D. Joseph. Virtics: A system for privilege separation of legacy desktop applications. Technical Report UCB/EECS-2010-70, University of California, Berkeley, 2010.
[41] PolarSSL library - Crypto and SSL made easy. `http://www.polarssl.com`.
[42] N. Provos. Encrypting virtual memory. In *USENIX Security*, 2000.
[43] Qubes. `http://qubes-os.org/`.
[44] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security*, 2012.
[45] M. Seaborn. Plash: Tools for practical least privilege. `http://plast.beasts.org`, 2008.
[46] P. Stahlberg, G. Miklau, and B. Levine. Threats to privacy in the forensic analysis of database systems. In *SIGMOD*, 2007.
[47] K. Sun, J. Wang, F. Zhang, and A. Stavrou. SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In *NDSS*, 2012.
[48] Q. Sun, D. Simon, Y.-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted Web browsing traffic. In *S&P*, 2002.
[49] S. Swanson and M. Wei. SAFE: Fast, verifiable sanitization for SSDs. Technical Report cs2011-0963, UCSD, 2010.
[50] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *CCS*, 2011.
[51] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.
[52] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting mobile data exposure with idle eviction. In *OSDI*, 2012.
[53] Tor. `http://www.torproject.org`.
[54] SwapFaq. `https://help.ubuntu.com/community/SwapFaq`. Retrieved on 5/3/12.
[55] A. Vasudevan, B. Parno, N. Qu, and A. Perrig. Lockdown: A safe and practical environment for security applications. Technical Report CMU-CyLab-09-011, CMU, 2009.
[56] J. Viega. Protecting sensitive data in memory. `http://www.ibm.com/developerworks/library/s-data.html?n-s-311`, 2001.
[57] M. Wei, L. Grupp, F. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, 2011.
[58] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2001.
[59] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *NSDI*, 2007.
[60] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.

# CleanOS: Limiting Mobile Data Exposure with Idle Eviction

Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, Nikhil Sarda
*Columbia University*

## Abstract

Mobile-device theft and loss have reached gigantic proportions. Despite these threats, today's mobile devices are saturated with sensitive information due to operating systems that never securely erase data and applications that hoard it on the vulnerable device for performance or convenience. This paper presents CleanOS, a new Android-based operating system that manages sensitive data rigorously and maintains a clean environment at all times. To do so, CleanOS leverages a key property of today's mobile applications – the use of trusted, cloud-based services. Specifically, CleanOS identifies and tracks sensitive data in RAM and on stable storage, encrypts it with a key, and evicts that key to the cloud when the data is not in active use on the device. We call this process *idle eviction* of sensitive data. To implement CleanOS, we used the TaintDroid mobile taint-tracking system to identify sensitive data locations and instrumented Android's Dalvik interpreter to securely evict that data after a specified period of non-use. Our experimental results show that CleanOS limits sensitive-data exposure drastically while incurring acceptable overheads on mobile networks.

## 1 Introduction

Mobile technology is replacing desktops as the primary personal computing platform and is being used in increasingly sensitive contexts. For example, today's users rely on smartphones and tablets to access their personal and corporate email, prepare tax returns, and review customer documents [32]. Even the US military recently announced that it will equip soldiers with Android devices for accessing classified documents [28]. The draw to new mobile technology is justifiable: mobile devices offer convenient and constant connectivity, increase productivity, and provide access to feature-rich, cloud-based applications (a.k.a. "apps").

Despite these advantages, the transition to mobile devices raises serious and yet unresolved concerns, particularly with respect to data security in the event of device theft and loss. Unlike desktops, generally assumed to be physically secure, mobile devices are extremely prone to theft and loss. Statistics here are staggering: 49% of the New York City population has experienced mobile phone loss/theft [24], and the FCC recently declared mobile theft an "epidemic" in major US cities [13].

Though alarming, these statistics have yet to prompt mobile OSes to address the serious data-security threats posed by device theft or loss. Like their desktop precursors, such as Linux and Mac OS X, mobile OSes let sensitive data accumulate uncontrollably on the device. For example, the OS accumulates significant amounts of data in cleartext memory, and the file system retains deleted files by not purging their contents. Despite being backed by clouds, applications hoard sensitive data – such as emails, documents, and banking information – on the vulnerable device. Although encrypted file systems [26], encrypted RAM [34], and remote-wipeout systems [3, 21] help protect this data, they are imperfect stopgaps for OSes that were simply not designed with physical insecurity in mind. For example, a recent study shows that 57% of corporate users employ no locking mechanisms on their smartphones, rendering encryption useless [32].

This paper presents *CleanOS*, a new Android-based mobile operating system[1] designed to *manage sensitive data rigorously and maintain a clean environment at all times in anticipation of device theft*. The crucial insight in CleanOS is to leverage the tight integration of today's mobile applications with trusted cloud-based services in order to evict sensitive in-memory and on-disk data to those services whenever it is not needed on the device. CleanOS thus ensures that the minimal amount of sensitive data is exposed on the vulnerable device at any time.

CleanOS extends Android in two major ways. First, it introduces *sensitive data objects* (SDOs), a new abstraction that facilitates management of sensitive data on mobile devices. An SDO is a logical collection of Java objects, files, and database items that applications create and use to manage their sensitive data, such as emails, financial data, or documents. SDOs and their data "disappear" from the device unless they are frequently used by an application. For example, if an email app adds an email's content to an SDO, any "trace" of that content automatically disappears from RAM and stable storage unless the user is actively reading that email on an unlocked screen. Recovering the email requires interaction with the cloud.

Second, to evict idle SDOs, CleanOS modifies Android's Java interpreter (Dalvik) to introduce a new type of Java garbage collector (GC), called an *evict-idle GC* (eiGC). While a traditional GC deallocates only those objects guaranteed to never be used in the future (i.e., no pointers to them exist), eiGC eliminates objects that have not been used for a period of time *even if* they might be used again in the future (i.e., pointers to them still exist). To do so, eiGC walks through all Java objects in an idle SDO and encrypts their data-bearing fields, such as primitives and arrays of primitives, with a key that is escrowed in the cloud. Our modified Dalvik interpreter then faults when a bytecode instruction executes on an evicted

[1] We view the OS notion broadly in this paper to include both the traditional OS and the entire Android framework on which apps run.

| Component | New or Changed Features |
|---|---|
| Dalvik (JVM) | Evict-idle Garbage Collector (eiGC)<br>Eviction-aware bytecode interpretation<br>Secure deallocation of interpreted stacks |
| Android SDK | SDO API<br>Default SDO heuristics |
| TaintDroid | Support for millions of taints<br>SQLite vulnerability fix |
| SQLite | Taint tracking in database |
| Webkit | Screen-buffer purging |
| Bionic (libc) | Secure user-space deallocation |
| Linux Kernel | Secure page deallocation with grsecurity |

Figure 1: **CleanOS Modifications to Android, TaintDroid.**

object, retrieves the key from the cloud, and decrypts the object. Thus, data eviction in CleanOS is logical; the data itself remains on the device in encrypted form, while the key is shipped to the cloud.

The major security benefit of CleanOS stems from the value-added services that app clouds can build on top of it. For example, a cloud could revoke data access following a theft report, provide an audit log of data exposed upon theft, or monitor data access to detect anomalous uses. Building such services on today's "dirty" devices would be tremendously challenging and likely require sacrificing semantics or performance. For example, Gmail allows email access revocation [18], but emails cached on the device remain exposed. Conversely, not caching sensitive data on the device degrades performance over slow mobile networks. CleanOS provides device-side OS support for building robust, secure, and efficient value-added cloud services.

We built CleanOS in Android using the TaintDroid taint-tracking system [12] and also implemented a value-added cloud service that provides post-theft data-exposure auditing. To do so, we modified several core components in Android and TaintDroid, summarized in Figure 1. Together, our changes provide: (1) eviction of idle Java objects, (2) heuristics for identifying sensitive data without requiring app changes, (3) support for millions of taints in TaintDroid, and (4) multi-layer secure deallocation of freed data in Java, native, and kernel space. While CleanOS' design extends in-memory eviction to stable storage, this paper and our current prototype focus on in-memory data eviction.

Overall, we make the following contributions:

1. We demonstrate the sensitive data exposure problem by analyzing 14 popular Android apps (§2).
2. We define SDOs, a new abstraction for managing sensitive data on theft-prone devices (§3).
3. We implement CleanOS, an Android OS extension that combines known encryption-based data destruction [4, 16, 30] with a new GC process that evicts idle sensitive data (§4 and §5).
4. We present a set of valuable add-on services that clouds could build on top of CleanOS (§6).

## 2 Case Study: Data Exposure on Android

We selected for analysis 14 Android apps according to their popularity in five sensitive categories: email, finance, document editing, password management, and social networking. We define as *exposed* any data that persists on the device – either in RAM or on storage – for a prolonged period of time, such as 10 minutes (§3 describes our rationale for this threat model). Our goal in the analysis was to answer three questions: (1) Is sensitive-data exposure a real problem? (2) If so, what are its causes? and (3) Is the exposure necessary? We tackle each question using examples from our analysis.

**Is Data Exposure a Real Problem?** We installed the 14 apps on a rooted Nexus S phone with Android 2.3.4 and asked the following question: what kinds of sensitive data can one find by dumping RAM and database contents while apps run in the background? Our acquisition process was vastly simplified by our rooted phone and the lack of encryption on the default Android configuration. Nevertheless, we believe that our findings indicate the level of data exposure on better-protected phones in face of realistic, albeit sophisticated, attacks, such as cold boot RAM imaging [19]. We created a stable-state environment – akin to the one a thief might find on a lost device – by ensuring that apps had not been used for 10 minutes prior to taking RAM and DB dumps.

The answer to our question is eye-opening: with simple techniques, we retrieved cleartext copies of sensitive information from *all but one app*. Figure 2(a) shows examples of cleartext sensitive data we extracted from a select subset of the apps. Figure 2(b), column *"Extracted Cleartext Data,"* expands the result set to all 14 apps and categorizes data in three classes of varied sensitivity: passwords, contents (e.g., email body, document content, bank account), and metadata (e.g., email subject, document title). Overall, we captured passwords in 5/14 apps, contents in 11/14 apps, and metadata in 13/14 apps.

**What Causes Data Exposure?** Given these results, an obvious question is what leads to so much leakage. There are several possible answers:

*Insecure Deletion:* The Android OS, including the kernel, system libraries, and the Java framework, leaks sensitive information by not erasing data securely after it is deallocated or by not securely erasing files when an app asks it to do so. These problems are well known in desktop and server settings and have been addressed with secure deallocation [6] and assured deletion [30, 39], respectively.

*OS Data Buffering:* Recent work shows that OSes and device drivers retain data in buffers past its intended life. It also shows how to limit OS-buffered data exposure [10].

*App Data Hoarding:* Although most of the apps are cloud-based, our experiments show that they hoard significant amounts of cleartext sensitive information on the device, either in RAM or in the local database. For exam-

| App | Extracted Cleartext Data |
|---|---|
| Email | password, email contents, subjects, from/to, contacts |
| OI Notepad (doc) | document and metadata |
| KeePass (password mgr) | app password, all stored passwords & descriptions |
| Pageonce (finance) | password, transactions, bank account information |
| Facebook (social) | wall posts and messages |

**(a) Examples of data extracted from RAM / DB.**

| App | Data | When App Uses Data |
|---|---|---|
| Email | password | user/automatic refresh |
| | subjects | on the email list screen |
| | contents | user opens the email |
| OI Notepad | note title | on the note list screen |
| | note body | user edits the note |
| KeePass | master password | app launches |
| | entry name | on the entry list screen |
| | entry password | user opens the entry |

**(c) Example usage of hoarded data by apps.**

| App | Description | Extracted Cleartext Data | | | Cleartext Data Hoarding | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | RAM | | | SQLite DB | | |
| | | Pass-word | Cont-ents | Meta-data | Pass-word | Cont-ents | Meta-data | Pass-word | Cont-ents | Meta-data |
| Email | email (default) | Y | Y | Y | Y | | Y | Y | Y | Y |
| GMail | email | | Y | Y | | | | | Y | Y |
| Y! Mail | email | Y | Y | Y | | | | | Y | Y |
| GDocs | documents | | | Y | | | Y | | Y | Y |
| OI Notepad | documents | | Y | Y | | Y | Y | | | |
| DropBox | documents | | | Y | | | Y | | | Y |
| KeePass | password mgr | Y | Y | Y | Y | Y | Y | | | |
| Keeper | password mgr | Y | Y | Y | Y | | Y | | | |
| Amazon | commerce | | | | | | | | | |
| Pageonce | finance | Y | Y | Y | Y | Y | Y | | | Y |
| Mint | finance | | Y | Y | | Y | Y | | Y | Y |
| Google+ | social | | Y | Y | | | | | Y | Y |
| Facebook | social | | Y | Y | | | | | Y | Y |
| LinkedIn | social | | Y | Y | | | Y | | Y | Y |

**(b) Exposure of cleartext sensitive data across all 14 apps.**

Figure 2: **Sensitive Data Exposure.** (a) Examples of captured sensitive data. (b) A 'Y' indicates that we obtained cleartext copies from RAM/DB. A blank cell does not mean that the data is not on the device, but just that we did not find it in cleartext; the data could exist in some encrypted form. (c) Examples of when hoarded sensitive data is being actually used by the apps.

ple, the default Android email app maintains the email account password in cleartext RAM *at all times*, while KeePass, a popular password manager, loads its entire password database into RAM at startup and keeps it there. Column *"Cleartext Data Hoarding"* in Figure 2(b) shows the *persistent*, app-intended cleartext data we found in RAM or DBs.[2] It demonstrates that the hoarding behavior is pervasive: all but one of the 14 apps permanently maintain at least one type of sensitive data either in RAM or in the database, while 6/14 apps permanently maintain their passwords or some sensitive content in RAM.

*Memory Leaks:* Beyond the scope of our experiments is the well-known ease of unwittingly introducing memory leaks into Android applications [2]. If small, these leaks may go undetected and expose sensitive information.

**Is Data Exposure Necessary?** Although apps hoard significant amounts of sensitive data on mobile devices, they tend to access this data fairly infrequently, suggesting that data is often exposed longer than it needs to be. By way of example, Figure 2(c) identifies situations where three of our most problematic apps use hoarded sensitive data. For example, the password in the default Android Email app, which we know is exposed in RAM at all times, is in fact used only during inbox refreshes (the default is every 15 minutes). Similarly, each email's content is exposed in SQLite at all times but accessed only when the user opens that particular email. While the frequency of these operations depends on the workload, intuitively it should be relatively rare, making prolonged exposure unnecessary.

**Implications for Mobile OS Design.** Secure deletion for storage, RAM, and OS buffers has been acknowledged as, and developed into, a primary OS function [6, 30, 10]; however, the management of app-driven data hoarding or leakage has thus far been considered an app's own responsibility. For example, faced with similar data-hoarding practices in desktop and server applications, Chow, et al. [6] conclude that "little can be done without modifying the application" and that "leaks are recognized as bugs by application programmers, so they are actively sought after and fixed." Unfortunately, relying on the app to manage sensitive data is problematic. Sensitive-data caching presents tradeoffs between security on one hand and performance, usability, and energy/bandwidth consumption on the other hand. Without solid abstractions, calibrating these tradeoffs is challenging. For example, should a document-editing app cache the documents locally for good performance over cellular networks (as recommended in some mobile app guidelines [14]), or should it not do so for security reasons (as recommended in other guidelines [40])? Should it cache the user's password for convenience, or should it prompt the user for it whenever it is needed?

We argue that mobile OSes *can* and *should* offer abstractions for apps to manage their sensitive data rigorously *without* sacrificing their performance, usability, or other properties. This paper introduces one such abstraction in CleanOS, whose goals we next describe.

## 3 Goals and Assumptions

**Goals.** The primary goal of CleanOS is to minimize the exposure of an app's *allocated* sensitive data by evicting it from the device whenever the data is idle (i.e., not being actively used by the application). The key insight that makes this possible is the tight integration between today's mobile apps and cloud services. CleanOS leverages clouds to create a new abstraction, called a *sensitive data object* (SDO). SDOs track sensitive information as

[2] For RAM, we conservatively assume an object to be persistent if it always appears in the app's Java object dump.

it flows through RAM and stable storage. As soon as they become idle, they are automatically evicted to the cloud and are recovered only when the app needs them again.

Specific design goals of CleanOS include:

1. *Eviction:* SDOs should "disappear" as soon as they become idle whether or not they are expected to be used by an application in the future.
2. *Reasonable performance:* We seek to provide reasonable performance for popular mobile apps despite data eviction over Wi-Fi or cellular networks.
3. *Reasonable defaults:* While we admit app changes for best performance and semantics, we aim to offer reasonable defaults even for unmodified apps.
4. *Leverage technology trends:* CleanOS must integrate naturally with existing tech trends, such as the tight integration of mobile apps with cloud services.
5. *Design for mobiles:* CleanOS' design should target mainstream mobile technologies, such as Android.

Eviction of idle data (Goal 1) is our primary goal and contribution in CleanOS. We strive to ensure that a thief cannot get a "free lunch" by capturing a device. Rather, he should be required to contact the cloud in order to access data of interest, at which time the cloud could deny access, log it, rate-limit it, etc. However, enforcement of precise timeouts on idle sensitive data is a non-goal. From a performance perspective (Goal 2), we wish to ensure that popular apps remain usable despite eviction across Wi-Fi or cellular networks (e.g., 3G/4G).

A common pitfall when proposing new OS abstractions is to require application changes to gain any benefit. To avoid this, CleanOS should include heuristics to construct default SDOs that provide reasonable eviction and performance properties even for unmodified apps (Goal 3). Finally, we aim to exploit unique properties of popular mobile technologies in CleanOS' design (Goals 4 and 5). First, we leverage the tight integration between most mobile apps and trusted cloud services to evict device data to those services. For local-only apps, however, the user can still integrate them with his own CleanOS service. Second, while the data eviction concept is applicable to any mobile OS, we focus our design on Android, which lets us leverage its technological properties to facilitate data eviction. For example, since all Android apps are written in Java, we decided to tap into the garbage collector to evict idle sensitive Java objects.

**Threat Model.** Our threat model considers *any data on a mobile device to be vulnerable to data-driven thieves*. While many data protection systems exist – including encrypted file systems [26, 38, 11], encrypted RAM [34, 23, 31], and data wipeout systems [3, 21] – they are imperfect when confronted with negligent users or (sophisticated) physical attacks. First, users can foil any protection system by not locking their devices [32], assigning trivial PINs or passwords [20], or writing passwords down in easily retrievable locations [36]. Second, mobile devices are prone to physical attacks, which are notoriously difficult to protect against. For example, an attacker could use cold boot attacks [19] to retrieve in-RAM decryption keys or data, break the seal of tamper-resistant hardware [1, 35], or shield the device from the network to prevent remote wipeout [3]. Such threats are especially relevant for corporate, government, and military users, who interact with particularly sensitive data, such as trade secrets, customer data, health data, or state secrets.

To maintain post-loss control over data despite such threats, CleanOS evicts data to a cloud service, which is assumed to be trusted and non-compromisable. In reality, mobile users are already required to trust the clouds on which their apps rely, so our assumption is reasonable. Depending on the deployment model, these clouds could integrate directly into CleanOS to help cleanse their apps automatically. For apps without a cloud component, we assume that users can evict data to a trusted community or self-administered CleanOS service. Finally, we assume that the cloud learns about a monitored device's theft, either directly from the user or via an automatic mobile-theft detection mechanism.

CleanOS explicitly assumes that the mobile device, along with all software running on it, is trusted until it is lost. For example, the thief cannot install malware on a user's device, tamper with the device physically, or inspect it prior to stealing the device. After loss, we trust neither the hardware nor the software on the device.

We assume that disconnection is the exception rather than the rule. With pervasive wireless and cellular network coverage, this assumption is becoming increasingly realistic. Moreover, CleanOS is especially geared toward cloud-based apps, which typically require connectivity for full functionality. Nevertheless, we present techniques to allow disconnected operation in certain cases.

CleanOS is most applicable to long-lived daemon-like apps, whose execution consists of brief computation sessions interspersed with long periods of inactivity. Most of today's mobile apps follow this model, including email, browsers, document editors, and social apps. CleanOS disables exposure during periods of inactivity.

Finally, we explicitly assume the existence of robust secure deallocation and OS buffer-cleanup techniques [6, 30, 10] and do not aim to improve the state of the art in these intensely-researched directions. Rather, we focus on limiting the exposure of sensitive data that *applications* hoard or leak, a problem previously thought intractable from an OS perspective (see §2).

## 4 The CleanOS Architecture

We now describe our CleanOS design for Android. We focus initially on in-RAM data eviction, after which we show how to extend SDOs to stable storage.

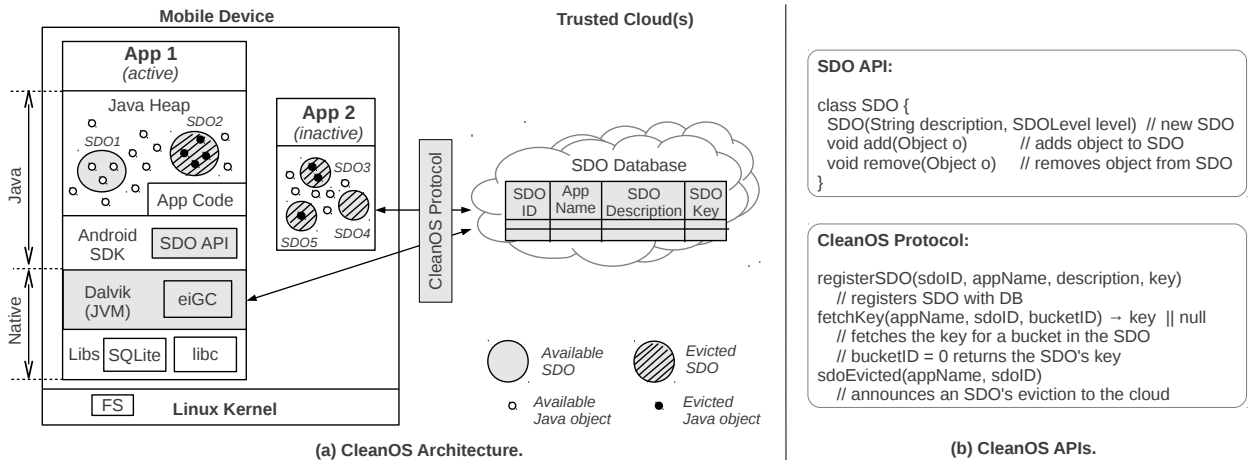



Figure 3: **The CleanOS Architecture and APIs.** (a) The architecture, with key components highlighted in grey. We add or modify in some way all of the boxed components (except for FS and kernel). (b) The CleanOS SDO API and device-cloud protocol.

## 4.1 CleanOS Overview

Figure 3(a) shows the CleanOS architecture, which includes three major components: (1) the *sensitive data object* (SDO) abstraction, (2) a modified, eviction-aware version of the Dalvik interpreter, along with an *evict-idle garbage collector* (eiGC), and (3) the SDO cloud store. Briefly, apps create SDOs and place their sensitive Java objects in them. The modified Dalvik tracks their propagation across RAM with TaintDroid and monitors their bytecode-level accesses. The eiGC evicts SDOs to the cloud if they remain idle for a specified period.

An SDO is a logical collection of Java objects, such as string objects representing the emails in a thread or objects pertaining to a bank account in a finance app. Upon creation, SDOs are assigned app-wide unique IDs and encryption keys ($K_{SDO}$), and are registered with the cloud.

We implement three functions for SDOs. First, we track objects in an SDO with a modified TaintDroid system, using the ID as a taint. As objects are tainted with an SDO's ID, they become part of the SDO. For example, `SDO1` in Figure 3(a) includes three objects added to it either explicitly by the app or automatically by our modified Android framework. Second, we monitor accesses to SDOs and record their timings. Whenever an app accesses an object in an SDO (e.g., to compute on it, send it over the network, or display it on screen), that SDO is marked as used. Third, we evict SDOs when they are idle for a time period (e.g., one minute).

To evict idle SDOs, the eiGC eliminates unused Java objects from RAM *even if* they are still reachable. It periodically sweeps through Java objects and evicts them if they are tainted with an idle SDO's ID. In Figure 3(a), the active app (App 1) has one available SDO (`SDO1`) and one evicted SDO (`SDO2`). For example, an SDO associated with an email thread might be available while the user reads emails in that thread, but the password SDO might remain evicted. When the app goes into the background, all of its SDOs might be evicted, as shown for App 2. An SDO is evicted when all Java objects in it have been evicted; however, an available SDO may have both evicted and available Java objects.

Conceptually, eviction occurs at the level of logical SDOs. In practice, however, CleanOS must eliminate the actual data-bearing objects from the vulnerable device. To do so, eiGC leverages encryption-based data destruction from assured-delete file systems [30, 16] and applies it to the memory subsystem. Specifically, eiGC replaces data-bearing fields in objects, such as primitives and arrays of primitives, with encrypted versions and then securely destroys the encryption key. To encrypt a data field $F$, eiGC uses a key $K_F$ that is uniquely generated from the SDO's key $K_{SDO}$ in the cloud (see §5 for details). We modified Dalvik to fault when an app attempts to access the evicted data, at which time it retrieves $K_{SDO}$ from the cloud, generates $K_F$, and decrypts the data. $K_{SDO}$ is then cached onto the device and securely removed when the SDO as a whole is again evicted.

We next provide more detail on the two main contributions of CleanOS: the SDO abstraction and the eiGC.

## 4.2 The SDO Abstraction

SDOs fulfill two functions in CleanOS. First, they let CleanOS identify sensitive data and focus its cleansing on that data for improved performance. Indeed, evicting all Java objects indiscriminately would be prohibitively expensive, while evicting a few at random would diminish security benefits. Second, SDOs are instrumental in supporting some of our envisioned add-on cloud services, such as the auditing service described in §6, as they identify and classify sensitive data for the auditor.

**APIs.** Figure 3(b) shows the SDO API. To realize the data-control benefits of CleanOS, apps create SDOs and `add`/`remove` Java objects to/from them. To create an SDO, an app specifies a description, which is a short, human-readable string that describes the sensitive data

associated with the SDO. For example, our modified email app, CleanEmail, creates an SDO using "*password*" for the description and `add`s the password object to it. It also creates one SDO for each email thread, specifying the thread's subject as the description, and `add`s each email in the thread to it. Section 6 describes two apps that we trivially ported to CleanOS with minimal modifications (fewer than 10 LoC).

Figure 3(b) also shows the protocol used to register SDOs, retrieve their keys after eviction, and report their eviction to the cloud. To create an SDO, the app registers it with the cloud database using the SDO API, specifying its ID, the app's package name (such as `com.android.email`), the description, and the encryption key. For example, the description for an SDO associated with a certain thread might be the subject of that thread. In the database (whose schema is included in Figure 3(a)), the tuple ⟨app package name, SDO ID⟩ is a phone-wide unique identifier. Although not implemented in our current prototype, the database can use the app user id to restrict access to keys only to the apps that created them. Finally, to enable auditing services such as Keypad [15], CleanOS notifies the cloud asynchronously whenever it evicts an SDO (message `sdoEvicted`). The notification is needed because, unlike Keypad, CleanOS does not forcibly evict keys at an exact time after they were fetched; rather, it does so when convenient, depending on load and networking conditions (see §4.5 and §5).

**Default SDOs.** As mentioned in §3 (Goal 3), we aim not to rely on app modifications to gain tangible benefit from CleanOS. To this end, we modified the Android framework to register a set of default SDOs and use simple heuristics to identify and classify Java objects coarsely on behalf of the apps. For example, our current prototype creates several default SDOs by plugging into various core classes in the SDK: a *User Input* SDO for all input a user types into the keypad (class `InputConnection`), a *Password* SDO for any Java objects that capture input a user types into a password field (based on attributes of class `TextView`), a coarse *SSL* SDO for all objects read from incoming SSL connections (class `SSLSocket`), and SDOs for input from particularly sensitive sensors, such as the camera. Some of these heuristics (e.g., SSL) were inspired by prior work on automatic identification of sensitive data [9]. Although default SDOs are coarse and may potentially include many non-sensitive objects (particularly SSL), we believe that they offer comprehensive identification of most sensitive data in unmodified apps. For example, all the sensitive data we analyzed in §2 would be capturable by a default SDO. For apps willing to adapt, CleanOS allows the overriding of default assignments of objects to SDOs.

**Eviction Granularities and Buckets.** Thus far, eviction granularities have been determined by SDOs, which is problematic for two reasons. First, it forces app writers to consider granularities and taint propagation when they design their SDOs. Second, our default SDOs, such as SSL, are coarse. In our view, CleanOS should offer eviction benefits even when an app dumps all of its sensitive objects into one big SDO, e.g., "*sensitive*."

To support fine-grained eviction with coarse-grained SDOs, we introduce *buckets*. Specifically, an SDO is "split" into several disjoint buckets, which are evicted independently. Java objects added to the SDO – either by the app or by our framework – are placed in random buckets. Eviction occurs at the bucket level: when a bucket has been idle for a period, all objects in it will be evicted using a bucket key, which is derived from the SDO's key using a key derivation function [22]. For example, in an unmodified email app, we would place all emails into the *SSL* SDO. With buckets, different emails would be placed into different buckets of the *SSL* SDO and might therefore be evicted independently. Also, with bucketing, we cache bucket keys instead of SDO keys on the device.

## 4.3 The Evict-Idle Garbage Collector

A simple but important innovation in CleanOS is the evict-idle GC (eiGC). At its core (and independent of CleanOS), eiGC implements for a managed language what swaping implements for OSes: it monitors when objects are being accessed during bytecode interpretation and evicts them when they have not been used for a while. We believe that the eiGC concept has applications beyond CleanOS, such as limiting the amount of memory used by Java applications on memory-constrained devices at a finer grain than OS-level paging would be able to sustain. In the context of CleanOS, however, eiGC evicts Java objects in idle SDO buckets.

Using the GC to evict sensitive data is not the only design worth considering when building a "clean" OS. We contemplated modifying the kernel's paging mechanism to swap idle pages to a Keypad-like encrypted file system [15], which at its core achieves for files a similar eviction function to the one we achieve for RAM. We chose the GC approach for two reasons. First, evicting Java objects provides finer-grained control over sensitive-data lifetime than full-page eviction. Second, by evicting at the JVM level we can leverage TaintDroid, the only taint tracking system for Android. Tracking sensitive data is vital for constructing the SDO abstraction, which in turn is the base for building powerful add-on services, such as auditing. However, our decision has a downside: coverage. By evicting Java objects, we may miss data intentionally maintained by native libraries. We discuss this limitation further in §8.

## 4.4 SDO Extension to Stable Storage

Like RAM, stable storage requires sanitization. At first glance, systems such as Keypad [15] could be directly

leveraged to evict unused files in CleanOS. Unfortunately, we found that eviction at file granularity is unsuitable for Android, where apps typically rely on a database layer to manage their data. For example, 11 of the 14 apps in Figure 2(b) store their data in SQLite, which maps entire databases as single files in the FS. As a result, if the DB file were exposed, then *all* of its items would be exposed, including long-unaccessed emails and documents.

CleanOS tailors storage eviction specifically for Android by extending the in-RAM SDO abstraction to include files *and* individual database items. For this, we use two mechanisms. First, we propagate SDO taints to files and database items. Unfortunately, TaintDroid supports only the former, not the latter, an important vulnerability we discuss in §5. We fixed this in CleanOS by modifying the SQLite DB. Specifically, we automatically alter the schema of any table to include for each data column, *C*, a new column, *Taint_C*, which stores the taint for each item in that column (SDO ID and bucket ID). Second, before storing a tainted data object in a DB, we first *evict* that object, i.e., encrypt it with its eviction key. When the database needs the object, it must decrypt it.

## 4.5 Disconnected Operation

While we assume that disconnection is the exceptional case, we present techniques to deal with two types of disconnection: (1) short-term disconnection, such as temporary connectivity glitches, and (2) long-term, predictable disconnection, such as a disconnection during a flight. To address short-term disconnection, we can extend eviction of already available SDOs by a bounded amount of time (e.g., tens of minutes). This allows an app to continue executing normally while temporarily disconnected until it reaches an evicted object. For example, a user might be able to load recently accessed emails, but not older ones.

To address long-term disconnection, such as during air travel, we hoard SDO keys before entering into disconnection mode. For example, our prototype implements Dalvik support for hoarding SDO keys upon receipt of a signal. We plan to wrap this functionality into a privileged app that provides users with a "Prepare for Disconnection" button, which they can press before boarding a flight. To prevent a thief from using this button to retrieve all SDO keys, the cloud would require the user to enter a password. While we generally shun user-configured passwords in CleanOS, we believe that long-term disconnection is a sufficiently rare case to warrant enforcement of particularly strong password rules with limited impact on usability [27]. In contrast, imposing such rules on frequent unlock operations would be impractical.

## 4.6 Deployment Models

CleanOS presents multiple deployment opportunities. First, security-conscious apps can use their own, dedicated clouds to host keys and provide add-on services, such as auditing. In such cases, we expect that the mobile side of apps would define meaningful SDOs. Second, users who are particularly concerned with apps that have not yet integrated with CleanOS might use a CleanOS cloud offered by a third party or that they host themselves. For example, our prototype hosts all keys for all apps on a Google App Engine service that we implemented.

# 5 Prototype Implementation

We built a CleanOS prototype by modifying Android 2.3.4 and TaintDroid in significant ways (see Figure 1). To date, our prototype fully implements eviction of in-memory SDOs and propagates taints to SQLite, but it does not yet encrypt sensitive items in SQLite. Doing so will require changing the native part of the SQLite library – a single, massive, over-100K-LoC file – the major deterrant we encountered thus far. We next describe modifications we made to components of particular interest.

**TaintDroid with Millions of Taints.** Most dynamic taint-tracking systems, including TaintDroid, support limited numbers of taints, which would prevent CleanOS from scaling to many SDOs. For example, TaintDroid supports only 32 taints by representing them as 32-bit shadow tags, where each taint corresponds to one tag bit. This limitation allows propagation of multiple taints on one object for tracking completeness and security against malicious applications. For CleanOS, which trusts applications, we modified propagation to allow many taints.

We rely on a simple observation, which we validate experimentally: in practice, when multi-tainting occurs, we can usually define a strict, natural ranking for taints in terms of their sensitivity. As intuitive examples, a *Password* SDO should be more sensitive than a generic *User Input* SDO, and a KeePass secret's SDO should be more sensitive than its description SDO. In these cases, "losing" the less sensitive taint would be admissible, because it does not weaken the user's perception of the gravity of an object's exposure. Using a 24-hour real-usage trace for the Email app (see §7.1), we confirmed that 98.8% of the tainted objects were either assigned a single taint during their lifetimes or received multiple taints whose sensitivity could be strictly ordered using a simple, static, three-level ranking system: `HIGH`, `MEDIUM`, and `LOW`. The remaining 1.2% of the objects received multiple taints of undecidable ordering within this ranking system (i.e., equal sensitivity levels). Similar traces for Facebook and Mint indicated even fewer undecidable cases ($< 0.01\%$).

Based on this observation, we introduce the concept of *sensitivity level* for taints and use it to propagate a single taint per object. Apps specify a sensitivity level for each SDO upon its creation. If an object were added to two SDOs during taint propagation, CleanOS retains the one with the higher sensitivity level. For equal sensitivities (the rare case), CleanOS retains the most recent


Figure 4: **CleanOS Taint Tag Structure.** We impose a structure on TaintDroid taints to support arbitrary numbers of taints.

taint. Figure 4 illustrates the revised structure for the taint tag, in which we pack together the sensitivity level, SDO ID, and bucket ID into 32 bits while supporting up to $2^{25}$ SDOs. In our experience, assigning sensitivity levels to SDOs is natural, as demonstrated in §6. The idea of propagating a single taint was used before in hardware-based taint tracking systems for improved performance [5].

**Eviction-Aware Interpretation in Dalvik.** We reserved the most significant bit in the taint tag to denote the eviction state of a field. We modified the Dex bytecode instructions that access object instance fields and array members. This includes instructions such as `OP_AGET`, `OP_IGET`, `OP_SGET` (used to retrieve array members, instance fields, and static fields, respectively). Our new instruction implementations first test the value of the eviction bit in the field's taint tag. When the bit is set, we request the aforementioned $K_{SDO}$ and decrypt the value before allowing the instruction to proceed. If a key is not available, execution is suspended.

**The Evict-Idle Garbage Collector.** While eiGC walks the reachable objects, we inspect the taint tag for each object field and retrieve its idle time. If it exceeds the configured threshold, then eiGC retrieves the key associated with the tag and encrypts the value. Only fields that represent actual data are evicted (primitives and arrays of primitives); fields implemented as pointers are not evicted, as a pointer is not in and of itself sensitive.

To evict data, we use AES in counter mode to generate a keystream, which we use as input to an XOR operation with each byte of the data to be evicted. The size of the keystream depends on the data's type. For primitives, it is either 4 bytes (for `char`, `int`, `float`, etc.) or 8 bytes (for `double` or `long`). For arrays, many bytes may be necessary. We use the bucket key to generate an appropriately sized keystream. For primitives, we replace the data with a pointer to a structure containing metadata necessary for decryption (e.g., initialization vectors) and the resulting ciphertext. For arrays, we evict the contents in place and store the necessary metadata inside the ArrayObject.

Running the eiGC continuously would prevent the CPU from turning off when the mobile device is idle, thereby wasting energy. Fortunately, eiGC needs to run only while sensitive objects are left unevicted. Hence, in our prototype, eiGC stops executing as soon as it has evicted all data, which should occur shortly after the app goes idle. The eiGC resumes execution once the app faults on an evicted object or assigns a new taint to an object. Hence, eiGC runs only while the app also runs.

**Optimizations: Bulk Eviction and Prefetching.** Performance and energy are major concerns with CleanOS, for two reasons. First, garbage collection is expensive; hence performing it frequently hurts app performance and energy (e.g., the eiGC's full-heap scans block interpretation for 1-2s). Second, our reliance on the network to fetch decryption keys causes app delays and dissipates energy.

To address the first problem, we developed *bulk eviction*, in which the eiGC evicts sensitive Java objects *all at once*, soon after the app itself becomes idle. More specifically, while the app is executing, we evict nothing and perform no GC; once the app has remained idle for a predefined time (e.g., one minute), the eiGC performs a full-heap scan-through and evicts all cleartext tainted objects. This technique reduces the number of heavyweight GCs to just one per app execution session, thereby minimizing the eiGC's impact on performance and energy.

To address the second problem, we developed *bulk key prefetch*, which prefetches all keys that were accessed during the last eviction period upon the app's first miss on a key. For example, if a user opened his inbox subject list and read two emails during a previous interaction session with his email app, then the next time the user brings the app into the foreground, CleanOS will fetch the decryption keys for the subjects and the two emails' contents – all in one network request. If the user views only his subject list but reads no emails in a previous session, then the next time around, CleanOS will fetch only subject keys again, not any email content keys. This technique improves app launches and the latency of repeated operations, such as re-reading an email. It can be extended to prefetch keys used in the last $N$ sessions.

Although these optimizations may improve performance and energy, they may also increase sensitive-data exposure. For example, prefetching previously-used keys may expose some sensitive data needlessly. We quantify this performance/exposure tradeoff in §7.2.

**Multi-Level Secure Memory Deallocation.** Android goes to great lengths to keep an application running in the background so it can re-launch quickly. This can cause an accumulation of sensitive data in areas of memory that are no longer in use but have not been returned to the kernel. The object heap in Dalvik is implemented using dlmalloc mspaces and relies on the implementation of `free()` in dlmalloc to return memory to the mspace. To implement secure deallocation, we changed both `free()` and an Android-specific modification to dlmalloc that merges chunks of adjacent free memory. These functions now overwrite the space being released with a fixed pattern. We also modified Dalvik to overwrite interpreted stack frames on method exit, scrubbing them of sensitive data. Finally, when assigning default taints to Java objects, we made explicit efforts to taint objects as soon as they enter Java space from native libraries.

**Addressing a TaintDroid Vulnerability.** When implementing CleanOS, we uncovered a surprising implication

of a known limitation in TaintDroid. Specifically, TaintDroid does not track changes in native libraries, which, as acknowledged by its authors, may allow a *malicious* library to leak tainted data without triggering an audit log. To address this problem, TaintDroid prevents untrusted apps from loading any native libraries other than system libraries (e.g., SQLite and WebKit), which are included in Android itself and are therefore *trusted*. This measure has thus far been thought sufficient.

Nevertheless, we discovered that even *trusted* system libraries can be exploited by a malicious app to expose tainted data with no alarms. For example, because SQLite is written in native code, a malicious app could wash taints off a tracked data item simply by storing it into the database and reading it back. More generally, any stateful libraries that provide the ability to put and later retrieve data are vulnerable to attacks. Since disabling system libraries is impractical (e.g., 12/14 apps in §2 depend on SQLite), we instead suggest identifying and modifying all stateful system libraries to propagate taints.

To date, we modified two such libraries: SQLite and WebKit. For SQLite, we implemented taint propagation by persisting taints along with the data (see §4.4). For WebKit, we disabled caching of rendered Web pages. While important for security, we leave identifying and fixing other libraries for future work and for now suggest notifying the cloud about a potential leak if sensitive data were handed over to an unchecked native library. We suggest that TaintDroid proceed similarly. We discuss the coverage limitation further in §8.

# 6 Applications

We ported three of our "dirtiest" apps from §2 onto CleanOS and built a proof-of-concept, add-on service.

## 6.1 Extending Apps with SDOs

Although unmodified apps can benefit from the coarse default SDOs that CleanOS offers, they can also define their own SDOs for fine-grained control of sensitive data. To demonstrate how apps can be "ported" to our API, we modified two open-source apps – Email and KeePass – to define fine-grained SDOs. Changes for both apps were trivial. For Email, we added these seven lines of code:

```
SDO subjectSDO = new SDO("Subject", SDO.LOW);
subjectSDO.add(mSubject);
SDO bodySDO = new SDO("Content_of_" + mSubject, SDO.MED);
bodySDO.add(mTextContent);
bodySDO.add(mHtmlContent);
bodySDO.add(mTextReply);
bodySDO.add(mHtmlReply);
```

We added each email's subject to a global, low-sensitivity SDO and created a medium-sensitivity content SDO for its body, using the subject itself as the description. Passwords, already embedded in an SDO by our default heuristics, needed no changes.

For KeePass, changes were similarly trivial (7 lines):

| device | message | time |
|---|---|---|
| cd5493c1befeb9075442862afa046182 | fetchKey(9.408 - com.android.email - password) | 2012-04-28 17:26:50.590000 |
| cd5493c1befeb9075442862afa046182 | registerSDO(com.android.email - Invitation to develop "Clean OS") | 2012-04-28 17:27:01.140000 |
| cd5493c1befeb9075442862afa046182 | fetchKey(30.709 - com.android.keepass - Entry) | 2012-04-28 17:27:48.500000 |

Figure 5: **Screenshot of Audit Service Log in App Engine.**

```
SDO masterSDO = new SDO("Master_key", SDO.MED);
SDO entrySDO = new SDO("Entry", SDO.HIGH);
masterSDO.add(mPassword); // In SetPassword.java
masterSDO.add(masterKey); // In PwDatabase.java
entrySDO.add(password); // In PwEntryV3.java
entrySDO.add(pass); // In EntryEditActivity.java
entrySDO.add(conf); // In EntryEditActivity.java
```

## 6.2 Add-on Cloud Services

CleanOS evicts sensitive data to the cloud to prevent unmediated accesses by device thieves. However, by itself, CleanOS cannot guarantee data security. For example, a thief could interact with the apps in an unlocked device or force all SDOs to decrypt. Therefore, CleanOS provides device-side mechanisms necessary for clouds to build clean-semantic security add-ons, such as assured remote wipeout or data exposure auditing. Such services already exist today (e.g., Apple's iCloud and Gmail's two-step verification), but we maintain that their semantics are unclear given the state of today's devices. We next describe an add-on service we trivially built on CleanOS.

**Prototype Auditing Service.** Inspired by Keypad [15], we implemented an auditing service on CleanOS. Its goal is to provide users with audit logs of what was on the device at the time of theft and what has been accessed since. The auditing service integrates with the CleanOS service and both are hosted on App Engine. When a device registers an SDO or requests a decryption key, the cloud logs that operation with the app name, SDO, and current time. In this way, the user can learn from the audit log exactly what data was leaked. For instance, Figure 5 shows a sample audit log that contains entries for SDO registration and key fetching. Were these operations to occur after the device was stolen, the user will know that the email password and KeePass entry may have been leaked.

Crucial to any auditing system is precision. In the audit log, data in different buckets of the same SDO are indistinguishable. Thus, accessing the data in one bucket may cause false alarms for evicted buckets of the same SDO. Using a finer SDO granularity helps reduce false positives. We evaluate audit precision in §7.1.

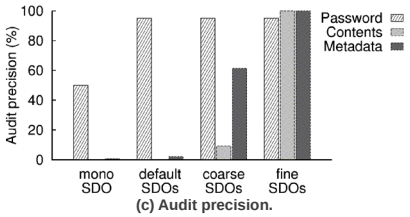
**Further Examples.** A cloud could build many other useful services on CleanOS. For example, the cloud could: allow its mobile users to revoke data access from their missing devices, disable access to sensitive data while the phone is outside the corporate network, and perform theft detection based on access patterns. A variety of entities would find such services useful to host. For example, a company might integrate with CleanOS on the

| App | Password | Contents | Metadata |
|---|---|---|---|
| Email without CleanOS | 100% | 95.5% | 99.0% |
| Email with default SDOs | 6.5% | 5.9% | 5.9% |
| CleanEmail (fine SDOs) | 6.5% | 0.3% | 1.6% |

(a) Sensitive data exposure period.

| App | | Without CleanOS | With CleanOS | | |
|---|---|---|---|---|---|
| | | | 1 bucket | 32 buckets | 1024 buckets |
| Email | Password | 22.5h | 1h 28m | 1h 19m | 1h 11m |
| | Contents | 20.9h | 3 min | 1 min | 1 min |
| | Metadata | 20.9h | 18 min | 6 min | 6 min |
| Facebook | | 24h | 3h 54m | 3h 51m | 3h 29m |
| Mint | | 24h | 1h 10m | 1h 2m | 55 min |

(b) Sensitive data lifetime.




(c) Audit precision.

Figure 6: **Data Exposure.** (a) Fraction of time in which sensitive data was exposed. (b) Maximum sensitive data retention period. (c) Average probability over time that tainted data was actually exposed, given that the audit log shows its SDO as exposed.

device for all corporate apps (e.g., corporate email, customer database), to access its auditing, revocation, and geography-constrained services. Similarly, Gmail could integrate with CleanOS to prevent email exposure after authentication-token revocation.

# 7 Evaluation

We next quantify CleanOS' security, performance, and energy characteristics. Our goal is to show that CleanOS significantly reduces sensitive data exposure while providing reasonable performance and energy consumption, even over cellular networks. We conducted all experiments on rooted Samsung Nexus S phones running CleanOS on Android 2.3.4 and TaintDroid 2.3.

## 7.1 Data Exposure Evaluation

To evaluate the data exposure benefits of CleanOS, we pose three questions: How much does eviction limit exposure of sensitive data? How much do default SDO heuristics limit exposure? How effective is the auditing service? To answer these questions, we recorded a 24-hour trace of one of the authors' phone running CleanOS as it was used to interact with regular apps, including Email, Facebook, and Mint. For Email, we experimented with both the unmodified app and our modified version of it, which we call CleanEmail (see §6.1). The Email app was configured with the author's personal account, which receives about ten new mails daily, and with the default 15-minute refresh period. Facebook and Mint had widgets enabled, which made them continuous services.

**Sensitive Data Exposure Period.** We measured the exposure period for three types of tainted data (password, content, and metadata) in the Email app. Figure 6(a) shows the fraction of time that each type of tainted data was exposed in RAM. Without CleanOS, the password was maintained in RAM all the time, and the content and metadata were exposed over 95% of the time. CleanOS reduced password exposure to 6.5%. For email content, the unmodified Email app with default SDOs reduced exposure time from 95.5% to 5.9%, and modifying the app to support fine-grained SDOs further reduced it to 0.3%. Similar observations held for metadata. To be clear, these results depend on workloads. From another, much more intensive email workload – that registered for many mailing lists and Twitter feeds – we obtained a result of 7.3% and 12.7% for content and metadata, respectively. Overall, results demonstrate a significant reduction in exposure times for tainted data. Moreover, they show that our default heuristics protect sensitive data reasonably well.

**Sensitive Data Lifetime.** As SDO lifetime is critical to system security, we must also examine the maximum period that a tainted object could be retained in RAM. Figure 6(b) shows the retention time for the longest-lived tainted object in three applications, where we break down email into three types. Without CleanOS, all observed applications retained certain tainted objects for more than 20 hours. With CleanOS, the maximum SDO lifetime was dramatically reduced. For instance, the Email app kept some metadata objects for as long as 20.9 hours, which CleanOS reduced to only 6 minutes when using 1024 buckets. For Facebook and Mint, the impact of bucketing on sensitive data lifetime was more limited because these apps tend to use most objects in an SDO at the same time. Overall, these results indicated that the mobile device was significantly cleaner with CleanOS.

**Audit Precision.** We next evaluated the effectiveness of the auditing service we built on CleanOS (see §6.2). We compared audit precision across four levels of SDO granularity in Email: (1) mono-SDO, where we marked data as only "sensitive" or "non-sensitive," (2) default SDOs, where we used default heuristics, (3) coarse SDOs, where the application defined one content SDO and one metadata SDO for all emails, and (4) fine SDOs, where each email had its own content and metadata SDOs. We define *audit precision* as the average probability over time that the tainted data is actually exposed on the device, given that the audit log shows its SDO has not been evicted.

Figure 6(c) shows audit precision for the Email app's password, content, and metadata. Password auditing was 50.0% precise with mono-SDO but increased to 95.1% with default SDOs. The content and metadata, however, had poor precision (<3%) without application support: CleanOS could not differentiate data coming from the Internet and hence added every incoming object to the *SSL* SDO. With coarse, application-specific SDOs, audit precision for email content and metadata was 9.1% and 61.3%, respectively. When fine application-specific SDOs were available, audit precision reached 100%. Thus, our default SDOs were effective in auditing password exposure, but application adaptation was needed to provide precise auditing for other types of sensitive data.

| | Android 2.3.4 | TaintDroid 2.3 | CleanOS | | | |
|---|---|---|---|---|---|---|
| | | | not evicted | evicted, cached | evicted, Wi-Fi | evicted, 3G |
| Untainted Primitive | 0.00021 | 0.00022 | 0.00026 | - | - | - |
| Tainted Primitive | - | 0.00023 | 0.00056 | 1.24 | 22.844 | 336.07 |
| Untainted Array | 0.00027 | 0.00029 | 0.00035 | - | - | - |
| Tainted Array (S) | - | 0.00030 | 0.00075 | 1.4 | 21.652 | 308.71 |
| Tainted Array (M) | - | 0.00030 | 0.00075 | 1.331 | 21.702 | 316.79 |
| Tainted Array (L) | - | 0.00030 | 0.00075 | 2.355 | 22.365 | 317.97 |

Figure 7: **Micro-operation Performance (milliseconds).** CleanOS Java object field access times compared with Android, TaintDroid. Times for non-sensitive and sensitive fields for various eviction states. Averages over 1,000 accesses.

## 7.2 Performance Evaluation

We next evaluate the performance impact of CleanOS under different workloads and networking conditions. Here, we aim to: (1) quantify raw performance overheads, (2) demonstrate that CleanOS is practical over Wi-Fi for popular apps, and (3) show how our optimizations make CleanOS practical even over slow, cellular networks. In our experience, obtaining reliable and repeatable results from the cellular network is tremendously difficult; hence, our results used emulated Wi-Fi and 3G networks with RTTs configured at 20ms and 300ms, respectively. Because our transmission units were tiny (keys were 16-byte long), we did not enforce bandwidth restrictions.

**Micro-operation Performance Overheads.** To evaluate raw performance overheads, we measured Java object field-access times for Android, TaintDroid, and CleanOS. Figure 7 compares them for four field types: primitives (`int`), small arrays (16 bytes), medium arrays (4KB), and large arrays (16KB). For CleanOS, we show access times both for non-sensitive fields (the vast majority) and sensitive fields under various eviction states. CleanOS' access overhead for non-sensitive fields was small compared with TaintDroid (16%), which itself was close to raw Android (6% overhead for TaintDroid). The overhead for sensitive field access increased to 141% over TaintDroid: CleanOS performed last-time-of-use bookkeeping *on every Dalvik field access instruction* (e.g., `OP_AGET`, `OP_IGET`) that involved a tainted field. Further, when evicted, CleanOS access overhead spiked dramatically, especially when the evicted field's key was not cached on the device but was fetched over Wi-Fi or 3G. Moreover, unlike in Android and TaintDroid, access times for evicted arrays in CleanOS depended on the array's size because decryption times increase with data size. For example, the "evicted, cached" column shows that decrypting a tainted array grew by 68% when the array's size increased from 16B to 16KB. Fortunately, in practice, sensitive fields are extremely rare compared with non-sensitive fields. For example, our email trace showed an average of 102,907 fields at any time, of which merely 1,889 were tainted (or 1.83%). Hence, CleanOS should acceptably affect real app performance, as shown next.

**Application Performance.** Figure 8(a) shows the time to launch several popular apps (i.e., bring them into the foreground) and perform typical actions, such as opening an email, viewing a KeePass entry, or loading a Web page. We chose three Web pages: a simple one (`https://iana.org/domains/example`) and two popular and more complex ones (`https://news.google.com` and `https://cnn.com`). For CleanOS, results labeled "not evicted" correspond to cases where all accessed objects were decrypted, while results labeled "evicted" correspond to cases where objects were all evicted.

In the "not evicted" case, interaction with the apps incurred a limited performance penalty compared with both TaintDroid and Android. For example, 8/13 operations incurred less than 100ms penalties over TaintDroid, and 7/13 did so over Android. Such penalties will likely go unnoticed by users, who are known to perceive delays coarsely [29]. Hence, when users interact with a recently used app, they should not feel CleanOS' presence.

When users interact with a cold app ("evicted" columns for unoptimized CleanOS), however, performance degraded but remained usable for Wi-Fi networks. Our cheapest app is the browser, for which CleanOS incurred 8-23% overheads over Android for all operations. The reason is two-fold: (1) the browser deals with little sensitive data, and (2) during page loads, the browser fetches large amounts of data over Wi-Fi, which dwarf CleanOS' key traffic delays. The most expensive app for CleanOS is CleanEmail, which incurred a larger penalty than Email for "evicted" launches due to more granular tainting. For example, while Email needed to fetch 2 keys to load an email, CleanEmail needed to fetch 3 keys.

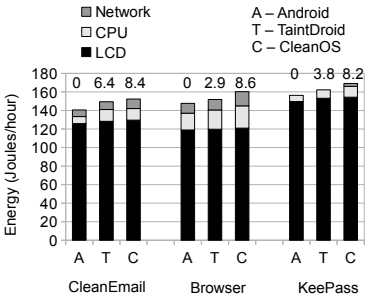
Over 3G, CleanOS penalties after eviction became significant. While some operations remained within reasonable bounds (e.g., launching the browser and loading `iana.org` or `cnn.com`), many operations incurred overheads in excess of 100%. For example, loading an email onto the screen jumped from 197ms to 1.1s for Email and 1.4s for CleanEmail. Such delays likely affect usability.

**Effect of Optimizations on Application Performance.** Column "Optimized CleanOS" in Figure 8(a) shows the elapsed time of repeat operations under our bulk prefetching optimization (see §5). All timed operations were invoked in the previous application session; therefore, all of their relevant keys were prefetched together as part of one bulk request during the timed session. The results show dramatic improvements in performance for both launching and interacting with the apps. For example, CleanEmail – our most expensive application – launched in 589ms over 3G compared with 919ms on unoptimized CleanOS (35.9% improvement) and loaded a previously read email in 420ms compared with 1.4s (71% improvement). In general, this optimization lets an app re-launch incur little more than one RTT over non-evicted CleanOS, while subsequent repeat operations incur no RTT. Natu-

| Application | Action | | Android 2.3.4 | TaintDroid 2.3 | CleanOS | | | Optimized CleanOS |
|---|---|---|---|---|---|---|---|---|
| | | | | | not evicted | evicted, Wi-Fi | evicted, 3G | evicted, 3G* |
| Email | Launch | | 197 | 202 | 241 | 312 | 919 | 589 |
| | Read Message | | 212 | 254 | 387 | 501 | 1165 | 379 |
| CleanEmail | Launch | | - | - | 291 | 315 | 902 | 598 |
| | Read Message | | - | - | 452 | 526 | 1472 | 421 |
| KeePass | Launch | | 173 | 192 | 217 | 221 | 527 | 672 |
| | Read Entry | | 125 | 150 | 146 | 155 | 479 | 135 |
| Browser | Launch | | 130 | 151 | 160 | 144 | 222 | 138 |
| | Load Page (iana) | Wi-Fi | 488 | 483 | 658 | 605 | - | - |
| | | 3G | 2067 | 2114 | 2125 | - | 2136 | 2031 |
| | Load Page (GNews) | Wi-Fi | 1072 | 1043 | 1270 | 1160 | - | - |
| | | 3G | 1717 | 2475 | 2475 | - | 3536 | 2942 |
| | Load Page (CNN) | Wi-Fi | 1065 | 1136 | 1394 | 1446 | - | - |
| | | 3G | 4570 | 4709 | 4325 | - | 4619 | 4538 |

* Actions were performed before.

(a) App Performance (milliseconds).




(b) Energy over Wi-Fi.

Figure 8: **Application Performance and Energy Consumption.** (a) Performance of various popular app activities under Android, TaintDroid, and CleanOS for various eviction states and configurations. Results are averages over 40 runs. (b) Hourly energy consumption attributed by PowerTutor to the three apps when running a long-term synthetic workload for at least 3 hours. Numbers on top of each bar show energy overhead over default Android in percent.

rally, our optimization will not benefit non-repeat operations, such as loading a brand new or long-unread email. However, one type of operation that will always benefit is app launch, a latency-sensitive operation on mobiles.

Despite their performance benefits, our optimizations may increase data exposure. When applying these optimizations to the workloads in §7.1, we obtained limited, but non-trivial, exposure impact. The period for each type of tainted data increased by up to 0.9 percentage points for the workload in Figure 6(a), and by up to 23.2 percentage points for our intensive Email workload. Prefetching keys from multiple sessions would cause further exposure. Hence, CleanOS should best apply this optimization only in specific cases (e.g., over 3G).

**Overhead Estimation for SDO Stable Storage Extension.** Thus far, our results show CleanOS' overheads for eviction of *in-RAM SDOs*. While we have not fully implemented the SDO extension to stable storage, we now offer rough estimates for the extra overheads to expect from such an extension. We expect the major sources of overhead to be: (1) the key fetches required to access encrypted database items, and (2) the extra encryption/decryption that occurs when accessing these items. To account for (1), we ran experiments with our test applications that instruct CleanOS to fetch the appropriate decryption keys for any tainted database items being accessed. To account for (2), we added an extra 20% overhead per query, a number reported by CryptDB [33], which also does per-item encryption.

With this methodology, we estimate that extending SDOs to SQLite would result in additional overheads ranging between 0-65% on 3G over CleanOS with in-RAM SDOs. We predict that these operations will suffer the most: KeePass Launch (869ms, or 64.9% additional overhead), CleanEmail Read (1887ms, or 28.2% additional overhead), and Browser Load (2542ms, 4086ms, and 4573ms for `iana.org`, `news.google.com`, and `cnn.com`, respectively, or 15-19% additional overhead). Most of these overheads (82-99% across all apps) are due to extra RTTs incurred by necessary key fetches, which are optimizable via batch prefetching. Thus, overall, we believe that our system will be practical from a performance perspective even when implemented in full.

## 7.3 Energy and Network Evaluation

CleanOS' encryption, network traffic, and extra GCs raise concerns about its impact on energy consumption. To evaluate this impact, we ran coarse-grained experiments that drove a simple, long-term workload against each app (CleanEmail, Browser, and KeePass) using MonkeyRunner [17] and measured consumption using the PowerTutor online power monitor [42]. The workload repeatedly launched an app, performed a set of typical tasks (such as reading emails, accessing entries in KeePass, and visiting Web pages in the browser), sent the app into the background, and then slept for 15 minutes. Each app interaction lasted for 36-46s, after which we promptly turned off the LCD. We ran the workload continuously for at least 3 hours and plotted per-app power consumption as reported by PowerTutor.

Figure 8(b) shows energy consumption for Android, TaintDroid, and CleanOS over a *real* home Wi-Fi network. For each app, we show the energy consumed by the LCD, CPU, and Wi-Fi. Results show that CleanOS' total energy overheads over Wi-Fi were small compared with both Android and TaintDroid: 8.2-8.4% over Android (see labels above bars) and 1.9-5.5% over TaintDroid. Drilling down on resource overheads, we observe that CleanOS increased energy consumption of both the network (44-45%) and the CPU (32-74%), but those overheads were dwarfed by the LCD energy draw. In general, our overheads were smallest for the browser, which itself consumed relatively more CPU and network energy, and largest for KeePass, a lightweight application that performed little computation and had no network traffic.

Over 3G, energy overheads due to network traffic will likely increase. Our experience shows that experiment-

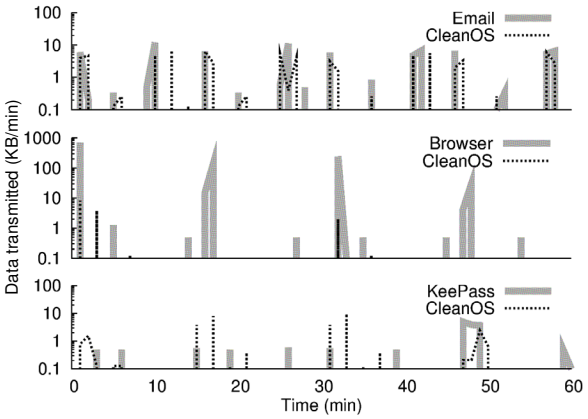



Figure 9: **Network Traffic Patterns of Apps vs. CleanOS.** CleanOS traffic vs. app traffic for a one-hour trace. The Y axis is in log scale. In our cases, the phone has background traffic, which is included in both app and CleanOS lines.

ing with 3G networks leads to very unstable and unrepeatable results; hence, for these networks, we rely on an analytic evaluation grounded in a study of CleanOS' network traffic. Figure 9 compares CleanOS' traffic patterns to those of the three apps using one-hour traces from our energy experiments. It shows that CleanOS' network consumption depends on the application's own network profile. For networked apps, such as email and browser, CleanOS' traffic closely follows the app's own traffic distribution over time. For example, for email, of the 24 minutes during which CleanOS issued some traffic, only 9 of those had no accompanying app traffic; for the browser, only 1 out of 5 one-minute periods did so. From an energy perspective, this means that CleanOS usually piggybacks on the app's own use of the network and only rarely needs to hold the interface up for its own purposes. On the other hand, for local-only apps, such as KeePass, CleanOS uses the network mostly for its own purpose; but even in such cases, however, its traffic will be rare, brief, and small ($\leq 10KB/min$). Thus, we expect CleanOS to be practical from an energy perspective.

## 8 Security Discussion and Limitations

We now discuss CleanOS' security implications and limitations. There are two types of data that an attacker might seek: unevicted data and evicted data. CleanOS does not protect unevicted data on a stolen device; instead, it seeks to minimize the amount of such data. An audit-enabled cloud service can provide users with a robust audit trail of data exposed at the time of loss and data retrieved since. For evicted data, clouds can do much more. For example, after theft has been detected, they can revoke the device's access to still evicted data. They can also monitor accesses to keys to detect anomalous behavior.

A thief might also try to retrieve keys for all evicted SDOs before the cloud disables them. Such aggressive attackers could be identified via anomalous access-pattern detection. To evade detection, the attacker could retrieve SDO keys only for objects of interest, such as emails with tempting subjects. While some attackers may be unwilling to do so for fear of revealing their identities, the cloud can provide an audit log of such accesses.

Attackers might also attempt to break the disconnection password to hoard keys for apps of interest without raising suspicion. CleanOS could enforce sufficient entropy to make the disconnection password, which is extremely rarely used, much stronger than a regular password (which a user must type every time he unlocks his device). However, even if the password were broken, the cloud could provide evidence of the attacker's behavior.

Adversaries may perform network attacks to sniff or disrupt CleanOS device-cloud traffic. To prevent sniffing of keys from network traffic, we encrypt connections and authenticate the device to the cloud using a preestablished secret key (akin to the device token in Gmail's two-factor verification) and the cloud using public key cryptography. An attacker could also disrupt CleanOS device-cloud communication to induce CleanOS into an accumulation mode, where it defers eviction until cloud connectivity returns. To defend, CleanOS bounds its eviction delay for temporary disconnections. Moreover, a thief could prevent eviction messages from arriving at the cloud. However, dropping those messages will not affect confidentiality since data eviction will complete as planned, but it might raise auditing false positives.

One CleanOS limitation is its limited coverage outside the Java realm. To be clear, expunging sensitive data from Java is an important contribution: 9/14 apps in Figure 2(b) would expose some sensitive data permanently in RAM if we did not do so. Moreover, we have incorporated some basic multi-level secure deallocation techniques and have modified two popular native libraries to limit exposure (SQLite and WebKit). However, any data retained in other buffers or caches in the OS or native libraries remains exposed. To limit this exposure, we recommend: (1) incorporating additional OS data scrubbing mechanisms [10], (2) inspecting all remaining system libraries for caches as we do for SQLite and WebKit, and (3) either disabling all third-party libraries (an approach similar to TaintDroid's [12]) or informing the cloud about any data leakages to uninspected third-party libraries.

## 9 Related Work

CleanOS builds upon prior work that we now describe.

**Encrypted File Systems.** Encrypted file systems [11] and full-disk encryption [26, 38] are designed to protect data stored on a vulnerable device, but they do not protect data in RAM. Moreover, as discussed in §3 (Threat Model) and in prior work [15, 41], these systems can fail in the real world due to human factors (e.g., non-existent or poor passwords) and physical attacks (e.g., key re-

trieval from RAM via cold-boot attacks [19]). CleanOS recognizes these limitations and promptly removes unused data from the vulnerable device.

**Encrypted RAM Systems.** Encrypted RAM systems – such as XOM [23], CryptKeeper [31], and encrypted swap [34] – encrypt data while it sits in RAM. CryptKeeper resembles the CleanOS model by encrypting all memory pages except for a small working set, thereby achieving a similar encrypted-unless-in-use effect as CleanOS. However, while the data is encrypted in these systems, the decryption keys themselves are still available in RAM and potentially accessible to memory-harvesting unless extra hardware is deployed. Moreover, if the device were unlocked or the thief found the user's password, encrypted RAM would have no effect.

ZIA [7, 8] encrypts mobile data in RAM and on disk whenever a device is not near its owner. The user wears a beaconing token at all times, whose presence is detected by the mobile. Like ZIA, CleanOS encrypts data after a period of non-use, but the granularities, method, and usage model are different. For example, we disable unused data at the Java object level as opposed to the device level, evict data to clouds for increased post-theft control, and do not require users to carry (and secure!) tokens.

**Mobile Wipe-Out Systems.** Varied commercial wipe-out systems exist and help increase users' post-theft data control. For example, remote wipe-out systems, such as iCloud [3], let the users send "kill" messages to lost devices. Unfortunately, these systems require network connectivity to function correctly. If the thief prevents device connectivity (e.g., by wrapping it into a Faraday cage), the device will not receive the message and therefore not complete its wipeout. Moreover, configuring the device to self-destruct after a number of failed authentication attempts helps prevent access to file system data, but it does not preclude memory harvesting attacks, such as coldboot imaging [19]. Such attacks are particularly problematic on mobile devices, which hardly ever power off.

**Cloud-based Mobile Security Services.** The value of the cloud for increased data control is being increasingly recognized. Examples of cloud-based security services include: online data access revocation with two-step verification [18], location-based access control with location-aware encryption [37], and cloud-based authentication with capture-resilient cryptography [25]. Generally, these systems prevent the compromise of data not already exposed on the device, but they do not guarantee security for mobile-resident data. For example, none of these systems takes RAM-resident data into account, and the Google two-step verification does not even consider storage. CleanOS cleanses device RAM and storage in support of such security services.

**Keypad.** Particularly relevant is Keypad [15], an auditing file system for old-generation mobile devices, such as laptops and USB sticks, that achieves file-level, strong-semantic auditing. CleanOS shares Keypad's threat model, and our auditing service was inspired by it. However, in addition to its support for in-RAM data auditing, CleanOS also differs from Keypad in its focus on new-generation mobile technologies, such as Android, which have distinct auditing granularity requirements. For example, file-level auditing in Keypad would be ineffective for apps using the SQLite database since they all would be stored within one single file. Instead, CleanOS defines SDOs, an abstraction that encompasses fine-grained objects, database items, and sdcard files.

**Secure-Deletion Systems.** Secure deletion has been recognized as a key OS primitive. It erases data in memory [6], OS buffers [10], and stable storage [30, 39, 4] once the data is not needed by the application. CleanOS explicitly assumes the existence and robustness of such systems, but addresses a distinct, important part of the sensitive data exposure problem for the first time: securing data explictly hoarded by applications for performance or convenience. CleanOS SDOs resemble the self-destructing data abstraction in Vanish [16] in that they "disappear" over time, but the setting is different: Vanish makes Web data disappear after a specified time post-creation, whereas SDOs make mobile data disappear if they are unused for a specified time.

## 10 Conclusions

This paper described CleanOS, a new design for the Android OS that manages sensitive data rigorously and keeps mobile devices clean at any point in time. Unlike Android, which lets sensitive data accumulate in cleartext RAM and on disk, CleanOS eliminates it from the vulnerable device by evicting it to the cloud whenever it is not needed on the device. It provides a clean-semantic foundation for clouds to build add-on services, such as data access revocation after a device has been lost or post-theft data exposure auditing. We implemented CleanOS by instrumenting Android's Java virtual machine to securely evict sensitive data objects after a specified period of non-use. On top of CleanOS, we built a sample auditing cloud service. Our experiments demonstrate that CleanOS limits its data exposure significantly while imposing acceptable performance overheads and offering sound semantics for cloud-based applications.

## 11 Acknowledgements

We thank our shepherd, Petros Maniatis, and anonymous reviewers for their valuable comments. We also thank Steve Gribble, Angelos Keromytis, Hank Levy, Simha Sethumadhavan, Salvatore Stolfo, and Junfeng Yang for their feedback. This work was supported by DARPA through FA8650-11-C-7190 and FA8750-10-2-0253 and NSF through CNS-0905246.

## References


[1] R. Anderson and M. Kuhn. Tamper resistance: A cautionary note. In *Proc. of the USENIX Workshop on Electronics Commerce*, 1996.

[2] Android Developers Blog. Avoiding memory leaks. android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html, 2009.

[3] Apple iCloud. Find my iPhone, iPad, and Mac. www.apple.com/icloud/features/find-my-iphone.html, 2012.

[4] D. Boneh and R. Lipton. A revocable backup system. In *Proc. of USENIX Security*, 1996.

[5] S. Chen, M. Kozuch, T. Strigkos, and et.al. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[6] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. of USENIX Security*, 2005.

[7] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proc. of the ACM Annual International Conference on Mobile Computing and Networking*, 2002.

[8] M. D. Corner and B. D. Noble. Protecting applications with transient authentication. In *Proc. of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[9] L. P. Cox and P. Gilbert. Redflag: Reducing inadvertent leaks by personal machines. Technical Report TR-2009-02, Duke University, 2009.

[10] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] EncFS. www.arg0.net/encfs, 2010.

[12] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[13] Federal Communications Commission. Announcement of new initiatives to combat smartphone and data theft. www.fcc.gov/document/announcement-new-initiatives-combat-smartphone-and-data-theft, 2012.

[14] Future of Privacy Forum, Center for Democracy & Technology. Best practices for mobile applications developers. www.futureofprivacy.org/wp-content/uploads/Apps-Best-Practices-v-beta.pdf, 2011.

[15] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[16] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of USENIX Security*, 2009.

[17] Google Inc. MonkeyRunner. developer.android.com/tools/help/monkeyrunner_concepts.html, 2012.

[18] Google Inc. Two-step verification. support.google.com/accounts/bin/topic.py?hl=en&topic=28786, 2012.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of USENIX Security*, 2008.

[20] Imperva. Consumer password practices. www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf, 2010.

[21] Intel Corporation. Laptop security with Intel Anti-Theft technology. www.intel.com/content/www/us/en/architecture-and-technology/anti-theft/anti-theft-general-technology.html, 2012.

[22] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). tools.ietf.org/html/rfc5869, 2010.

[23] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[24] Lookout Mobile Security. Lost and found: The challenges of finding your lost or stolen phone. blog.mylookout.com/blog/2011/07/12/lost-and-found-the-challenges-of-finding-your-lost-or-stolen-phone, 2011.

[25] P. MacKenzie and M. Reiter. Networked cryptographic devices resilient to capture. In *Proc. of USENIX Security*, 2001.

[26] Microsoft Corporation. Windows 7 BitLocker executive overview. technet.microsoft.com/en-us/library/dd548341(WS.10).aspx, 2009.

[27] Microsoft Corporation. Create strong passwords. www.microsoft.com/security/online-privacy/passwords-create.aspx, 2012.

[28] M. Milian. U.S. government, military to get secure Android phones. www.cnn.com/2012/02/03/tech/mobile/government-android-phones/index.html, 2012.

[29] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[30] R. Perlman. File system design with assured delete. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[31] P. A. H. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *Proc. of the IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[32] Ponemon Institute. The lost smartphone problem. www.mcafee.com/us/resources/reports/rp-ponemon-lost-smartphone-problem.pdf, 2011.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[34] N. Provos. Encrypting virtual memory. In *Proc. of USENIX Security*, 2000.

[35] J. Robertson. Security chip that does encryption in PCs hacked. www.usatoday.com/tech/news/computersecurity/2010-02-08-security-chip-pc-hacked_N.htm, 2010.

[36] M. Savage. NHS 'loses' thousands of medical records. www.independent.co.uk/news/uk/politics/nhs-loses-thousands-of-medical-records-1690398.html, 2009.

[37] A. Studer and A. Perrig. Mobile user location-specific encryption (MULE): Using your office as your password. In *Proc. of the ACM Conference on Wireless Network Security (WiSec)*, 2010.

[38] Symantec Corporation. PGP whole disk encryption. www.symantec.com/whole-disk-encryption, 2012.

[39] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. FADE: Secure overlay cloud storage for file assured deletion. In *Proc. of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.

[40] W3C. Mobile app best practices. www.w3.org/TR/mwabp, 2010.

[41] A. Whitten and J. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proc. of USENIX Security*, 1999.

[42] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2000.

# COMET: Code Offload by Migrating Execution Transparently

Mark S. Gordon† D. Anoushe Jamshidi† Scott Mahlke† Z. Morley Mao† Xu Chen⋆

†EECS Department,
University of Michigan
Ann Arbor, MI, USA
{msgsss,ajamshid,mahlke,zmao}@umich.edu

⋆ AT&T Labs - Research
chenxu@research.att.com

## Abstract

In this paper we introduce a runtime system to allow unmodified multi-threaded applications to use multiple machines. The system allows threads to migrate freely between machines depending on the workload. Our prototype, COMET (Code Offload by Migrating Execution Transparently), is a realization of this design built on top of the Dalvik Virtual Machine. COMET leverages the underlying memory model of our runtime to implement distributed shared memory (DSM) with as few interactions between machines as possible. Making use of a new VM-synchronization primitive, COMET imposes little restriction on when migration can occur. Additionally, enough information is maintained so one machine may resume computation after a network failure.

We target our efforts towards augmenting smartphones or tablets with machines available in the network. We demonstrate the effectiveness of COMET on several real applications available on Google Play. These applications include image editors, turn-based games, a trip planner, and math tools. Utilizing a server-class machine, COMET can offer significant speed-ups on these real applications when run on a modern smartphone. With WiFi and 3G networks, we observe geometric mean speed-ups of 2.88X and 1.27X relative to the Dalvik interpreter across the set of applications with speed-ups as high as 15X on some applications.

## 1 Introduction

Distributed Shared Memory (DSM) systems provide a way for memory to be accessed and modified between computing elements. It was an active area of research in the late 1980s. Classically, DSM has been applied to networks of workstations, special purpose message passing machines, custom hardware, and heterogeneous systems [18]. With the onset of relatively low performance smartphones, a new use case for DSM has presented itself. In our work, we apply DSM to offloading – the task of augmenting low performance computing elements with high performance elements.

Offloading is an idea that has been around as long as there has been a disparity between the computational powers of available computing elements. The idea has grown in popularity with the concept of ubiquitous computing where many low-powered, well-connected computing elements would exist that could benefit from the computation of nearby server-class machines. The popular approach to this problem is visible in specialized systems like Google Translate and Apple iOS's Siri. For broad applicability, COMET and other recent work [9, 8] have aimed to generalize this approach to enable offloading in applications that contain no offloading logic. These systems when compared to specialized offloading systems can offer similar benefits that compilers offer over hand-optimized code. They can save programmer effort and in some cases outperform many specialized efforts. We believe our work is unique as it is the first to apply DSM to offloading. Using DSM instead of remote procedure calls (RPC) offers many advantages including full multi-threading support, migration of a thread at any point during its execution, and in some cases, more efficient data movement.

However, in applying DSM, we faced many challenges unique to our use case. First, the latency and bandwidth characteristics of a smartphone's network connection to an external server are much worse than those of a cluster of workstations using a wired connection. Second, the type of computation is significantly different: while previous DSM systems focused on scientific computing, we aim to augment real user-facing applications with more stringent response requirements. Despite these challenges, we show in §5 that performance improvements can be significant for a wide range of applications across both WiFi and 3G networks.

Our design aims to serve these primary goals:

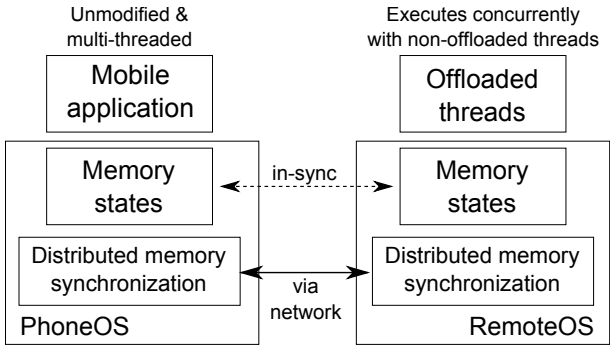



Figure 1: High-level view of COMET's architecture

- Require only program binary (no manual effort)
- Execute multi-threaded programs correctly
- Improve speed of computation
- Resist network and server failures
- Generalize well with existing applications

Building upon previous work, our work offers several new contributions. To the best of our knowledge, we propose a new approach to lazy release consistency DSM [20] that operates at the field level granularity to allow for multiple writers of the same memory unit. This allows us to minimize the number of times the client and server need to communicate to points where communication *must* occur. Using DSM also allows us to be the first offloading engine to fully support multi-threaded computation and allow for threads to move between endpoints at any interpreted point in their execution rather than offloading only entire methods. Despite these features, COMET is designed to allow computation to resume on the client if the server is lost at any point during the system's execution. Finally, we propose a very simple scheduling algorithm that can give some loose guarantees on worst case performance. Figure 1 gives an overview of the design of the system.

We implemented COMET within the Dalvik Virtual Machine of the Android Gingerbread operating system and conducted tests using a Samsung Captivate phone and an eight core server for offloading. Our tests were comprised of a suite of nine real-world applications available on Google Play including trip planners, image editors, games, and math/science tools. From our tests, we found that COMET achieved a geometric mean speedup of 2.88X and average energy savings of 1.51X when offloading using a WiFi connection. 3G did not perform as well with a geometric speedup of 1.27X and a modest increase in energy usage. With lower latency and higher bandwidth, we expect better offloading performance for 4G LTE networks compared to 3G.

The rest of the paper is organized as follows. In §2, we summarize works in the field of DSM and offloading and discuss how they relate to COMET. §3 presents the overall system design of COMET. §4 describes how we implemented COMET using Android's Dalvik Virtual Machine. §5 presents how we evaluated COMET including the overheads introduced by the offloading system, performance and energy improvements for our benchmark suite, and two case studies demonstrating some interesting challenges and how COMET solves them. §6 covers some limitations of our system, and §7 ends with final remarks.

## 2 Related Work

A significant amount of work has gone into designing systems that combine the computation efforts of multiple machines. The biggest category of such systems are those that create new programming language constructs to facilitate offloading. Agent Tcl [14] was one such a system that allowed a programmer to easily let computation "jump" from one endpoint to another. Other systems that fall into this broad category include Emerald [19], Network Objects [5], Obliq [6], Rover [17], Cuckoo [21], and MapReduce [11]. Other work instead focuses on specific kinds of computation like Odessa [26] which targets perception applications, or SociableSense [27] designed for harvesting social measurements. COMET takes an alternative approach by building on top of an existing runtime and requiring no binary modifications.

Other related offloading systems include OLIE [15], which applied offloading to Java to overcome resource constraints automatically. Messer et al. [23] proposed a system to automatically partition application components using MINCUT algorithms. Hera-JVM [22] used DSM to manage memory consistency while doing RPC-style offloading between cores on a Cell processor. Helios [25] was an operating system built to utilize heterogeneous programmable devices available on a device.

Closer to our use case, MAUI [9] enabled automated offloading and demonstrated that offloading could be an effective tool for performance and energy gains. In their system, the developer was not required to write the logic to ship computation to a remote server; instead he/she would decide which functions *could* be offloaded and the offloading engine would do the work of deciding what *should* be offloaded and collecting all necessary state. Requiring annotation of what could be offloaded limited the reach of the system leaving room for CloneCloud [8] to extend this design, filling this usability gap by using static analysis to automatically decide what could be offloaded. Other works, like ECOS [13], attempt to address

the problem of ensuring data privacy to make offloading applicable for enterprises.

JESSICA2 implemented DSM in Java for clusters of workstations. JESSICA2 implemented DSM at the object level using an object homing technique. This approach is not suitable for our use case because each synchronization operation triggers a network operation. Moreover it is not robust to network failures.

Munin [7] was one of a couple early DSM prototypes built during the 90s that targeted scientific computation on workstations. Shasta [29] aimed to broaden the applicability of software DSM by making a system that worked with existing applications. These were followed up by systems like cJVM [4] and JESSICA2 [30] which brought together DSM and Java. However these systems are constructed for low latency networks where the cost communication is relatively low encouraging designs with more communication events and less data transfer. Additionally, as is the case with cJVM and JESSICA2, the DSM design is not conducive to failure recovery.

Our work can be seen as a combination of the efforts of these DSM systems with the offloading frameworks of Maui and CloneCloud. Unlike much of the work on offloading discussed above, we focus not on *what* to offload but more on the problem of *how* to offload. COMET makes use of a new approach to DSM to minimize the number of communication events necessary while still supporting multi-threaded applications. We attempt to demonstrate COMET's applicability by evaluating on several existing applications on Google Play.

# 3 Design

This section contains the overall design of COMET to meet the five goals listed in §1. Although some specific references to Java are made, the design of COMET aims to be general enough to be applied to other managed runtime environments (such as Microsoft's Common Language Runtime). Working with a virtualized runtime gives the additional benefit of allowing the use of heterogeneous hardware.

To facilitate offloading we use DSM techniques to keep the heap, stacks, and locking states consistent across endpoints. These techniques together form the basis of our distributed virtual machine, allowing the migration of any number of threads while being consistent with the Java Memory Model. Our DSM strategy can be considered both lazy and eager – lazy in the classic sense that our protocol acts on an acquire, and eager in the sense that we eagerly transmit all dirtied data when we do act. This strategy is useful for reducing the frequency of required communication between endpoints, necessary to deal with high latency between endpoints that often exists in wireless networks.

While it appears our design could be extended to more than two endpoints, we discuss how operations work specifically for the two endpoint case that our prototype supports. Our intended endpoints are one client (a phone) and one server (a high-performance machine). However, the design rarely needs to identify which endpoint is which and our work could be used in principle to combine two similarly powered devices.

## 3.1 Security

Under our design a malicious server can take arbitrary action on behalf of the client process being offloaded. Additionally we have no mechanism to ensure the accuracy of the computed data nor the privacy of the inputs provided. Therefore it seems necessary that the offload server be trusted by the client.

However there is no need for the server to trust the client. In this initial design the server only grants access of its computation resources to the client and it has no private data to compromise or dependency on the accuracy of computation.

## 3.2 Overview of the Java Memory Model

The Java Memory Model plays an important role in our design dictating what data must be observed by a thread. In the JMM, memory reads and writes are partially ordered by a transitive "happens-before" relationship. A read must observe a write if the write "happens-before" the read. The Java specification also supports threads and locks which directly tie into the JMM.

Within a single thread, all memory operations are totally ordered by which happened first during execution. Across threads, release consistency is used; when thread *A* acquires a lock previously held by thread *B*, a "happens-before" relationship is established between all operations in thread *B* up to the point the lock was released and for all future memory operations of thread *A*, meaning whatever writes thread *B* made before releasing the lock should be visible to thread *A*. Other miscellaneous operations like volatile memory accesses or starting a thread, can also create "happens-before" relationships between accesses in different threads.

Still there remain cases where there may be several writes that a read could observe. These situations are called data races and usually indicate a buggy program. This situation, however, is occasionally intentional. For example, let $W$ be the set of writes that "happen-before" the read. Then the VM may let the read observe any maximal element of $W$ or any write not ordered with respect to the read.

An existing method to follow the JMM while offloading is to offload one thread at a time and block all other local execution before the offloaded thread returns [9]. This prevents multiple threads from making progress simultaneously, reducing the benefit of offloading. This also eliminates the possibility of offloading computation (e.g., a function) that calls synchronization primitives unless it can be checked automatically that the code will not deadlock.

A slight modification is to let local threads continue executing, until they access shared state across threads [8]. This again limits the usefulness of offloading and, for example, prevents offloading a thread that may grab a lock later on. Additionally, this kind of scheme can introduce deadlocks into the program's execution if the offloaded thread tries to wait for data from another thread. COMET overcomes these limitations by relying on DSM and VM-synchronization techniques to keep the distributed virtual machine in a consistent state without limiting what can execute at any time.

## 3.3 Field-based DSM

Our key contribution over past DSM systems is the use of field level granularity to manage memory consistency. By doing things at this granularity, we can avoid tracking anything more than a single bit indicating the dirtiness of each field. Our DSM mechanism allows for multiple readers and writers to simultaneously access the same field without communication.

This is possible with DSM for Java, but not for other less-managed runtimes, because reads and writes can only happen at non-overlapping memory locations of known widths. In particular, this means we can always *merge* changes between two endpoints, even if they have both dirtied a field, by just selecting one of the writes to "happen-before" the other as described in §3.2. If instead we worked at a coarser granularity, it would be unclear how to merge two dirty memory regions. Even a copy of the original memory region does not allow us to merge writes without a view into the alignment of fields in memory. This is particularly important in Java where values must not appear "out of thin air."

## 3.4 VM-Synchronization

At the heart of our design is the directed VM-synchronization primitive between two endpoints; the pusher and the puller. In full, the primitive synchronizes the states of the virtual heap, stacks, bytecode sources, class initialization states, and synthetic classes.

Synchronizing the virtual heap is accomplished by tracking dirty fields of objects discussed in §3.3. During a VM-synchronization, the pusher sends over all of the dirty fields it has in the shared heap. The puller then reads in the changes and overwrites the fields sent by the pusher. Both endpoints mark the fields involved as clean.

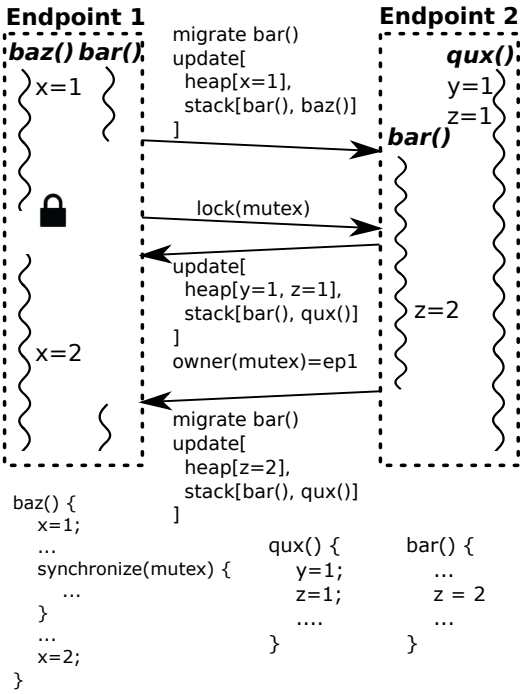



Figure 2: At the beginning, baz() and bar() are running in separate threads on ep1, while qux() is running as a different thread on ep2. ep2 holds the ownership of mutex at the beginning, but no thread lock on it.

Stack synchronization then is done by sending over the stacks of any shared threads that are running locally. This includes each method called, the program counter into each method, and any method level registers. This encodes enough information about the thread's execution state that the puller could resume execution. This makes the act of migrating a thread trivial after a VM-synchronization.

An example of the primitive's operation is shown in Figure 2. In the example, *ep1* pushes one VM-update (indicated by "update[...]" in the diagram) to *ep2*, and two VM updates go from *ep2* to *ep1*. The pusher operates by assembling a heap update from all changes to the heap since the last heap update and an update to each locally running stack. The puller then receives the update and merges the changes into its heap and updates the stacks of each received thread. Note that the pusher's heap and thread stacks do not change as a result of this process. For example in the third VM-synchronization shown in Figure 2, only the modification to $z$ needs to be sent because $x$ and $y$ have not changed since the previous update from *ep2* to *ep1*.

This primitive is our mechanism for establishing a "happens-before" relationship that spans operations on different endpoints. This includes each of the situations mentioned in §3.2. Any operations that need to establish a "happens-before" relationship, such as migrating a thread or acquiring a lock when it was last held elsewhere, will use this operation.

## 3.5 Locks

From §3.2 we found that a "happens-before" relationship needs to be established whenever a lock is acquired that was last held on another thread. To solve this problem we assign an owner to each lock, indicating the endpoint that last held the lock. In Java, any object can be used as a lock. Thus, each object is annotated with a lock-ownership flag. When attempting to lock an object that an endpoint does not own, the requesting thread can simply be migrated or the endpoint can make a request for ownership of the lock. Which choice should be followed is dictated by the scheduler discussed in §3.7.

In the latter case, the other endpoint will usually respond with a VM-update and a flag indicating ownership has been transferred. Figure 2 demonstrates this behavior when *baz()*, running on *ep1*, attempts to lock on the object *mutex* that is originally owned by *ep2*. This causes a VM-update to be sent to *ep1* as well as ownership of the *mutex* object.

Java also supports a form of condition variables that allow wait and signaling of objects. This comes almost for free because waiting on an object implies you hold a lock on the object. When you wait, the lock is released and execution is suspended until the object is signaled to continue. Then the lock is re-acquired, which will cause the appropriate synchronization, if required. COMET only needs to, in some situations, send a signal to wake up a waiting remote endpoint. Some additional tracking of how many threads are waiting on a given condition variable is maintained to serve this purpose.

Volatiles are handled in a similar fashion to locks. Only one endpoint can be allowed to perform memory operations on a volatile field at a time. This is stronger synchronization than what is required, but it suffices for our design. Fortunately volatile operations are fairly rare, especially in situations when offloading is desirable.

## 3.6 Native Functions

In §3.4 and §3.5, we have described the key parts of our offloading system. Still, we have to decide what computation to offload. The obvious question is why not offload everything? This works fine until a native function is encountered. These are functions usually implemented in C with bindings to be accessed from Java. COMET cannot, in general, offload these functions. These functions are usually implemented in native code for one of the following three reasons: (1) reliance on phone resources (file system, display), (2) performance, and (3) reliance on readily-available C libraries.

Each of these cases provides its own challenges if you wish to allow both endpoints to run the corresponding function. It is very common for these functions to rely on hidden native state, frequently making this a more difficult task than it might first appear. So, in general, we assume that native methods cannot be offloaded unless we manually mark them otherwise. As applications rarely include their own native libraries [12], we can do this once as a manual effort for the standard Android libraries. Indeed, we have manually evaluated around 200 native functions as being suitable to run on any endpoint.

Additionally, to support failure recovery we need to be even more careful about what native methods are allowed to run on the server. Methods that modify the Java heap or call back into Java code can be dangerous because if a VM-synchronization happens while the native method is executing the client has no way to reconstruct the state hidden on the native stack of the partially executed native method. Some native methods warrant a special exemption (e.g. Java's reflection library) . For other non-blocking native methods, we may simply force a pending VM-synchronization to wait for the method to exit before continuing.

## 3.7 $\tau$-Scheduling

The last component of the system is the scheduler. Analogous to schedulers used by modern operating systems, the scheduler is charged with the task of moving threads between endpoints in an attempt to maximize throughput (alternative goals are possible as well). During a push, the scheduler decides which local threads should be migrated and which non-essential lock/volatile ownership flags should be transferred. Additionally, the scheduler should initiate a push when it wants to migrate a thread or to avoid requesting ownership of a lock.

For our first iteration of a solution to this problem, we have used a basic scheduler that relies on past behavior to move threads from client to server where they remain as long as possible. This is achieved by tracking how long a thread has been running on the client without executing client-only code (a native method) and migrating the thread when this time exceeds $\tau$, where $\tau$ is some configurable parameter. For our prototype, we initially choose $\tau$ to be twice the round trip time (RTT). Over the execution of the application, $\tau$ is replaced with twice the average VM-synchronization time.

This choice of $\tau$ has the nice property of limiting

the damage of mistakes to twice the runtime without offloading. This is because we need to have made forward progress for a time of at least $\tau$ before migrating. If we immediately had to migrate back, we have selected $\tau$ so that it should take a time of $\tau$ before we are running locally again. Thus the $\tau$ parameter lets us tune risk and potential benefit of the scheduler.

# 4 Implementation

We built COMET by extending the Dalvik Virtual Machine (DalvikVM) targeted for the Android mobile operating system. The DalvikVM is an interpreter for the Dalvik byte-code based on the Java VM specification. Our code was written on top of the CyanogenMod's Gingerbread release. While the DalvikVM is intended to be run on Android, it can be compiled for x86 architectures and run on common Linux distributions. Because the JIT was not yet available on x86, we were forced to disable the JIT for our prototype. The additions to the code-base sum to approximately 5,000 lines of C code with the largest component, the one managing most DSM operation, at 900 lines of code.

## 4.1 Threads and Communication

To effectively handle multi-threaded environments, threads are virtualized across different endpoints. The DalvikVM uses kernel threads to run each Java thread. In our system, kernel threads are paired between endpoints to represent and act on behalf of a single virtualized thread. Figure 3 shows pairs of parallel threads that represent a single thread virtualized across endpoints.

To facilitate operations across endpoints, a full-duplex byte stream connects two parallel threads. This abstraction is useful because all of the communication can be expressed easily between parallel threads. Figure 3 shows the path that data travel to get from a thread to its corresponding parallel thread. When a thread has data to write, it sends a message to the controller. The controller then multiplexes the data being written by all threads over a single TCP connection, applying a compression filter to save bandwidth and time. The remote controller will demultiplex and decompress the message, sending the data to the kernel thread parallel to the sender. This allows for multiple byte-streams while avoiding the need to establish multiple connections between endpoints.

To allow messages to be demultiplexed, virtual threads are assigned IDs (this is the same identifier *Thread.getId()* will return). When the controller receives a message, it can use the ID to look up which thread to deliver the message to or create the thread if it does not already exist. When assigning IDs we set the high bit of the ID with the

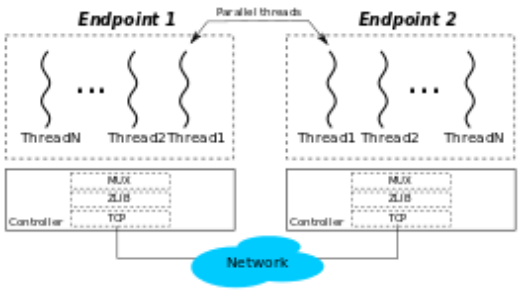



Figure 3: Communication between endpoints

endpoint ID that created the thread to avoid ID collisions. When a thread exits, it sends a message to the parallel thread so that the thread may exit on both endpoints (and joins so the thread can complete) and the IDs may be released.

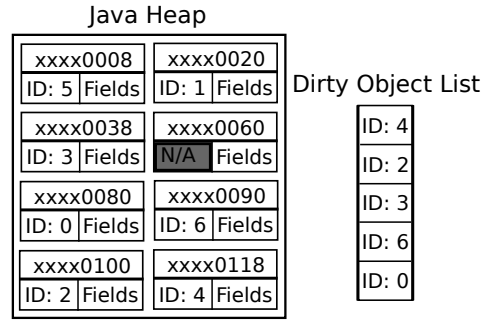
## 4.2 Tracked Set

In §3.4, we mentioned that updates only need to be sent for objects that can be accessed by both endpoints. To identify such objects, we introduce the the notion of the tracked set of objects.

The tracked set contains all class objects and is occasionally updated to include other local objects, for example objects present on a stack during a VM-synchronization. Global fields are considered to be part of the class object they are defined in, and thus are included in the tracked set. Additionally during a push operation, the tracked set is closed, meaning that all objects reachable from objects in the tracked set are added to the tracked set as well. See §4.5 for how tracked objects can eventually be garbage collected.

To support our DSM design, every tracked object is annotated with a bitset indicating which fields are dirty. Figure 4 illustrates how this data is stored and accessed. Each write to a field of a tracked object causes that field to be marked as dirty. Additionally, to be able to quickly find only the objects with dirty fields, a dirty object list is maintained, to which an object is added the first time one of its fields is made dirty. As a result, it becomes easy for a push operation to find all updated fields and add untracked objects that appear in those modified fields to the tracked set. After a push, all of the dirty bits are cleared and the dirty object list is emptied.

Similar to existing systems [9, 8], COMET assigns IDs to objects in the tracked set. This allows each endpoint to talk about shared objects in a coherent way, as pointers are meaningless across endpoints. For efficiency reasons, objects are assigned incremental IDs when added to the tracked set, so an object lookup turns into an array lookup into the tracked set table as shown in Figure 4. Similar




Tracked Set Table

| ID | Address | Lock owned | Volatile owned | Object Dirty | Field 0 dirty | Field 1 dirty | ••• |
|---|---|---|---|---|---|---|---|
| 0 | xxxx0080 | | X | X | X | X | |
| 1 | xxxx0020 | X | | | | | |
| 2 | xxxx0100 | X | X | X | X | | |
| 3 | xxxx0038 | X | X | X | | X | |
| 4 | xxxx0118 | | | X | | X | |
| 5 | xxxx0008 | X | X | | | | |
| 6 | xxxx0090 | X | X | X | X | X | |
| 7 | null | | | | | | |

Figure 4: Tracked Set Table

to thread ID assignment, the high order bit of object IDs is filled with the endpoint ID to avoid conflict so both endpoints can add objects to the tracked set. The overhead associated with tracking field writes and maintaining the tracked set structures is examined in §5.2.

## 4.3 VM-Synchronization

Now we are ready to explain the core of our design, the VM-synchronization operation. VM updates are always performed between parallel threads with one endpoint sending updates, the pusher, and the other endpoint receiving the updates, the puller. There are three major steps to each synchronization operation.

First, the pusher and puller enter into an executable exchange protocol where any new bytecode sources that have been loaded on the pusher are sent over to the puller. Usually, this step is just as simple as the pusher sending over a message indicating that there are no new binaries. Otherwise, the pusher sends over a unique binary identifier, so the puller can try to look up the binary in its cache. If it is not found, the puller can request the entire binary which will be stored it in its cache for future use.

Second, the pusher sends over information about each thread. To simplify this operation, all locally running threads are temporarily suspended. We can track what portion of the stack needs to be resent since the last update and send each frame higher up on the call stack. Each interpreted frame contains a method, a program counter, and the registers used by that method. Using register maps produced by the DalvikVM, we can detect which registers are objects and add them to the tracked set. In addition to the stack information, we also transmit information about which locks are held by each thread. This enables the thread on the puller to acquire any activated locks in addition to allowing functions like *Thread.holdsLock()* to execute properly without any communication.

Finally, the pusher sends over an update of the shared heap. It goes through the dirty object list, finds which fields have changed, and sends the actual modifications to those fields. If we attempt to transmit an object field, the referenced object is added to the tracked set and the ID of that object is transmitted. Otherwise an endian neutral encoding of the field is transmitted. Lastly, it clears all of the dirty flags and the dirty object list and resumes all other threads. Performance data on how long this takes and how much data needs to be sent is available in §5.3.2.

After the executable exchange, the puller first buffers the rest of the VM-synchronization. Then it temporarily suspends each of the local threads as done during the push. It then merges in the update to the heap first. This often involves creating new objects that have not yet been seen or could involve writing a locally dirty field. In the latter case the JMM gives us liberty to use either value in the field but we choose to overwrite with the new data so the dirty bit can be cleared. After this, the puller reads in the changes to the stack, again using the register maps from the DalvikVM to convert any object registers from their IDs to their corresponding object pointers. After this completes, the puller can again resume threads locally. Additionally, heap updates are annotated with a revision ID, so that they can be pulled in the same order they were pushed.

As an example, take Figure 4 as the initial state of the pusher. The heap update will contain five object definitions corresponding to the five dirty objects. If we assume that each of these objects only has the two shown fields, then in total seven field updates will be sent out. If one of those fields was an object field that pointed to `xxxx0060`, it would be assigned ID 7, and instead we would send out six object definitions and nine field updates. In either case, after the heap update, the pusher's dirty object list will be empty and all of the dirty flags are cleared.

## 4.4 Thread Migration

Next we can discuss the operations built on top of the VM-synchronization. In the core of the offload engine is the thread loop. At most instances in time, at least one of a pair of parallel threads is waiting in the thread loop for a message from its peer. These messages are used to initiate the actions described below. Additionally, a

special *resume* message can be sent to tell the thread to exit the message loop and continue on with its previous operation.

The primary operation is the *migrate* operation which indicates that the requesting thread wishes to migrate across endpoints. The VM-synchronization handles most of the challenges involved. In addition, an *activate* and *deactivate* operation is performed on the puller and pusher respectively. *Activate* is responsible for grabbing any locks and setting lock-ownership of objects that are currently held in the execution of the thread. After that, the thread either calls directly into an interpreter or falls back into the interpreter that called into the thread loop depending on where the thread now is in its execution. *Deactivate* is comparatively simpler and just needs to release any held locks and ownership over them.

Transferring lock ownership is another important operation. As discussed in §3.5 and shown in Figure 4, each object is annotated with a lock ownership flag so that a "happen-before" relationship can be established correctly when needed. Initially each object is owned by the endpoint that created it. When a thread attempts to lock an object its endpoint does not own, it needs to request ownership of the object from the remote endpoint (or alternatively the scheduler could decide to migrate the thread). It does so by sending a *lock* message to the other endpoint along with the object ID it wishes to gain ownership of. The parallel thread wakes up and responds either with a heap update if the endpoint still owns the object or a failure message if ownership has already been transferred (i.e., by another virtual thread). In case of failure, the initial thread will simply repeat the entire process of trying to lock the object again. Special care must be taken so that exactly one endpoint always owns the lock, with the small exception of when ownership is being transferred and nobody owns the lock.

Volatiles are handled in much the same way as locks. In addition to the lock ownership flag annotated to objects, there is also a volatile ownership flag. This flag mirrors the lock flag so that a "happens-before" relationship is established when an endpoint that does not hold volatile ownership of an object needs to access a volatile field of that object.

### 4.5 Garbage Collection

The tracked set as described so far will keep growing indefinitely. Occasionally, some of the shared state will no longer be needed by either endpoint and it can be removed. To resolve this issue we have a distributed garbage collection mechanism. This mechanism is triggered after a normal garbage collection has failed to free enough memory and the VM is about to signal it is out of memory.

To begin distributed garbage collection, both endpoints *mark* every object that is reachable locally. Then a bitvector indicating whether each tracked object is locally reachable is sent to the remote endpoint. Receiving this bitvector from the all remotely *marked* objects are *marked* locally in addition to any other objects reachable. If any new objects are marked this way on either endpoint the bitvectors must be sent again. This process continues until both endpoints have agreed on which tracked objects are reachable. In pathological cases this process could take quite a few round trips to converge. In these cases the client could just disconnect and initiate failure recovery.

### 4.6 Failure Recovery

From the design of COMET failure recovery comes almost for free. To properly implement failure recovery, however, the client must never enter a state that it could not recover from if no more data was received from the server. Therefore each operation that can be performed must either wait for all remote data to arrive before committing any permanent changes (e.g. data is buffered in the VM-synchronization) or the change must be reversible if the server is lost before the operation completes. To recover from a failure the client needs only resume all threads locally and reset the tracked object set.

In the case of resuming execution, the server is required with each synchronization to send an update of all of the thread stacks. This way if the server is lost it has the necessary stack information to resume execution. The client, however, needs only to send stacks of threads it is attempting to migrate. Detection of server loss at this point is simple. If the connection to the server is closed or if the server has not responded to a heartbeat soon enough.

## 5 Evaluation

In this section we evaluate the overheads of COMET (§5.2), and the performance and energy improvements that COMET enables for a set of applications available on Google Play as well as one hand-made computation-intensive benchmark (§5.3). In §5.4 we take a deeper look at how COMET works in some unique situations as a series of short case studies.

### 5.1 Methodology

We tested COMET on a Samsung Captivate smartphone running the Android 2.3.4 (Gingerbread) operating system. Because the phone has some proprietary drivers, we implemented COMET within a CyanogenMod [10] build of the Android operating system. Our server is a 3.16GHz, eight core (two quad core Intel Xeon X5460 processors) system with 16GB of RAM, running Ubuntu

| $\mu$Benchmark | T & S | T & !S | !T & S | !T & !S |
|---|---|---|---|---|
| write-array | 30.9% | 30.4% | 8.96% | 7.58% |
| write-field | 36.7% | 35.1% | 10.6% | 9.17% |
| read-array | 8.17% | 6.05% | 9.71% | 5.75% |
| func-read | 12.2% | 9.26% | 9.76% | 9.58% |

Table 2: Overheads of COMET relative to an uninstrumented client. Results are shown for when heap tracking(T) and the scheduler(S) are enabled/disabled. An exclamation mark indicates that the specified function is disabled.

11.10. COMET was tested using 802.11g WiFi in the Computer Science and Engineering building at the University of Michigan, as well as a real 3G connection using AT&T's cellular network. To gather energy data, we used a Monsoon Power Meter [24], which samples the battery voltage and current every 200$\mu$s to generate a power waveform. We integrated these power waveforms over time to collect our energy readings.

We evaluated COMET using nine real applications from Google Play that are diverse in their functionality, but all have non-trivial computation that may benefit from offloading. Table 1 lists the package names and their functionality. Additionally, two purely computational applications were chosen, Linpack (available from Google Play) and Factor (hand-coded), to highlight the capabilities of COMET. Factor features multi-threaded execution to illustrate COMET's capability to offload multiple threads simultaneously, which will be discussed in detail in §5.4.

In order to create repeatable tests, we used the Robotium 3.1 user scenario testing framework [28]. This framework allowed us to script user input events and reliably gather timing information for all of our benchmarks. All test data has been gathered from five test runs of each benchmark for each network configuration.

## 5.2 Microbenchmarks

We test the overheads of our COMET prototype with four microbenchmarks: *Write-array* that writes to an array in one linear pass, *Write-field* that writes to one index in an array, *Read-array* that reads an entire array in one linear pass, and *Func-read* that performs array reads through many small function calls.

Table 2 displays the results of these microbenchmarks. All results are relative to a client running an uninstrumented Dalvik VM using its portable interpreter. Results are shown for all combinations of when heap tracking or the scheduler are enabled or disabled.

When performing heavy writes to an object the write tracking code is triggered and causes performance degradation, which is shown by the *write-array* and *write-field* tests when tracking is enabled. There is never any tracking of reads so the overhead in *read-array* and *func-read* gives the overheads from modifications to the interpreter in those cases.

While these overheads seem significantly high, these microbenchmarks represent worst case scenarios that are unlikely to appear when running real applications. The high costs can be offset by benefits from computation speed-ups and may be reducible with more work.

## 5.3 Macrobenchmarks

This section discusses high level tests of the applications listed in Table 1. We have divided each application's computation into "UI" and "Computation" components based on which portions of the applications we believe are doing interesting computation. The input events used to operate our tests do not come from actual user traces and in some cases include additional delays to ensure synchronization. Therefore the ratio of "UI" to "Computation" time means very little but we still present the "UI" times to give some indication of how these applications might be used and because the "UI" portion of execution is difficult to discount in our energy measurements.

Moreover our application suite and the functionality exercised in our tests were not chosen based on any real user data. Therefore these tests serve only as an indication that our system can work on some applications in the wild. A user study is required to get some metric on how well COMET can do in the wild when the system has matured.

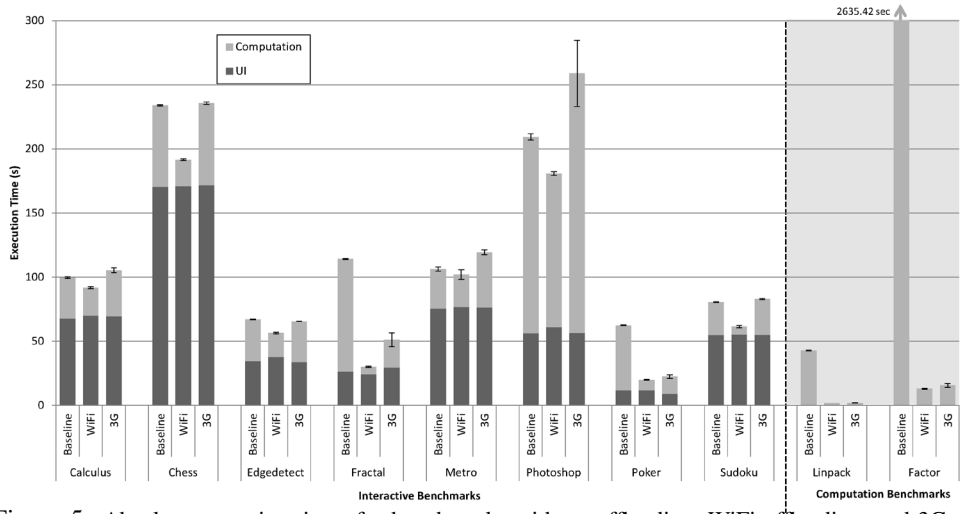
### 5.3.1 Performance and Energy Benefits

The primary goal of COMET is to improve the speed of computation. As a side-effect, we also expect to see improvements in energy-efficiency if computation-intensive portions of code are executed remotely on a server. Because there is no standard set of benchmarks in this area of research and readers may be unfamiliar with the applications chosen as benchmarks, we present our performance results as absolute times in seconds and energies in Joules in Figures 5 and 6. Each figure also shows the computation speed-ups and energy efficiency improvements relative to the same benchmarks run without offloading enabled.

Figure 5 shows that COMET achieves significant computation speed-ups for most benchmarks when offloading using WiFi. The geometric mean of computation speed-up that is observed for offloading interactive applications over WiFi is 2.88X. The high latency and poor bandwidth of 3G made it much less successful than WiFi however. Most of our benchmarks did not see performance improvements with the exception of Fractal, Poker, Linpack, and Factor. In the other benchmarks the scheduler found it too costly to offload so only small performance impacts

| | Benchmark | Package Name | Description |
|---|---|---|---|
| Interactive Benchmarks | Calculus | com.andymc.derivative | Math tool, computes discrete integrals using Riemann sums |
| | Chess | com.alonsoruibal.chessdroid.lite | Chess game, does BFS to find best moves |
| | Edgedetect | com.lmorda.DicomStudio | Image filter, detects and blurs edges |
| | Fractal | com.wimolife.Fractal | Math tool, zooms and re-renders Mandelbrot fractals |
| | Metro | com.mechsoft.ru.metro | Trip planner, Finds route between subway stations using Dijkstra's Algorithm |
| | Photoshop | com.adobe.psmobile | Image editor, performs cropping, filtering, and image effects |
| | Poker | com.leslie.cjpokeroddscalculator | Game aid, uses Monte Carlo simulation to find odds of winning a poker game |
| | Sudoku | de.georgwiese.sudokusolver | Game aid, solves sudoku puzzles given known values |
| Computation Benchmarks | Linpack | com.greenecomputing.linpack | Computation benchmark, standard linear algebra benchmark suite testing floating point computation |
| | Factor | Not available on Google Play | Computation benchmark, hand written application that uses multiple threads to factors large numbers |

Table 1: Description of benchmarks used in this evaluation




| Interactive Benchmarks | WiFi | 3G |
|---|---|---|
| Calculus | 1.47x | 0.89x |
| Chess | 3.06x | 0.99x |
| Edgedetect | 1.75x | 1.03x |
| Fractal | 15.01x | 4.05x |
| Metro | 1.22x | 0.72x |
| Photoshop | 1.28x | 0.76x |
| Poker | 6.15x | 3.87x |
| Sudoku | 4.18x | 0.92x |
| GEOMEAN | 2.88x | 1.28x |
| **Computation Benchmarks** | **WiFi** | **3G** |
| Linpack | 21.38x | 20.94x |
| Factor | 201.68x | 168.34x |
| GEOMEAN | 65.66x | 59.37x |

Figure 5: Absolute execution times for benchmarks with no offloading, WiFi offloading, and 3G offloading. Times are broken down to reflect portions of time that the benchmarks are doing something computationally interesting versus navigating the UI. Whiskers show the standard deviation. Computation speed-up figures relative to not offloading are shown on the right.

were seen. On average, offloading using 3G allowed a 1.28X speed-up for our interactive applications.

As a back-of-the-envelope calculation, we can estimate how many WiFi clients running applications from our benchmark an eight core server could handle by computing the average CPU utilization on the server for each application. This comes out to 28% percent utilization which suggests that a COMET server could sustain about 28 active clients.

As a consequence of reducing the absolute runtime of the application and the amount of heavy computation done on the client, energy improvements are observed in all but one case when offloading using WiFi. Figure 6 details the energy costs of each test. The short latency and high bandwidth of WiFi allows a client to spend less energy transmitting and receiving data when transferring state and control over to a server. Thanks to this, COMET was able to improve energy efficiency on average by 1.51X for interactive applications. Again due to 3G's network characteristics and higher energy costs we found that offloading with 3G usually consumed more energy than it saved. Fractal was the only interactive benchmark that saw significant energy improvement when using 3G.

### 5.3.2 The Amount of Transferred State

We now examine how much data COMET's VM-synchronization primitive transfers when our benchmarks offload execution using WiFi and 3G. Table 3 shows the amount of state transferred downstream and upstream, in KB, from the client. This data is averaged over the same five runs used to gather our performance and energy data.

Because all but one of our benchmarks were not designed by us, the authors of these applications likely did not intend for their code to ship state between a client and server. Furthermore, we made no effort or modifications

| | | Download (KB) | | Upload (KB) | |
|---|---|---|---|---|---|
| Benchmark | Init. (KB) | WiFi | 3G | WiFi | 3G |
| Calculus | 765.7 | 68.7 | 8.3 | 361.0 | 138.3 |
| Chess | 849.7 | 2777.0 | 0 | 860.9 | 0 |
| Edgedetect | 774.3 | 3818.2 | 0 | 641.6 | 0 |
| Fractal | 817.9 | 619.3 | 518.2 | 222.8 | 249.5 |
| Metro | 805.7 | 13.8 | 0 | 635.0 | 0.1 |
| Photoshop | 792.8 | 41757.0 | 587.0 | 25939.1 | 2309.8 |
| Poker | 773.5 | 2.6 | 1.1 | 185.6 | 138.0 |
| Sudoku | 876.4 | 292.7 | 0 | 300.0 | 0.3 |
| Linpack | 807.7 | 1873.6 | 1872.9 | 19.0 | 0.4 |
| Factor | 732.6 | 13.0 | 11.2 | 20.6 | 7.3 |
| GEOMEAN | 798.6 | 294.3 | 20.0 | 331.4 | 13.8 |

Table 3: Total number of bytes transmitted when offloading. Init refers to the initial heap synchronization.

to the binaries of these applications to optimize how state is packaged for offloading. Thus the size of state transfers may seem large when compared to figures presented in the literature [9]. Mixed DSM strategies will be used to reduce this cost in future work.

The first thing that occurs when COMET begins to offload an application's execution to a server is an initial heap synchronization including all of the globally reachable state. This first sync includes a great amount of data that will never change later in the execution and is thus considerably larger than future updates typically. This initial heap synchronization is typically between 750–810KB, regardless of the means of connectivity. This takes, on average, 1.69s for a WiFi connection and 6.39s over 3G to complete. As a point of comparison, the average push and pull operations between two endpoints are only 5.77ms over WiFi and 111ms over 3G.

COMET is capable of adapting to the available network conditions. When a low latency connection is available COMET will more eagerly use this connection to exploit more offloading opportunities. Figures 5 and 6 show that for the Photoshop test, even though over 60MB of state needs to be transferred, there can still be performance gains and energy savings. Conversely when network latency is high and bandwidth is limited, as is the case when operating using 3G connectivity, COMET determines that offloading computation is not advised and scales back its decisions to offload. Table 3 reflects this observation, as the KB of data communicated downstream and upstream when using 3G is significantly less than that of WiFi.

While the state transmissions observed in Table 3 may seem large, future technologies may make it less costly, in terms of time, energy, or both, to transmit data and execute remotely than to perform computation locally. Cellular carriers are currently adopting 4G LTE, which provides higher bandwidths and lower latencies than 3G [16]. This trend in cellular technology promises that in the coming years it will be practical to transmit the amount of data necessary to keep state synchronized when exploiting more offloading opportunities.

```
Loop: for (;;) {
  int op = iCode[frame.pc++];
  ...
  switch(op) {
    ...
    case Token.ADD :
      --stackTop;
      do_add(stack, sDbl, stackTop, cx);
      continue Loop;
    case Token.CALL : {
      ...
    }
  }
}
```

Figure 7: Excerpt from Rhino's Interpreter.java. Method-granularity offloading is going to have difficulties offloading code of this style.

## 5.4 Case Studies

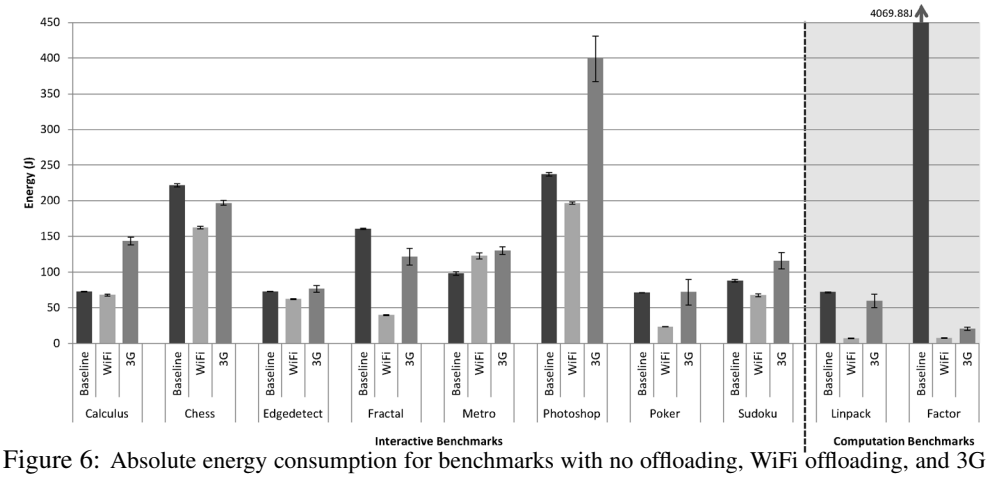
We now examine specific cases where COMET's design allows it to offload where other systems may not. Case study I focuses on the benefits of offloading at a fine granularity while case study II looks at the benefits of being able to offload multiple threads.

**Case Study I: Offloading JavaScript**

A natural question of this system is if it can work with derived runtimes. In particular, because of JavaScript's prevalence on the web, it would be interesting if JavaScript could be offloaded as well. Unfortunately, there are no existing web browsers that use a JavaScript engine written in Java. The nearest thing appears to be HtmlUnit [1], a "GUI-Less browser for Java programs" aimed at allowing automated web testing within Java. The JavaScript runtime backing HtmlUnit is called Rhino [2].

To test COMET's ability to offload JavaScript with Rhino, we ran the SunSpider [3] test suite with and without offloading. For the 18 benchmarks that could run correctly (some tests exceeded the Dalvik VM's stack size), we found that the entire test executed 6.6X faster with a maximum speed-up for a single test of 8.5X and a minimum speed-up of 4.7X. This suggests that COMET can effectively be applied to derived runtimes.

It is important to mention that method-granularity offloading would fall short in this scenario. Figure 7 gives an excerpt from Rhino's interpreter. The interpreter method cannot be offloaded as a whole as any subsequent client side only calls, such as accesses to the UI, would need to make an RPC right back to the client which could be quite expensive if there are even a few such calls. However, if we do not offload the interpreter loop, there is likely no other method that has a substantial running time.




| Interactive Benchmarks | WiFi | 3G |
|---|---|---|
| Calculus | 1.07x | 0.51x |
| Chess | 1.36x | 1.13x |
| Edgedetect | 1.17x | 0.95x |
| Fractal | 4.02x | 1.32x |
| Metro | 0.80x | 0.76x |
| Photoshop | 1.21x | 0.59x |
| Poker | 3.05x | 0.99x |
| Sudoku | 1.30x | 0.76x |
| GEOMEAN | 1.51x | 0.84x |
| Computation Benchmarks | WiFi | 3G |
| Linpack | 10.39x | 1.20x |
| Factor | 517.50x | 185.24x |
| GEOMEAN | 73.32x | 14.93x |

Figure 6: Absolute energy consumption for benchmarks with no offloading, WiFi offloading, and 3G offloading. Whiskers show standard deviation. Energy improvements relative to not offloading are displayed on the right.

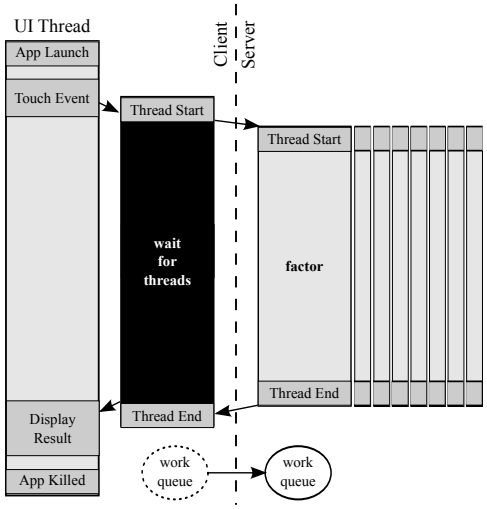



Figure 8: An illustration of multi-threaded execution being offloaded to a server for the Factor application. Time progresses from the top to the bottom of the diagram.

*do_add* for example is typically going to be fairly quick to execute. Therefore, offloading at a finer granularity than methods is necessary to offload with Rhino and programs like it. This feature is something that COMET offers over past offloading systems.

**Case Study II: Multi-threading in Factor**

The multi-threaded Factor benchmark shows the best case performance of COMET. It works by populating a work queue with some integers to be factored, and starting eight threads to process items from the queue. This also demonstrates that COMET can work well even when some amount of synchronization is required between threads. Figure 8 illustrates the offloading of the multi-threaded Factor benchmark.

The speed-up obtained by WiFi and 3G are 202X and 168X respectively. This is the difference between 44 minutes running locally and 13 seconds over WiFi demonstrating massive speed-ups using multiple cores. Other works cannot offload more than one thread at a time, especially when there are shared data accesses occurring [9, 8], and therefore can only get a speed-up of around 28.9X.

# 6 Limitations

Broadly, there are two important limitations of our work. First, COMET may decide to send over data that is not needed for computation. This is often wasteful of bandwidth and can make offloading opportunities more sparse. Two approaches that may help mitigate this challenge are using multiple DSM strategies like what is done in Munin [7] or applying static analysis to detect when data need not be sent.

The second limitation lies in the kinds of computation demanded by smartphones. We have not found a large number of existing applications that rely heavily on computation. Those that do frequently either implement offloading logic right into the application or write the computationally intensive parts of the application in C making it difficult to test with COMET. However tools like COMET can allow new kinds of applications to exist.

# 7 Conclusion and Future Work

In this paper, we have introduced COMET, a distributed runtime environment aimed at offloading from smartphones. We introduced a new DSM technique and VM-synchronization operation to keep endpoints in a consistent state according to the memory model of our runtime. This makes all virtualized code offloadable and allows multiple threads to be offloaded simultaneously. We demonstrated this system on nine real applications

and showed an average speed-up of 2.88X. In one hand-written application, we were able to reach as much as 202X speedup. To broaden the impact of our work, we plan on making the COMET system available upon publication.

Moving forward, the most promising line of work is in improving the scheduling algorithm used by COMET. The $\tau$-Scheduler described here is the simplest reasonable scheduler that we could come up with and the focus of this work lies elsewhere.

# 8 Acknowledgements

We thank Tim Harris for his time and energy of serving as the shepherd for this paper and providing numerous constructive comments and detailed feedback for improving the quality of this work. We also thank the anonymous reviewers for their feedback and helpful suggestions. This research was supported by the National Science Foundation under grants CNS-1050157, CNS-1039657, CNS-1059372 and CNS-0964478.

# References

[1] A Java GUI-Less Browser. `http://htmlunit.sourceforge.net/`.

[2] Rhino : JavaScript for Java. `http://www.mozilla.org/rhino/`.

[3] SunSpider JavaScript Benchmark. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`.

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proc. Int. Conf. Parallel Processing*, 1999.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proc. ACM Symp. Operating Systems Principles*, 1993.

[6] L. Cardelli. Obliq - A language with distributed scope. In *Proc. of the Symposium on Principles of Programming Langauges*, 1995.

[7] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computation*, 1995.

[8] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patt. CloneCloud : Elastic Execution between Mobile Device and Cloud. In *Proceedings of the European Conference on Computer Systems*, 2011.

[9] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Ch, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proc. MOBISYS*, 2010.

[10] CyanogenMod Community. CyanogenMod Android community rom based on gingerbread. `http://www.cyanogenmod.com/`.

[11] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *Proc. Int. Symp. Operating Systems Design and Implementation*. USENIX Association, 2004.

[12] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI*, 2010.

[13] A. Gember, C. Dragga, and A. Akella. ECOS: Practical Mobile Application Offloading for Enterprises. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, 2012.

[14] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the TCL/TK workshop*, 1996.

[15] X. Gu, K. Nahrstedt, A. Messer, D. Milojicic, I. Greenberg, I. Greenberg, and HP Laboratories. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proc. of IEEE International Conference on Pervasive Computing and Communications*, 2003.

[16] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, 2012.

[17] A. D. Joseph, A. E. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. ACM Symp. Operating Systems Principles*, 1995.

[18] A. Judge, P. Nixon, V. Cahill, B. Tangney, and S. Weber. Overview of distributed shared memory. Technical report, Trinity College Dublin, 1998.

[19] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 1988.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the international symposium on Computer architecture*, 1992.

[21] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: A computation offloading framework for smartphones. In *MOBICASE*, 2010.

[22] R. McIlroy and J. Sventek. Hera-JVM: A runtime system for heterogeneous multi-core architectures. In *Proceedings of the ACM international conference on object oriented programming systems languages and applications*, OOPSLA, 2010.

[23] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *ICDCS*, 2002.

[24] Monsoon Solutions Inc. Monsoon power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[25] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. ACM Symp. Operating Systems Principles*, 2009.

[26] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proceedings of Mobisys*, 2011.

[27] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. SociableSense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of MobiCom*, 2011.

[28] R. Reda et al. Robotium. `code.google.com/p/robotium`.

[29] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. *SIGOPS*, 1997.

[30] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *Proceedings of the IEEE international conference on cluster computing*, 2002.

# AppInsight: Mobile App Performance Monitoring in the Wild

*Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan,*
*Ian Obermiller, Shahin Shayandeh*
*Microsoft Research*

**Abstract—** The mobile-app marketplace is highly competitive. To maintain and improve the quality of their apps, developers need data about how their app is performing in the wild. The asynchronous, multi-threaded nature of mobile apps makes tracing difficult. The difficulties are compounded by the resource limitations inherent in the mobile platform. To address this challenge, we develop AppInsight, a system that instruments mobile-app binaries to automatically identify the critical path in user transactions, across asynchronous-call boundaries. AppInsight is lightweight, it does not require any input from the developer, and it does not require any changes to the OS. We used AppInsight to instrument 30 marketplace apps, and carried out a field trial with 30 users for over 4 months. We report on the characteristics of the critical paths that AppInsight found in this data. We also give real-world examples of how AppInsight helped developers improve the quality of their app.

## 1 INTRODUCTION

There are over a million mobile apps in various app marketplaces. Users rely on these apps for a variety of tasks, from posting mildly amusing comments on Facebook to online banking.

To improve the quality of their apps, developers must understand how the apps perform in the wild. Lab testing is important, but is seldom sufficient. Mobile apps are highly interactive, and a full range of user interactions are difficult to simulate in a lab. Also, mobile apps experience a wide variety of environmental conditions in the wild. Network connectivity (Wi-Fi or 3G), GPS-signal quality, and phone hardware all vary widely. Some platform APIs even change their behavior depending on the battery level. These diverse conditions are difficult to reproduce in a lab. Thus, collection of diagnostic and performance trace data from the field is essential.

Today, there is little platform support for tracing app performance in the field. Major mobile platforms, including iOS, Android, and Windows Phone, report app-crash logs to developers, but it is often difficult to identify the causes of crashes from these logs [1], and this data does not help diagnose performance problems. Analytics frameworks such as Flurry [8], and Preemptive [16] are designed to collect usage analytics (e.g., user demographics), rather than performance data. Thus, the only option left is for the developer to include custom tracing code in the app. However, writing such code is no easy task. Mobile apps are highly interactive. To keep the UI responsive, developers must use an asynchronous programming model with multiple threads to handle I/O and processing. Even a simple user request triggers multiple asynchronous calls, with complex synchronization between threads. Identifying performance bottlenecks in such code requires correctly tracking causality across asynchronous boundaries. This challenging task is made even more difficult because tracing overhead must be minimized to avoid impact on app performance, and also to limit the consumption of scarce resources such as battery and network bandwidth.

In this paper, we describe a system called *AppInsight* to help the app developers diagnose performance bottlenecks and failures experienced by their apps in the wild. AppInsight provides the developers with information on the *critical path* through their code for every *user transaction*. This information points the developer to the optimizations needed for improving user experience.

AppInsight instruments mobile apps mainly by interposing on event handlers. The performance data collected in the field is uploaded to a central server for offline analysis. The design of AppInsight was guided by three principles. $(i)$ **Low overhead:** We carefully select which code points to instrument to minimize overhead. $(ii)$ **Zero-effort:** We do not require app developers to write additional code, or add code annotations. Instrumentation is done by automatically rewriting app binaries. $(iii)$ **Immediately deployable:** We do not require changes to mobile OS or runtime.

We have implemented AppInsight for the Windows Phone platform. To evaluate AppInsight, we instrumented 30 popular apps and recruited 30 users to use these apps on their personal phones for over 4 months. This deployment yielded trace data for 6,752 app sessions, totaling over 33,000 minutes of usage time. Our evaluation shows that AppInsight is lightweight – on average, it increases the run time by 0.021%, and the worst-case overhead is less than 0.5%. Despite the low overhead, the instrumentation is comprehensive enough to allow us to make several detailed observations about app performance in the wild. For example, we can automatically highlight the critical paths for the longest user transactions. We can also group similar user transactions together and correlate variability in their performance with

```
void btnFetch_Click(
    object obj, RoutedEventArgs e) {
  var req = WebRequest.Create(url);
  req.BeginGetResponse(reqCallback, null);
}
void reqCallback(IAsyncResult result) {
  /* Process */
  UIDispatcher.BeginInvoke(updateUI);
}
void updateUI() {
  /* Update UI */
}
```

Figure 1: Example of asynchronous coding pattern

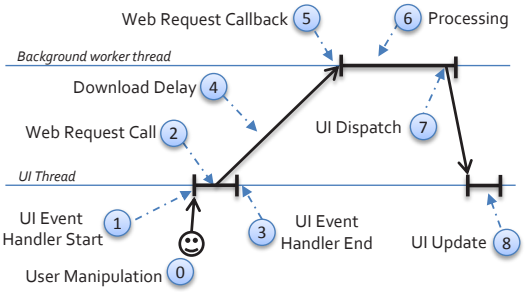



Figure 2: Execution trace for the code in Figure 1.

variation in the environment. In § 8.2, we will discuss how this feedback helped developers improve the quality of their app.

This paper makes two main contributions. First, we describe several innovative techniques that automatically instrument mobile apps to monitor user transactions with minimal overhead. These techniques are embodied in the current implementation of AppInsight. Second, we present results from a real-world study of 30 Windows Phone apps that we instrumented using AppInsight.

## 2 Mobile App Monitoring

We now discuss the typical asynchronous programming pattern used in mobile apps, and the challenge it presents for monitoring performance and failures.

Mobile apps are UI-centric in nature. In modern UI programming frameworks [6, 15], the UI is managed by a single, dedicated thread. All UI updates, and all user interactions with the UI take place on this thread. To maintain UI responsiveness, applications avoid blocking the UI thread as much as possible, and perform most work asynchronously. Some mobile-programming frameworks like Silverlight [15], do not even provide synchronous APIs for time-consuming operations like network I/O and location queries. Even compute tasks are typically carried out by spawning worker threads. Thus, user requests are processed in a highly asynchronous manner.

This is illustrated in Figure 2, which shows the execution trace for a simple code snippet in Figure 1. In the figure, horizontal line segments indicate time spent in thread execution, while arrows between line segments indicate causal relationships between threads.

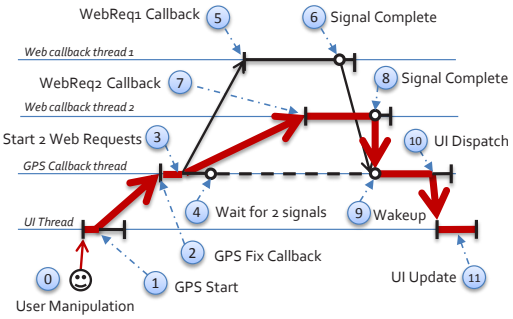



Figure 3: Execution trace of a location-based app.

(0) The user starts the transaction by clicking a button; (1) the OS invokes the event handler (`btn-Fetch_Click`) in the context of the UI thread; (2) the handler makes an asynchronous HTTP request, providing `reqCallback` as the callback; (3) the handler quits, freeing the UI thread; (4) time is spent downloading the HTTP content; (5) when the HTTP request completes, the OS calls `reqCallback` in a worker thread; (6) the worker thread processes the fetched data; (7) when the processing finishes, the worker thread invokes the UI Dispatcher, to queue a UI update; (8) the OS calls the dispatched function (`updateUI`) asynchronously on the UI thread, which updates the UI.

Real apps, of course, are much more complex. For example, (*i*) worker threads may in turn start their own worker threads, (*ii*) some user interactions may start a timer to perform periodic tasks through the lifetime of an app, (*iii*) transactions may be triggered by sensors such as accelerometers and, (*iv*) a user may interrupt a running transaction or start another one in parallel.

For example, Figure 3 illustrates a pattern common to location-based apps. The app displays information about nearby restaurants and attractions to the user. A typical user transaction goes as follows. Upon user manipulation, the app asks the system to get a GPS fix, and supplies a callback to invoke when the fix is obtained. The system obtains the fix, and invokes the app-supplied callback in a worker thread at (2). The callback function reads the GPS coordinates and makes two parallel web requests to fetch some location-specific data. Then, the thread waits (4), for two completion signals. The wait is indicated via a dotted line. As the two web requests complete, the OS invokes their callbacks at (5) and (7). The first callback signals completion to the blocked thread at (6), while the second one does it at (8). As a result of the second signal, the blocked thread wakes up at (9), and updates the UI via the dispatcher.

Given such complex behavior, it can be difficult for the developers to ascertain where the bottlenecks in the code are and what optimizations might improve user-perceived responsiveness. In Figure 3, the bottleneck path involves the second web request, which took longer to complete.

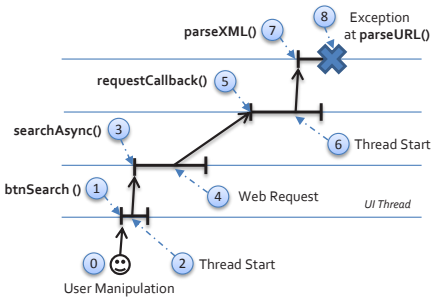



Figure 4: Execution trace of an app crash.

Worse, these bottlenecks may be different for different users, depending on their device, location, network conditions, and usage patterns.

Failure analysis is also complicated by the asynchronous nature of the app. Consider the example in Figure 4. Suppose the app crashes in the method `parseURL()` (8), which is called in a worker thread that started at `parseXML()` (7). Since the UI thread function that started the web request has exited, the OS has no information about the user context for this crash. Thus, in the exception log offered by today's popular mobile platforms, the developer will only see the stack trace of the crashed thread, from `parseURL()` to `parseXML()`. The developer however, might want more information, such as the user manipulation that triggered the crash, to speed up debugging. This underscores the need for a system that can track user transactions across thread boundaries. This is one of the goals of AppInsight, as we discuss next.

## 3 Goals

Our goal is to help developers understand the performance bottlenecks and failures experienced by their apps in the wild. We do this by providing them with *critical paths* for *user transactions* and *exception paths* when apps fail during a transaction. We now define these terms.

**User transaction:** A user transaction begins with a user manipulation of the UI, and ends with completion of all synchronous and asynchronous tasks (threads) in the app that were triggered by the manipulation. For example, in Figure 2, the user transaction starts when the user manipulation occurs and ends when the `updateUI` method completes. A user transaction need not always end with a UI update. For example, a background task may continue processing past the UI update, without impacting user-perceived latency. The notion of user-perceived latency is captured in our definition of *critical path*, which we turn to next.

**Critical path:** The critical path is the bottleneck path in a user transaction, such that changing the length of any part of the critical path will change the user-perceived latency. Informally, the critical path starts with a user manipulation event, and ends with a UI update event. In Figure 2, the entire path from (0) to (8) constitutes the critical path of the transaction. The latency can be reduced either by reducing the download delay (4) or the processing delay (6). In Figure 3, the critical path is shown in bold. Note that activities related to the download and processing of the first web request are not on the critical path.

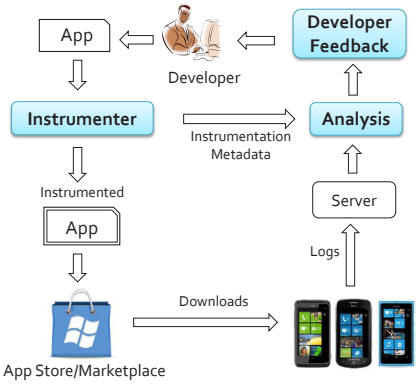



Figure 5: AppInsight System Overview

The critical path identifies the portions of the code that directly impacts user-perceived latency. However, the critical path may not always accurately characterize user experience. For example, a transaction may make multiple updates to the UI (one after the other), and the user may care about only one of them, or the user may interrupt a transaction to start a new one. We discuss this in § 6.2.

While the critical path is useful for understanding performance bottlenecks, to debug app failures, we provide the developer with *exception paths*.

**Exception path:** The *exception path* is the path from the user manipulation to the exception method, spanning asynchronous boundaries. In Figure 4, (0) to (8) is the exception path. The exception path points the developer to the user manipulation that started the asynchronous path leading to the crash.

We now describe how we collect the trace data needed to deliver the above information to the developer, while minimizing the impact on application performance.

## 4 AppInsight Design Overview

Figure 5 shows the architecture of AppInsight. The app binary is instrumented using an instrumentation tool (the instrumenter) that we provide. The developer only needs to provide the instrumenter with app binaries; no other input or source code annotation is needed.

The instrumenter leverages the fact that phone apps are often written using higher-level frameworks and compiled to an intermediate language (byte code). Our current implementation is designed for apps written using the Silverlight framework [15], compiled to MSIL [13] byte code. MSIL preserves the structure of the program, including types, methods and inheritance information.

Silverlight is used by a vast majority of the apps in the WP7 marketplace. AppInsight requires no special support from the Silverlight framework.

When users run the instrumented app, trace data is collected and uploaded to a server. We use the background transfer service (BTS) [18] to upload the trace data. BTS uploads the data when no foreground apps are running. It also provides a reliable transfer service in the face of network outages and losses. The trace data is analyzed and the findings are made available to the developer via a web-based interface (§ 7).

## 5 INSTRUMENTATION

We now describe our instrumenter in detail. Its goal is to capture, with minimal overhead, the information necessary to build execution traces of user transactions and identify their critical paths and exception paths.

A number of factors affect the performance of mobile applications: user input, environmental conditions, etc. Even the app-execution trace can be captured in varying degrees of detail. In deciding what to capture, we must strike the right balance between the overhead and our ability to give useful feedback to the developer.

Figures 3 and 4 indicate that, we need to capture six categories of data: (*i*) when the user manipulates the UI; (*ii*) when the app code executes on various threads (i.e., start and end of horizontal line segments); (*iii*) causality between asynchronous calls and callbacks; (*iv*) thread synchronization points (e.g., through Wait calls) and their causal relationship; (*v*) when the UI was updated; (*vi*) any unhandled exceptions. Apart from this, we also capture some additional data, as discussed in § 5.7.

To collect the data, we instrument the app in three steps. First, we read the app binary and assign a unique identifier to all methods in the app code and to system calls. Each call site is considered unique; if X is called twice, each call site gets its own identifier. This mapping is stored in a metadata file and uploaded to the analysis server for later use.

Second, we link two libraries to the app – a Detour library and a Logger library (see Figure 6). The Detour library is dynamically generated during instrumentation. It exports a series of detouring functions [11], which help attribute callback executions to the asynchronous calls that triggered them. The Logger library exports several logging functions and event handlers that insert trace records into a memory buffer. Each record is tagged with a timestamp and the id of the thread that called the logging function. The buffer is flushed to stable storage to prevent overflow as needed. When the app exits, the buffer is scheduled for upload using BTS.

Finally, we instrument the app binary with calls to methods in the Logger and Detour libraries from appropriate places to collect the data we need. Below, we describe this process in detail. We use the code fragment shown in Figure 1, and the corresponding transaction diagram in Figure 2 as a running example.

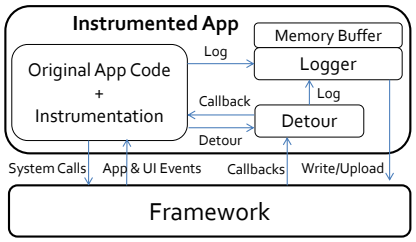



Figure 6: Structure of Instrumented App

### 5.1 Capturing UI Manipulation events

When the user interacts with the UI (touch, flick, etc.) the Silverlight framework delivers several UI input events on the UI thread of the app running in the foreground. The first event in this series is always a ManipulationStarted event, and the last is always the ManipulationEnded event. Further, any app-specified handler to handle the UI event is also called on the UI thread in between these two events. For example, in Figure 1, `btnFetch_Click` handles the click event for a button. When the user touches the button on the screen, the handler is called in between the two Manipulation events.

The logger library exports handlers for ManipulationStarted and ManipulationEnded events, which we add to the app's code. The handlers log the times of the events, which allows us to match the UI manipulation to the right app handler for that UI input.

### 5.2 Capturing thread execution

The horizontal line segments in Figure 2 indicate when the app code starts and ends executing on each thread. This can be determined from a full execution trace that logs the start and end of every method. However, the overhead of capturing and uploading a full execution trace from a mobile phone is prohibitive. We reduce the overhead substantially by observing that at the beginning of each horizontal line segment in Figure 2, the top frame in the thread's stack corresponds to an app method (as opposed to a method that is internal to the framework) and that this method is the only app method on the stack. These methods are *upcalls* from the framework into the app code. For our purpose, it is enough to log the start and end of only upcalls.

The upcalls are generated when the system invokes an app-specified handler (also called callback) methods for various reasons, for example, to handle user input, timer expiration, sensor triggers, or completion of I/O operations. Even spawning of worker threads involves upcalls: the app creates a thread, and specifies a method as a start method. This method is invoked as a callback of `Thread.Start` at some later time.

```
void btnFetch_Click(
    object obj, RoutedEventArgs e) {
  + Logger.LogUpcallStart(5);
  var req = WebRequest.Create(url);
  * Detour dt = DetourFactory.GetDetour(reqCallback, 7);
  * Logger.LogCallStart(7);
  req.BeginGetResponse(dt.Cb1, null);
  * Logger.LogCallEnd(7);
  + Logger.LogUpcallEnd(5);
}
void reqCallback(IAsyncResult result) {
  + Logger.LogUpcallStart(19);
  /* Process */
  * Detour dt = DetourFactory.GetDetour(updateUI, 13);
  * Logger.LogCallStart(13);
  UIDispatcher.BeginInvoke(dt.Cb2);
  * Logger.LogCallEnd(13);
  + AppInsight.LogUpcallEnd(19);
}
void updateUI() {
  + Logger.LogUpcallStart(21);
  /* Update UI */
  + Logger.LogUpcallEnd(21);
}
```

Figure 7: Instrumented version of the code in Figure 1. The actual instrumentation is done on MSIL byte code. We show decompiled C# code for convenience.

We identify all potential upcall methods using a simple heuristic. When a method is specified as a callback to a system call, a reference to it, a function pointer, called *delegate* in .NET parlance, is passed to the system call. For example, in Figure 1, a reference to `reqCallback` is passed to the `BeginGetResponse` system call. The MSIL code for creating a delegate has a fixed format [13], in which two special opcodes are used to push a function pointer onto the stack. Any method that is referenced by these opcodes may be called as an upcall[1].

We capture the start and end times of all potential upcalls, along with the ids assigned to them, as shown in Figure 7. The instrumentation added for tracking potential upcalls is prepended by '+'. All three methods in the example are potential upcalls and thus instrumented[2].

While this technique is guaranteed to capture all upcalls, it may instrument more methods than necessary, imposing unnecessary overhead. This overhead is negligible, compared to the savings achieved (§ 8.3).

### 5.3 Matching async calls to their callbacks

We described how we instrument all methods that may be used as upcalls. We now describe how we match asynchronous calls to the resulting upcalls (i.e., their callbacks). For example, in Figure 2, we need to match labels 2 and 5. To do so, we need to solve three problems.

First, we need to identify all call sites where an asynchronous system call was made, e.g., in Figure 1, the `BeginGetResponse` call is an asynchronous system call. Second, we need to log when the callback started executing as an upcall. We have already described how we track the start of upcall execution. Third, we need to connect the beginning of callback execution to the right asynchronous call.

```
public class DetourFactory {
  ...
  public static Detour GetDetour(
          Delegate d, int callId)  {
    int matchId = getUniqueId();
    Logger.LogAsyncStart(callId, matchId);
    return new Detour(d, matchId);
  }
}
public class Detour {
  int matchId; Delegate originalCb;
  public Detour(Delegate d, int matchId)  {
    this.originalCb = d; this.matchId = matchId;
  }
  public void Cb1(IAsyncResult result) {
    Logger.LogCallbackStart(this.matchId);
    Invoke(this.originalCb);
  }
  public void Cb2() {
    ...
  }
}
```

Figure 8: Detour library

We solve the first problem by assuming that any system call that accepts a delegate as an argument, is an asynchronous call. This simple heuristic needs some refinements in practice, which we will discuss in § 5.3.1.

The third problem of connecting the callback to the right asynchronous call is a challenging one. This is because a single callback function (e.g., a completion handler for a web request) may be specified as a callback for several asynchronous system calls. One possibility is to rewrite the app code to clone the callback function several times, and assign them unique ids. However, this is not sufficient, since the asynchronous call may be called in a loop (e.g., for each URL in a list, start download) and specify the same function as a callback. To handle such scenarios, we rewrite the callback methods to detour them through the Detour library, as described below.

Figure 7 shows instrumented code for the example in Figure 1. Instrumentation used for detour is tagged with '*'. Figure 8 shows relevant code inside the Detour library. We add instrumentation as follows.

(*i*) We identify the system call `BeginGetResponse` as an asynchronous call. The instrumenter has assigned a call id of 7 to this call site. We log the call site id, and the start and end time of the call[3].

(*ii*) We generate a new method called `cb1` that matches the signature of the supplied callback function, i.e., `reqCallback`, and add it to the `Detour` class in the Detour library. This method is responsible for invoking the original callback (see Figure 8).

(*iii*) We instrument the call site to call `GetDetour` to generate a new instance of the `Detour` object. This ob-

[1] Certain UI handlers are passed to the system differently. We identify them as well – we omit details due to lack of space.

[2] The method btn_FetchClick is a UI handler, and a pointer to it is passed to the system elsewhere.

[3] Async calls typically return almost immediately. We log both start and end of these calls not to collect timing data, but because such bracketing makes certain bookkeeping tasks easier.

```
Thread t = new Thread(foo);
...
...
t.Start();
```

Figure 9: Delayed callback

ject stores the original callback, and is assigned a unique id (called *matchId*) at runtime. This `matchId` helps match the asynchronous call to the callback.

(*iv*) We then rewrite the app code to replace the original callback argument with the newly generated detour method, `Detour.cb1`.

Notice from Figure 8 that the `GetDetour` method logs the beginning of an asynchronous call using the `LogAsyncStart` function of the Logger library. Similarly, the beginning of the callback is logged by the `LogCallbackStart`, which is called from `cb1`, just before the original callback is invoked. These records, and the `UpcallStart` record of the original callback method are linked by the `matchId`, the call site id, and their thread ids, allowing us to attribute the callback to the right asynchronous call. We show an example in § 5.8.

Figure 7 also shows another example of detouring. The `UpdateUI` method is a callback for the `BeginInvoke` method of the `UIDispatcher`, and hence is detoured.

#### *5.3.1 Refining async-call identification heuristic*

The simple heuristic used to determine which system calls are asynchronous calls, needs two refinements in practice. First, some system calls may invoke the supplied callback synchronously. This can be easily detected using thread ids in the trace. The second problem is more complex. Consider Figure 9. The callback delegate `foo` was specified when the constructor was called, but it is invoked only when `Thread.Start` is called, which may be much later. The simple heuristic would incorrectly match the callback to the call site of the constructor, instead of `Thread.Start`. We use domain knowledge about Silverlight system libraries to solve the problem. We know that the callback function is always invoked from `Thread.Start`. We log the id of the thread object at the constructor, and also at `Thread.Start`. The object ids, and the detour log described above allow us to match the callback to the `Thread.Start` call. We handle event subscriptions in a similar manner.

### 5.4 Capturing Thread Synchronization

Silverlight provides a set of methods for thread synchronization. The thread waits on a semaphore (e.g., Monitor.Wait(obj)), and is woken up by signaling that semaphore (e.g., Monitor.Pulse(obj)). We log calls to these functions and the identities of semaphore objects they use. These object ids can be used to determine the causal relationship between synchronization calls. Waiting on multiple objects, and thread join calls are handled similarly. Threads can also synchronize using shared variables. We will address this in § 9.

| RecordId | Records | ThreadId |
|---|---|---|
| 1 | UIManipulationStarted | 0 |
| 2 | MethodStart(5) | 0 |
| 3 | CallStart(7) | 0 |
| 4 | AsyncStart(7, 1) | 0 |
| 5 | CallEnd(7) | 0 |
| 6 | MethodEnd(5) | 0 |
| 7 | UIManipulationEnded | 0 |
| 8 | CallbackStart(1) | 1 |
| 9 | MethodStart(19) | 1 |
| 10 | CallStart(13) | 1 |
| 11 | AsyncStart(13, 2) | 1 |
| 12 | CallEnd(13) | 1 |
| 13 | MethodEnd(19) | 1 |
| 14 | CallbackStart(2) | 0 |
| 15 | MethodStart(21) | 0 |
| 16 | MethodEnd(21) | 0 |
| 17 | LayoutUpdated | 0 |

Table 1: Trace of code in Fig. 7. The UI thread id is 0.

### 5.5 Capturing UI updates

The Silverlight framework generates a special LayoutUpdated event whenever the app finishes updating the UI. Specifically, if an upcall runs on the UI thread (either event handlers, or app methods called via the UIDispatcher), and updates one or more elements of the UI as part of its execution, then a *single* LayoutUpdated event is raised when the upcall ends. The Logger library exports a handler for this event, which we add to the app code. The handler logs the time this event was raised.

### 5.6 Capturing unhandled exceptions

When an unhandled exception occurs in the app code, the system terminates the app. Before terminating, the system delivers a special event to the app. The data associated with this event contains the exception type and the stack trace of the thread in which the exception occurred. To log this data, the logger library exports a handler for this event, which we add to the app code.

### 5.7 Additional Information

For certain asynchronous calls such as web requests and GPS calls, we collect additional information both at the call and at the callback. For example, for web request calls, we log the URL and the network state. For GPS calls, we log the state of the GPS. The choice of the information we log is guided by our experience, and the inevitable tradeoff between completness and overhead. Our data shows that critical paths in a user transaction often involve either network or GPS accesses. By logging a small amount of additional information at certain points, we can give more meaningful feedback to the developer.

### 5.8 Example trace

Table 1 shows the trace generated by the instrumented code in Figure 7. Records 1 and 7 show a UI Manipulation event. They encompass an upcall (records 2-6) to the method `btnFetch_Click`. As described in § 5.1, we attribute this upcall to UI manipulation.

This method makes the asynchronous system call `BeginGetResponse` (record 4), the callback of which is

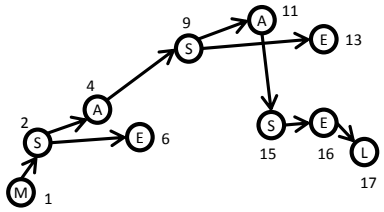



Figure 10: Transaction Graph for the trace in Table 1.

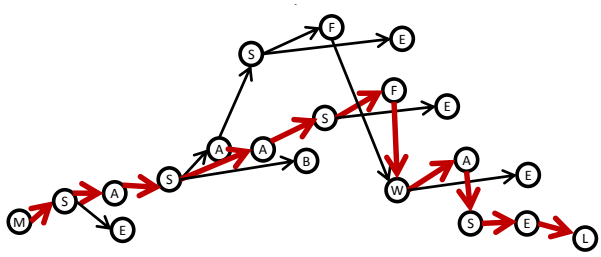



Figure 11: Transaction Graph for transaction in Figure 3. Record labels are omitted for simplicity.

detoured, and assigned a match id of 1. Record 8 marks the begining of the execution of the detoured callback. It calls the actual callback method, `reqCallback`, which has a method id of 19. This method executes between records 9 and 13. We can link records 8 and 9 because they have the same thread id, and will always follow each other (§ 5.3). When `reqCallback` executes, it makes another asynchronous call. This is the call to the UI dispatcher. We detour the callback, and assign it a match id of 2. The actual callback method, of course, is `UpdateUI`, which has the method id of 21.

The completion of this method is indicated by record 16. We note that this method ran on the UI thread. Record 17 indicates that a LayoutUpdated event was triggered immediately after the execution of this method, which means that this method must have updated the UI.

## 6 Analysis Methodology

We analyze the traces to delineate individual user transactions, and identify critical paths and exception paths. Transactions can also be analyzed in aggregate, to highlight broader trends.

### 6.1 User transactions

We represent user transactions by directed acyclic graphs. The graph is generated from the trace data. Consider the trace in Table 1. It is converted to the graph in Figure 10.

The graph contains five types of nodes, namely: $(M)$ User Manipulation, $(S)$ Upcall start, $(E)$ Upcall end, $(A)$ Async call start, and $(L)$ Layout updated. Each node represents one trace record [4] and is identified by the type and the record id. The mapping between node types $M, S, E, A$ and $L$ and the record types can be gleaned from Table 1.

[4] *CallStart*, *CallEnd* and *CallBackStart* records are used for bookkeeping purposes only, and are not mapped to nodes.

The edges between nodes represent causal relationships. For example, the UIManipulationStarted event $M1$ triggers the start of the handler $S2$. Similarly, the start of callback execution $S9$ was caused by the asynchronous call $A4$. We also say that an upcall start node "causes" any subsequent activity on that upcall. Hence we draw $S2 \rightarrow A4$, as the async call was made during execution of the upcall, and $S2 \rightarrow E6$, to represent the fact that the upcall end is triggered by upcall start.

The above graph does not show any thread synchronization events. These are represented by three types of nodes, namely: $(B)$ Thread blocked node, $(F)$ Semaphore fired node, and $(W)$ Thread wakeup node. We'll describe these nodes later.

When the app trace contains overlapping user transactions, this approach correctly separates them, and generates a graph for each.

We now discuss how we use this graphical representation to discover the critical path in a user transaction.

### 6.2 Critical Path

The critical path is the bottleneck path in the user transaction (§ 3). The basic algorithm for finding the critical path is simple. Consider Figure 10. We traverse the graph backwards, going from the last UI update $(L17)$, to the user manipulation event that signals the start of the transaction $(M1)$, traversing each directed edge in the opposite direction. This path [5], when reversed, yields the critical path: $M1$, $S2$, $A4$, $S9$, $A11$, $S15$, $E16$, $L17$. Even this simple example shows that we correctly account for time spent inside upcalls: for example, the edge $(S9, E13)$ is not on the critical path, which means that any activity in the `reqCallback` method (See Figure 7), after calling the dispatcher, does not affect user-perceived latency. This basic algorithm requires several refinements, as discussed below.

**Multiple UI Updates:** As discussed in § 3, the transaction may update the UI multiple times. This results in multiple $L$ nodes in the transaction graph. Only the developer can accurately determine which of these updates is important. In such cases, AppInsight, by default, reports the critical path to the last $L$ node. However, using the feedback interface (§ 7), the developer can ask AppInsight to generate the critical path to any of the $L$ nodes.

**Thread synchronization via signaling:** The basic algorithm implicitly assumes that each node will have only one edge incident upon it. This is not the case for the graph shown in Figure 11, which represents the transaction shown in Figure 3: Node $W$, which is a thread wakeup node, has two edges incident upon it, since the thread was waiting for two semaphores to fire (the two

[5] This algorithm always terminates because the transaction graph is always acyclic. Also, we are guaranteed to reach an $M$ node from an $L$ node, with backward traversal. We omit proofs.

*F* nodes). In such cases, we compare the timestamps of the semaphore-fire records, and pick the later event. This yields the critical path shown in the figure.

**Periodic timers:** An app may start a periodic timer, which fires at regular intervals and performs various tasks, including UI updates. In some cases, periodic timers can also be used for thread synchronization (§ 9). We detect this pattern, and then assume each timer firing to be the start of a separate transaction. We call these transactions *timer transactions*, to distinguish them from *user transactions*. These transactions need to be processed differently, since they may not end with UI updates. We omit details due to lack of space. We handle sensor-driven transactions in a similar manner.

### 6.3 Exception path

When the app crashes, we log the exception information including the stack trace of the thread that crashed (§ 5.6). We also have the AppInsight-generated trace until that point. We walk the stack frames until we find a frame that contains the method name of the last UpcallStart record in the AppInsight trace. The path from the start of the transaction to the Upcall start node, combined with the stack trace represents the exception path.

### 6.4 Aggregate Analysis

AppInsight helps the developer see the "big picture" by analyzing the transactions in aggregate. There are a number of ways to look at the aggregate data. Our experience shows that the developer benefits the most by using the aggregate data to uncover the root causes of performance variability, and to discover "outliers" – i.e. transactions that took abnormally long to complete compared to similar transactions.

To perform this analysis, we group together transactions with identical graphs; i.e. they have the same nodes and the same connectivity. These transactions represent the same user interaction with the app. This is a conservative grouping; the same user interaction may occasionally generate different transaction graphs, but if two transactions have the same graph, with a high probability they correspond to the same interaction.

**Understanding performance variance:** While the transactions in a group have the same transaction graph, their critical paths and durations can differ. To identify the major sources behind this variability, we use a standard statistical technique called Analysis of Variance (ANOVA). ANOVA quantifies the amount of variance in a measure that can be attributed to individual factors that contribute to the measure. Factors include network transfer, local processing and GPS queries which in turn can vary because of network type, device type, GPS state, user state, etc. We will discuss ANOVA analysis in more detail in § 8.1.3.

**Outliers:** AppInsight also flags outlier transactions to help developers identify performance bottlenecks. Transactions with duration greater than *(mean + (k * standard deviation))* in the group are marked as outliers. We use *k = 3* for our analysis.

## 7 Developer Feedback

The AppInsight server analyzes the collected traces using the methods described in § 6. The developers use a web UI to access the results. Figure 12 shows a collage of some of the views in the UI.

For ease of navigation, the UI groups together identical transactions (§ 6.4) ((a) in Figure 12). To allow easy mapping to the source code, groups are named by their entry event handler method. Within each group, transactions are sorted by duration and outliers are highlighted (b). Developers can select individual transactions to view their transaction graph which are shown as interactive plots (c). The plot also highlights the critical path (d). Within a critical path, we show the time spent on each component (e). The developer can thus easily identify the parts of the code that need to be optimized. Additional information, such as URLs and network type (3G or Wi-Fi) for web calls and the state of the GPS is also shown (e). We also provide variability analysis for each transaction group (f).

The UI also shows where each transaction fits within the particular app session. This view provides developers with the context in which a particular transaction occurred (e.g, at the start of a session).

The tool also reports certain common patterns within a group and across all transactions for an app. For example, it reports the most common critical paths in a transaction group, the most frequent transactions, common sequence of transactions, frequently interrupted transactions, etc. Using this information, the developer can focus her efforts on optimizing the common case.

Developers can also browse through crash reports. Crashes are grouped by their exception path. For each exception, the tool reports the exception type, shows the stack trace attached to the execution graph and highlights the exception path.

## 8 Results

We first present results from the live deployment of AppInsight, and some case studies of how AppInsight helped developers improve their app. Then, we present micro-benchmarks to quantify AppInsight's overhead and coverage.

### 8.1 Deployment

To select the apps to evaluate AppInsight with, we asked 50 of our colleagues to list 15 apps they regularly use on their Windows Phone. From these, we picked 29 most popular free apps. We also included an app that

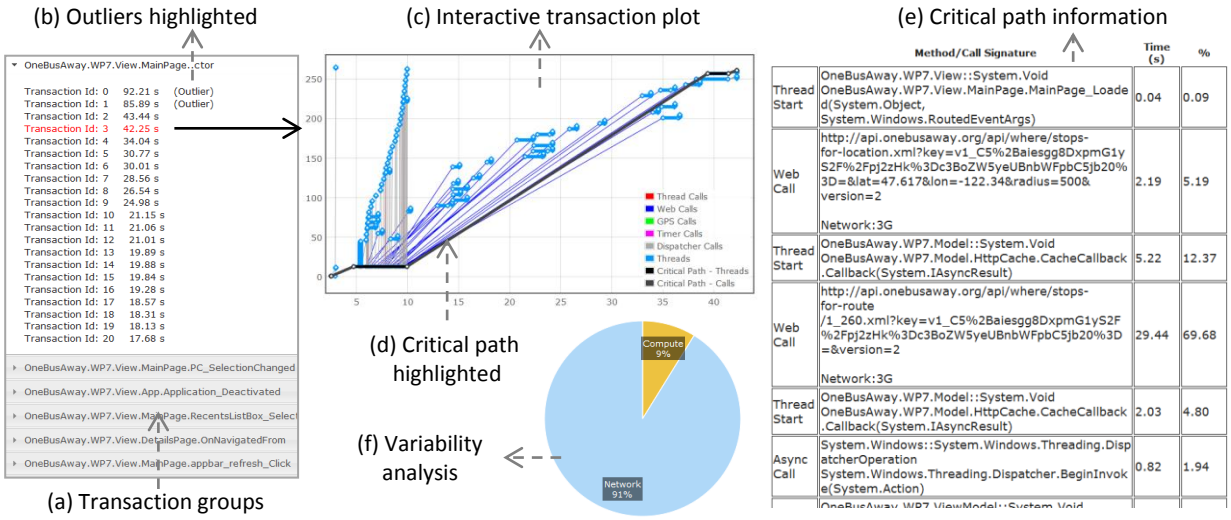



Figure 12: Collage of some of the views in the developer feedback UI.

| | |
|---|---|
| total num of apps | 30 |
| total participants | 30 |
| unique hardware models | 6 |
| unique hardware+firmware | 14 |
| start date | 03 April 2012 |
| end date | 15 August 2012 |
| total num of app launches (sessions) | 6752 |
| total minutes in apps | 33,060 |
| total user transactions | 167,286 |
| total timer transactions | 392,768 |
| total sensor transactions | 3587 |

Table 2: Summary statistics from our deployment

was developed by an author of this paper. The app was published several months before we started the AppInsight project, as an independent effort. We instrumented these 30 apps using AppInsight. Thirty users volunteered to run some of the instrumented apps on their personal phones. Often, they were already using many of the apps, so we simply replaced the original version with the instrumented version. All participants had their own unlimited 3G data plans.

Table 2 shows summary deployment statistics. Our data comes from 6 different hardware models. Over the course of deployment, we collected trace data from 6752 app sessions. There are a total of 563,641 transactions in this data. Over 69% of these are timer transactions, triggered by periodic timers (see § 6.2). Almost all of them are due to a one-second timer used in one of the gaming apps. In the rest of the section, we focus only on the 167,286 user transactions that we discovered in this data.

Table 3 shows basic usage statistics for some of the apps. Note the diversity in how often users ran each app, for how long, and how many user transactions were in each session. Over 40% of the user transactions were generated by a multiplayer game app. Figure 13 shows the CDF of the length of user transactions (i.e., the length of their critical path). Only 15% of the transactions last

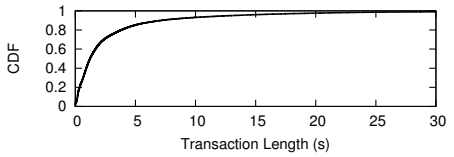



Figure 13: CDF of user-transaction duration.

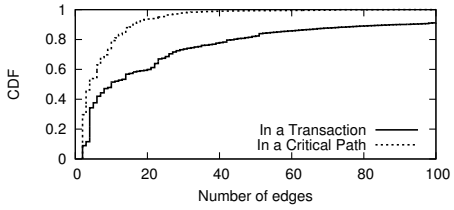



Figure 14: CDF of number of edges in user transactions and in critical paths. The X-axis has been clipped at 100. The top line ends at 347, and the bottom ends at 8,273.

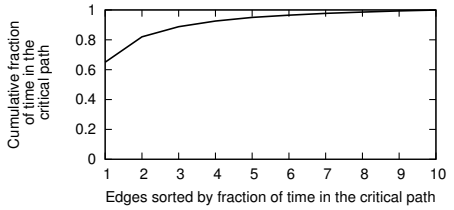



Figure 15: Cumulative fraction of time in the critical path as a function of number of edges.

more than 5 seconds. The app developer would likely want to focus his debugging and optimization efforts on these long-running transactions.

### 8.1.1 *User Transactions and Critical Paths*

In Table 3, we see that the average number of asynchronous calls per user transaction varies from 1.2 to 18.6

| App description | # Users | # Sessions | Avg session length (s) | #User trans--actions | #Async calls | Avg #parallel threads | #trans inter--rupted | Perf overhead ms/trans | Perf overhead ms/s | Network overhead b/trans | Extra data transfer |
|---|---|---|---|---|---|---|---|---|---|---|---|
| News aggregator | 22 | 604 | 88 | 11738 | 42826 | 5.50 | 1732 | 3.02 | 0.69 | 311 | 3.2% |
| Weather | 25 | 533 | 31 | 4692 | 8106 | 1.92 | 541 | 0.31 | 0.09 | 162 | 2.9% |
| Stock information | 17 | 460 | 32 | 4533 | 5620 | 1.00 | 486 | 0.20 | 0.06 | 91 | 8.6% |
| Social networking | 22 | 1380 | 622 | 48441 | 900802 | 7.60 | 6782 | 3.48 | 0.21 | 487 | 8.0% |
| Multiplayer game | 21 | 1762 | 376 | 68757 | 359006 | 2.28 | 719 | 0.18 | 0.26 | 27 | 79.0% |
| Transit info | 7 | 310 | 37 | 1945 | 40448 | 4.88 | 182 | 2.96 | 0.85 | 355 | 0.9% |
| Group discounts | 9 | 67 | 306 | 1197 | 3040 | 6.62 | 109 | 0.99 | 0.06 | 212 | 2.3% |
| Movie reviews | 7 | 48 | 394 | 1083 | 7305 | 6.56 | 80 | 0.51 | 0.08 | 97 | 2.7% |
| Gas station prices | 8 | 110 | 48 | 1434 | 2085 | 2.11 | 72 | 0.14 | 0.04 | 91 | 1.9% |
| Online shopping | 14 | 43 | 512 | 1705 | 25701 | 2.74 | 349 | 0.18 | 0.06 | 24 | 4.7% |
| Microblogging | 3 | 333 | 60 | 3913 | 19853 | 2.02 | 386 | 0.89 | 0.28 | 181 | 2.2% |
| Newspaper | 10 | 524 | 142 | 13281 | 24571 | 4.85 | 662 | 0.33 | 0.06 | 92 | 1.2% |
| Ticket service | 7 | 64 | 530 | 171 | 9593 | 3.70 | 38 | 0.05 | 0.57 | 9 | 2.9% |

Table 3: Summary statistics for 13 of the 30 apps. For conciseness, we highlight a single app out of each of the major app categories. The name of the app is anonymized. Overhead data is explained in § 8.3.1.

depending on the app. The average number of parallel threads per user transaction varies from 1 to 7.6. This high degree of concurrency in mobile apps is one of the key reasons why a system such as AppInsight is needed to identify the critical path in the complex graph that represents each user transaction.

Figure 14 offers another perspective on the complexity of user transactions and the value of AppInsight. It shows the CDF of the number of edges in a user transaction. While we have clipped the horizontal axis of this graph for clarity, there are user transactions with thousands of edges. Amidst this complexity, AppInsight helps the developers by identifying the critical path that limits the user-perceived performance. As the figure shows, the number of edges in critical paths are much fewer.

We also observe that not all edges in a critical path consume the same amount of time. Rather a few edges are responsible for most of the time taken by a transaction, as shown in Figure 15. This graph plots the cumulative fraction of transaction time as a function of the number of edges. We see that two edges are responsible for 82% of the transaction time. Application developers can focus on these edges to understand and alleviate the performance bottlenecks in their applications.

Investigating these time-hogging edges in critical paths, we find, expectedly, that network transfers are often to blame. In transactions that involve at least one network transfer (14.6% of total), 93% had at least one network transfer in the critical path and 35% had at least two. On an average, apps spend between 34-85% of the time in the critical path doing network transfer.

In contrast, location queries are not a major factor. In transactions that had a location query (0.03% of total), the query was in the critical path in only 19% of the cases. This occurs because most apps request for coarse location using WiFi or cell towers, without initializing the GPS device. Coarse location queries tend to be fast.

### *8.1.2 Exception paths*

AppInsight also helps in failure diagnosis. In our deployment, we collected 111 crash logs (from 16 apps), 43 of which involved asynchronous transactions where the standard stack trace that the mobile platform gives the app developer would not have identified the full path that led up to the crash.

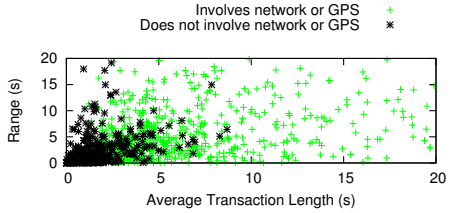



Figure 16: Variation in transaction length for each group. Both axes are clipped at 20.

### *8.1.3 Aggregate Analysis*

We analyzed the trace data from our deployment using techniques described in § 6.4. For the data in Table 2, we have 6,606 transaction groups across all apps.

**Understanding performance variance:** We first quantify the variance in transaction groups and then analyze the sources of the variance.

We find that 29% of the transaction groups contain multiple distinct critical paths. Further, even where there is a unique critical path, the dominant edge (the one that consumes most time) varies in 40% of the cases. This implies that the performance bottlenecks differ for different transactions even when the transactions correspond to the same activity.

Figure 16 shows the extent of performance variability we observe across transactions in a group. For each group, it plots the range (maximum - minimum) of transaction duration observed as a function of the average transaction duration. We see many activity groups with highly variable transaction duration. To show that this variability is not limited to cases with network transfers or location queries, we separately show activities that do not involve these two functions. While such activities have lower transaction duration on average, they too have highly variable performance. This variability can stem from the user's device, system load, user state, etc.

We identify the major sources behind the variability in transaction duration using ANOVA (§ 6.4). At the highest level, there are three factors that impact transaction duration: (*i*) network transfer, (*ii*) location queries, and (*iii*) local processing. Each of these factors can itself vary because of network type, device type, GPS state, user state, etc. For each transaction, we split the transaction duration into these three factors depending on where time is spent on the critical path and then find the contribution of each component to the variability of the transaction duration. For this analysis, we only use activity groups that have at least 10 transactions.

We focus first on activities that do not involve location queries. We find that the average contribution of network and local processing to the variability in the transaction duration was 66% and 34%. Much of the variability in transaction duration stems from the variability in network transfer time. Though, in 10% of the groups, local processing contributed to over 90% of the variability.

We further analyze those groups where network transfers were responsible for over 90% of the variability. We find that network type (3G or WiFi) plays an important role. On average, 3G-based transactions took 78% longer and had 155% more standard deviation compared to WiFi-based transactions. However, we also found groups with high variability in network-transfer time irrespective of the network type. This variation might be due to factors such as dynamic content and server delay that we do not capture.

We also analyze groups in which local processing was responsible for over 90% of the variability. We find groups where the variability can be entirely explained by the device type. For instance, in one group, transactions from Nokia Lumia 900 phones had 38% lower transaction times than those from Samsung Focus phones. One of the key differences between the two phones is that the Nokia has a 1.4 GHz processor compared to the Samsung with a 1 GHz processor. We also find transactions where the variability could be completely explained by the user herself. The duration of these transactions likely depend on user state that we do not capture.

Next, we analyze groups that have location queries in the critical path. We find that such queries contribute to the transaction duration variability in only one group. This is because, as noted above, most apps query for coarse location which is quick. In the group that queried for fine-grained location, the transaction time was highly correlated with the state of the GPS device. If it was not initialized, the query took 3–20 seconds; otherwise, it took roughly 1 ms.

**Outliers:** AppInsight flags transactions that take significantly longer than other transactions in the same group (§ 6.4). Overall, we find 831 outlier transactions and 287 groups with at least one outlier. These outliers span across 11 apps. 19% of the outliers are due to large network delays (with the transaction's network time being greater than the mean network time in the group by more than three orders of standard deviation), 76% are due to local processing and 5% are due to both. 70% of the transaction with large network delay was on 3G. The mean transaction duration of outliers with network delay was 16.6 seconds (14.1s median), and those because of local processing delay was 10 seconds (7.4s median). From the data, we can see that, local processing also plays a major role in long transactions.

Interestingly, the factors that explain most of the variability in a transaction group can be different from those that lead to outliers. We find groups in our data where the variability was primarily due to network transfers but the outlier was due to local processing.

### 8.2 Case Studies

We now describe how AppInsight helped app developers improve their applications.

#### *8.2.1 App 1*

One of the apps in our deployment was developed by an author of this paper (see § 8.1). AppInsight feedback helped the author improve the app in many ways. The following observations are based on 34 session traces representing 244 user transactions and 4 exception logs.

**Exceptions:** Before being instrumented with AppInsight, the app had been on the marketplace for 1.5 years. The developer had occasionally received crash logs from the Windows Phone developer portal, but logs contained only the stack trace of the thread that crashed. While the developer knew that a routine that split a line into words was crashing, there was not enough information for the developer to diagnose the failure. When the app was instrumented with AppInsight, the developer received the entire exception path. This included the web call and the URL from where the line was fetched. The developer replayed the URL in his app in a controlled setting, and discovered that his text-parsing routines did not correctly handle certain patterns of blank lines.

**UI sluggishness:** The aggregate analysis in AppInsight identified a user transaction with high variability in duration. The variability was attributed to local processing (time spent on thread execution). The developer spotted that only the user transactions at the start of user sessions experienced these abnormal latencies. He identified that certain system calls early in the app execution caused system DLLs to be loaded into memory. The time to load the DLLs was high and highly variable. Later transactions that used the same APIs did not experience high latency, as the DLLs were cached. This problem was not spotted in lab testing, since the DLLs are almost always in memory, due to continuous test runs. He redesigned his code to force-load the DLLs earlier.

**Wasted computation:** The feedback UI pointed the developer to frequently interrupted transactions. The developer noticed that in some cases, the background threads initiated by the interrupted transaction were not being terminated, thereby wasting the battery. The developer modified the code to fix the problem.

**Serial network operations:** The developer noticed that a common critical path consisted of web requests that were issued in a serial manner. The developer improved the user response time by issuing them in parallel.

#### *8.2.2 App 2*

AppInsight can help the developers optimize a "mature" app, that rarely experiences performance problems. For example, a popular app in our deployment has been in the marketplace for over 2 years and had gone through multiple rounds of updates. Our deployment traces had over 300 user sessions for this app, representing 1954 user transactions.

Aggregate analysis showed that 3G data latency significantly impacted certain common transactions in their app. In this case, the app developers were already aware of this problem and had considered adding caching to their app. However, they did not have good quantitative data to back up their decision. They were also impressed by the ease with which AppInsight highlighted the problem, for it had taken them a long time to pinpoint the fix. The developers are considering using AppInsight for their next release, especially to evaluate changes to the data caching policies.

#### *8.2.3 App 3*

We also instrumented an app that is under active development. This app was not part of our deployment – the developers tested the instrumented app in a small pilot of their own. Surprisingly, AppInsight revealed that custom instrumentation code that the developers had added was a major contributor to the poor performance of their app.

Analysis of trace data from other apps in our deployment has also shown many cases of wasteful computation, UI sluggishness, and serial network transactions in the critical path.

### 8.3 Micro-benchmarks

We now present micro-benchmarks to quantify AppInsight's overheads, and verify that AppInsight does not miss any user transactions.

#### *8.3.1 Overheads*

**App run time:** The impact of AppInsight on run time of the app is negligible. Individual logging operations simply write a small amount of data to a memory buffer, and hence are quite lightweight, as seen from Table 4. The buffer is flushed to disk when full[6] or when the app exits.

[6]We use a two-stage buffer to prevent data loss during flushing.

| Log Method | Overhead ($\mu$s) |
|---|---|
| LogUpcallStart | 6 |
| LogUpcallEnd | 6 |
| LogCallStart | 6 |
| LogCallEnd | 6 |
| LogCallbackStart | 6 |
| LogAsyncStart | 12 |
| LogObject | 12 |
| LogParameters | 60 |

Table 4: Overhead of AppInsight Logger. Averaged over 1 million runs, on a commonly used phone model.

In most cases, the buffer never gets full, so flushing happens only when the app exits. The disk write happens on a background thread, and takes only a few milliseconds.

To estimate the cumulative impact of logging operations on the apps that our users ran, we multiply the number of log calls in each user transaction by overheads reported in Table 4. The maximum overhead per user transaction is 30ms (average 0.57ms). Since most transactions are several seconds long (see Figure 13), we also calculated the approximate overhead per second. The maximum overhead is 5ms (average 0.21ms) per second. We believe that this is negligible. Table 3 shows the average overhead per transaction and per second for different apps. The overhead is quite low. We also note that our users reported no cases of performance degradation.

**Memory:** AppInsight uses a 1MB memory buffer. Typical apps consume around 50MB of memory, so the memory overhead is just 2%.

**Network:** AppInsight writes log records in a concise format to minimize the amount of data that must be uploaded. The median amount of trace data we upload is 3.8KB per app launch. We believe that this overhead is acceptable. We use two more metrics to further characterize the network overhead: $(i)$ bytes per transaction and $(ii)$ percentage of extra data transferred because of AppInsight compared to data consumed by the app. The last two columns of Table 3 shows these metrics for different apps. We see that the extra network overhead introduced by AppInsight is minimal for most apps. Recall that we use BTS (§ 4) to upload the data, which ensures that the upload does not interfere with the app's own communication. BTS also provides a "Wi-Fi Only" option, which defers data upload till the phone is connected to Wi-Fi.

**Size:** On average, the added instrumentation increased the size of the app binaries by just 1.2%.

**Battery:** The impact of AppInsight on battery life is negligible. We measured the overhead using a hardware power meter. We ran an instrumented app and the corresponding original app 10 times each. In each run, we manually performed the same UI actions. For the original app, the average time we spent in the app was 18.7 seconds across the 10 runs, and the average power consumption was 1193 mW, with a standard deviation of 34.8. For the instrumented version, the average time spent was also

18.7 seconds, and the average power consumption was 1205 mW. This 1% increase in power consumption is well within experimental noise (the standard deviation).

#### 8.3.2 Coverage

AppInsight uses several heuristics (see § 5) to reduce the amount of trace data it collects. To verify that we did not miss any user transactions because of these heuristics, we carried out a controlled experiment. First, we added extra instrumentation to the 30 apps that logs every method call as the app runs. Then, we ran these "fully instrumented" apps in a virtualized Windows Phone environment, driven by an automated UI framework, which simulates random user actions – tap screen at random places, random swiping, etc. We ran each app a 100 times, simulating between 10 and 30 user transactions each time[7]. Upon analyzing the logs, we found that the "extra" instrumentation did not discover any new user transaction. Thus we believe that AppInsight captures necessary data to reconstruct all user transactions. We also note that the full instrumentation overhead was as much as 7,000 times higher than AppInsight instrumentation. Thus, the savings achieved by AppInsight are significant.

## 9 Discussion

We now discuss some of the overarching issues related to AppInsight design.

**Causal relationships between threads:** AppInsight can miss certain casual relationship between threads. First, it does not track data dependencies. For example, two threads may use a shared variable to synchronize, wherein one thread would periodically poll for data written by another thread. Currently, AppInsight uses several heuristics to identify these programming patterns, and warns the developer that the inferred critical path may be incorrect. Tracking all data dependencies requires platform support [7], which we do not have. Second, AppInsight will miss implicit causal relationships, introduced by resource contention. For example, disk I/O requests made by two threads will get serviced one after the other, introducing an implicit dependency between the two threads. Monitoring such dependencies also requires platform support. Third, AppInsight cannot untangle complex dependencies introduced by counting semaphores. The Silverlight framework for Windows Phone [15] does not currently support counting semaphores. Finally, AppInsight does not track any state that a user transaction may leave behind. Thus, we miss dependencies resulting from such saved state.

**Definition of user transaction and critical path:** The definition of user transaction and critical path in § 3 does not address all scenarios. For example, some user interactions may involve multiple user inputs. Our current definition will break such interactions into multiple transactions. This may be incompatible with the developer's intuition of what constitutes a transaction. In case of multiple updates to the UI, our analysis produces one critical path for each update (§ 6.2). It is up to the developer to determine which of these paths are important to investigate. Despite these limitations, results in § 8 show that we can give useful feedback to the developer.

**Privacy:** Any system that collects trace data from user devices risks violating the user's privacy. To mitigate this risk, AppInsight does not store user or phone ids. Instead, we tag trace records with an anonymous hash value that is unique to that phone and that app. Since two apps running on the same phone are guaranteed to generate a different hash, it is difficult to correlate the trace data generated by different apps. This mechanism is by no means foolproof, especially since AppInsight collects data such as URLs accessed by the app. We continue to investigate this area further.

**Applicability to other platforms:** The current implementation of AppInsight works for the Windows Phone platform. However, the core ideas behind AppInsight can be applied to any platform that has certain basic characteristics. First, the applications need to have a single, dedicated UI thread. Second, we need the ability to rewrite byte code. Third, we need the ability to correctly identify all possible upcalls (i.e., calls into the user code by the system) and thread start events triggered by the UI itself. Fourth, the system needs to have a set of well-defined thread synchronization primitives. These requirements are not onerous. Thus we believe that AppInsight can be ported to other mobile platforms as well, although we have not done so.

## 10 Related Work

While we are not aware of a system with similar focus, AppInsight touches upon several active research areas.

**Correlating event traces:** AppInsight automatically infers causality between asynchronous events in the app execution trace. A number of systems for inferring causality between system events have been proposed, particularly in the context of distributed systems.

LagHunter [12] collects data about user-perceived delays in interactive applications. Unlike AppInsight, LagHunter is focused on synchronous delays such as rendering time. LagHunter requires the developer to supply a list of "landmark" methods, while AppInsight requires no input from the developer. LagHunter also occasionally collects full stack traces, which AppInsight does not do.

Magpie [4] is a system for monitoring and modeling server workload. Magpie coalesces Windows system event logs into transactions using detailed knowledge of application semantics supplied by the developer. On a Windows phone, system-event logs are not accessible to

[7] Some apps require non-random input at the beginning.

an ordinary app, so AppInsight does not use them. AppInsight also does not require any input from the app developer. Magpie's goal is to build a model of the system by characterizing the normal behavior. Our goal is to help the developer to detect anomalies.

XTrace [9] and Pinpoint [5] both trace the path of a request through a system using a special identifier attached to each individual request. This identifier is then used to stitch various system events together. AppInsight does not use a special identifier, and AppInsight does not track the request across process/app boundaries. Aguilera et. al. [2] use timing analysis to correlate trace logs collected from a system of "black boxes". While AppInsight can also use some of these log-analysis techniques, we do not treat the app as a black box, and hence are able to perform a finer grained analysis.

**Finding critical path of a transaction:** The goal of AppInsight is to detect the critical path in a user transaction. Yang and Miller did early work [19] on finding the critical path in the execution history of parallel and distributed programs. More recently, Barford and Crovella [3] studied critical paths in TCP transactions. While some of our techniques (e.g., building a graph of dependent events) are similar to these earlier works, our focus on mobile apps leads to a very different system design.

**Mobile application monitoring:** AppInsight is designed to monitor mobile-application performance in the wild. Several commercial products like Flurry [8] and PreEmptive [16] are available to monitor mobile-app usage in the wild. The developer typically includes a library to collect usage information such as number of app launches, session lengths and geographic spread of users. Through developer annotations, these platforms also allow for some simple timing information to be collected. But obtaining detailed timing behavior and critical-path analysis is not feasible with these platforms. To aid with diagnosing crashes, many mobile platforms report crash logs to developers when their application fails. While collecting such data over long term is important [10], it does not necessarily help with performance analysis [1]. Several researchers [17, 14] have studied energy consumption of mobile apps and have collected execution traces for that purpose. Our focus, on the other hand is on performance analysis in the wild.

## 11 Conclusion

AppInsight helps developers of mobile apps monitor and diagnose the performance of their apps in the wild. AppInsight instruments app binaries to collect trace data, which is analyzed offline to uncover critical paths and exception paths in user transactions. AppInsight is lightweight, it does not require any OS modifications, or any input from the developer. Data from a live deployment of AppInsight shows that mobile apps have a tremendous amount of concurrency, with many asynchronous calls and several parallel threads in a typical user transaction. AppInsight is able to correctly stitch together these asynchronous components into a cohesive transaction graph, and identify the critical path that determines the duration of the transaction. By examining such transactions from multiple users, AppInsight automatically identifies outliers, and sources of variability. AppInsight uncovered several bugs in one of our own app, and provided useful feedback to other developers.

## Acknowledgments

We thank Ronnie Chaiken and Gleb Kirosheev for discussions and support during AppInsight development. We also thank Petros Maniatis and the anonymous reviewers for their comments on earlier drafts of this paper.

## References

[1] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. There's an app for that, but it doesn't work. Diagnosing Mobile Applications in the Wild. In *HotNets*, 2010.

[2] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performane Debugging for Distributed System of Black Boxes. In *SOSP*, 2003.

[3] P. Barford and M. Crovella. Critical Path Analysis of TCP Transactions. In *ACM SIGCOMM*, 2000.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[5] M. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Mangement. In *NSDI*, 2004.

[6] J. Elliott, R. Eckstein, M. Loy, D. Wood, and B. Cole. *Java Swing, Second Edition*. O'Reilly, 2003.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Seth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[8] Flurry. `http://www.flurry.com/`.

[9] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP*, 2009.

[11] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Usenix Windows NT Symposium*, 1999.

[12] M. Jovic, A. Adamoli, and M. Hauswirth. Catch Me if you can: Performance Bug Detection in the Wild. In *OOPSLA*, 2011.

[13] S. Lidin. *Inside Microsoft .NET IL Assembler* . Microsoft Press, 2002.

[14] A. Pathak, Y. C. Hu, and M. Zhang. Where Is The Energy Spent Inside My App? Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys*, 2012.

[15] C. Perzold. *Microsoft Silverlight Edition: Programming Windows Phone 7*. Microsoft Press, 2010.

[16] Preemptive. `http://www.preemptive.com/`.

[17] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-Layer Approach. In *MobiSys*, 2011.

[18] A. Whitechapel. *Windows Phone 7 Development Internals*. Microsoft Press, 2012.

[19] C.-Q. Yang and B. P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *IEEE DCS*, 1988.

# Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE

Zhenyu Guo[†] Xuepeng Fan[†£] Rishan Chen[†‡] Jiaxing Zhang[†] Hucheng Zhou[†]
Sean McDirmid[†] Chang Liu [†§] Wei Lin[*] Jingren Zhou[*] Lidong Zhou[†]
[†]*Microsoft Research Asia* [*]*Microsoft Bing* [‡]*Peking University*
[£]*Huazhong University of Science and Technology* [§]*Shanghai Jiao Tong University*

## ABSTRACT

To minimize the amount of data-shuffling I/O that occurs between the pipeline stages of a distributed data-parallel program, its procedural code must be optimized with full awareness of the pipeline that it executes in. Unfortunately, neither pipeline optimizers nor traditional compilers examine both the pipeline and procedural code of a data-parallel program so programmers must either hand-optimize their program across pipeline stages or live with poor performance. To resolve this tension between performance and programmability, this paper describes PeriSCOPE, which automatically optimizes a data-parallel program's procedural code in the context of data flow that is reconstructed from the program's pipeline topology. Such optimizations eliminate unnecessary code and data, perform early data filtering, and calculate small derived values (e.g., predicates) earlier in the pipeline, so that less data—sometimes much less data—is transferred between pipeline stages. We describe how PeriSCOPE is implemented and evaluate its effectiveness on real production jobs.

## 1 INTRODUCTION

The performance of big data computations improves dramatically when they are parallelized and distributed on a large number of machines to operate on partitioned data [5, 14]. Such data-parallel programs involve pipelines of computation stages where I/O intensive data shuffling between these stages can dominate program performance. Unfortunately, data-shuffling I/O is difficult to optimize automatically because computations at each pipeline stage are encoded as flexible procedural code; current pipeline optimizers treat this code as a black box while compilers treat pipelines as black boxes and so are unaware of the data flow between the procedural code at different computation stages. The programmer must manually perform optimizations that require examining both the program's pipeline and procedural code; e.g., to not propagate unused data or to move the computation of smaller derived values to an earlier stage so less data is transmitted during data shuffling. Performing these optimizations by hand is not only tedious, it also limits code reuse from generic libraries.

So that programmers can write data-parallel programs with reasonable performance without sacrificing programmability, automatic optimizations must examine both the pipeline and procedural code of a data-parallel program. Common logical optimizations [8, 28, 34, 40, 43] for data-parallel programs focus on a high-level pipeline topology that is subject to relational query-optimization techniques. Unfortunately, at best relational components are extracted from procedural code into a relational optimization framework [20], which is limited by the inability of the relational framework to match the expressiveness of procedural code. We instead observe that projecting well-understood declarative pipeline properties into more flexible procedural code is intrinsically simpler than extracting declarative properties from procedural code. Such projection can then be used to reconstruct program data flow, enabling automatic optimizations of procedural code across pipeline stages that can improve I/O performance.

This paper presents PeriSCOPE, which automatically optimizes the procedural code of programs that run on SCOPE [8, 42], a production data-parallel computation system. PeriSCOPE connects the data flow of a SCOPE program's procedural code together by examining a high-level declarative encoding of the program's pipeline topology. PeriSCOPE then applies three core compiler-like optimizations to the program. *Column reduction* suppresses unused data in the pipeline based on the program's reconstructed data flow. *Early filtering* moves filtering code earlier in the pipeline to reduce how much data is transmitted downstream. Finally, *smart cut* finds a better boundary between pipeline stages in the data flow graph to minimize cross-stage I/O; e.g., the code that computes a predicate from two string values could be moved to an earlier stage, so that only a boolean value, rather than two string values, needs to be transmitted. The result is faster program execution because less data needs to be transferred between pipeline stages.

We have implemented PeriSCOPE and evaluated its effectiveness on 28,838 real SCOPE jobs from a large production cluster. We also evaluate end-to-end performance comparisons on eight real jobs.

The rest of the paper is organized as follows. Section 2 presents a sample SCOPE program to show the po-

```
1  t1 = EXTRACT query:string,clicks:long,market:int,...
2       FROM    "/users/foo/click_0342342"
3       USING   DefaultTextExtractor("-s")
4       HAVING  IsValidUrl(url) AND clicks != 0;
5  t2 = REDUCE  t1 ON query
6       PRODUCE query, score, mvalue, cvalue
7       USING   PScoreReducer("clicks")
8       HAVING  GetLength(query) > 4;
9  t3 = PROCESS t2 PRODUCE query, cscore
10      USING   SigReportProcessor("cvalue")
11      OUTPUT  t3 TO "/users/foo/click/0342342";
```

**Figure 1**: Declarative code that defines the pipeline of a sample SCOPE program. Rows of typed columns (line 1) are first extracted from a log file (line 2) using a default text extractor (line 3) and filtered based on certain conditions (line 4). Next, the input rows are *reduced* with a user-defined function `PScoreReducer` (line 7) to produce a table with four columns (line 6) after being filtered (line 8). Finally, the user-defined function `SigReportProcessor` (line 10) is applied to the result as it is emitted (line 11).

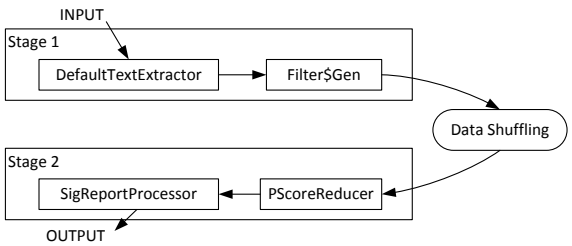



**Figure 2**: An illustration of the pipeline defined by the declarative code in Figure 1. The `Filter$Gen` operator is generated from the `HAVING` clauses on line 4 and 8 of Figure 1; other operators refer to user-defined functions. Each directed edge represents the data flow between operators.

tential benefits of PeriSCOPE's optimizations. The I/O-reduction optimizations in PeriSCOPE are described in Sections 3 and 4, with Section 3 covering column reduction and Section 4 discussing early filtering and smart cut, which are both forms of code motion. PeriSCOPE's implementation is covered in Section 5, followed by an evaluation in Section 6. We survey related work in Section 7 and conclude in Section 8.

## 2 A Motivating Example

We motivate PeriSCOPE by describing the pipeline-aware optimization opportunities that are found in a sample data-parallel program, which is adapted from a real SCOPE job. SCOPE is a distributed data-parallel computation system that employs a hybrid programming model where declarative SQL-like code describes a program's high-level pipeline structure, like other similar systems such as Hive [33], Pig [15], and DryadLINQ [40]. Figure 1 shows the declarative code of our sample job that is compiled into an execution pipeline, which we illustrate in Figure 2.

The operators of a SCOPE pipeline manipulate a data model of *rows* and *columns* and can be encoded as *user-defined functions* of procedural code that are either defined by the user or reused from generic libraries. A *computation stage* consists of one or more chained operators, and runs on a group of machines independently with partitioned data stored locally; *data-shuffling phases* then connect computation stages together by transmitting requisite data between machines. The pipeline in Figure 2 contains two *computation stages* that are separated by one data-shuffling phase according to the reduce call on line 5 in Figure 1. SCOPE applies logical optimizations, such as early selection, to programs according to the declarative structure of their pipeline. For example, the filtering clause on line 8 of Figure 1 can be applied before data shuffling; and so the `Filter$Gen` operator in the first stage of Figure 2 therefore includes the conditions from line 8 as well as line 4. Such logical optimizations apply only to the declarative code defined in Figure 1, treating the procedural code of the `DefaultTextExtractor`, `PScoreReducer`, and `SigReportProcessor` as black boxes.

The SCOPE program of Figure 1 is easily written by reusing two functions (`DefaultTextExtractor` and `SigReportProcessor`) from generic libraries while the encoding of the custom `PScoreReducer` function, shown in Figure 3, is straightforward. However, the program contains three serious I/O inefficiencies that need to be eliminated before it is "fast enough." First, the `if` statement on line 7 of Figure 3 is actually a procedural filter that discards rows. Such rows can be filtered out early so that they are not transmitted during the data-shuffling phase, which can be accomplished by splitting `PScoreReducer` into two parts as encoded in Figure 5: a `PScoreReducerPre` function that executes the computations of lines 5–7 in Figure 3 before data-shuffling; and a `PScoreReducerPost` that executes the rest of the computations from the original `PScoreReducer` function after data-shuffling. Our sample program's declarative SCOPE code is updated in Figure 4 to reflect this split, whose pipeline is illustrated in Figure 6.

Next, the `alteredQuery` column is transmitted only for computing a simple predicate on line 9 of Figure 3; the predicate computation can be done before the shuffling phase so that smaller boolean values are transmitted instead of strings. This is accomplished by computing the predicate in `PScoreReducerPre` on line 16 of Figure 5 and propagating its result as a column to

```
1  public class PScoreReducer : Reducer {
2  List<Row> Reduce(List<Row> input, string[] args){
3    maxImpr = 0; score = 0; mvalue = 0; cvalue = 0;
4    foreach (Row row in input) {
5     int impr = SmoothImpr(row[args[0]].Long());
6     bool incl = row["ctrls"].Contains(args[0]);
7     if (!incl && impr < 0) continue;
8     string[] keys = row["query"].Split(',');
9     bool p = row["alteredQuery"].ContainsAny(keys);
10     if (p)
11       score += ...;
12     if (impr > maxImpr)
13       maxImpr = impr;
14     if (impr * IMPR_RATIO > maxImpr) continue;
15     ... cvalue += ...
16     ... mvalue += ... row["market"] ...
17   }
18   outRow[1] = Normalize(score, ...);
19   outRow["mvalue"] = mvalue;
20   outRow["cvalue"] = cvalue;
21   ...
22   yield return outRow;
23 }}
```

**Figure 3**: The procedural code of the **PScoreReducer** user-defined function. Because **PScoreReducer** is a reduce operator, the preceding data shuffling ensures that rows having the same shuffling key are grouped together. For each group (**input**) of rows sharing the same shuffling key (line 2), this reduce operator iterates on each row in that group using a loop (lines 4-17) and outputs a single row as **outRow** for that group (line 22). The **impr** variable of line 5 represents an "improvement" that regulates accumulation of **mvalue** and **cvalue**.

**PScoreReducerPost** where it is used on line 29. An analogous transformation can be applied to **clicks**, which is used for computing **impr**, converting it from a long to an int.

Finally, the **SigReportProcessor** function called on line 10 of Figure 1 uses only the **cvalue** column, bound to its input parameter, that is computed by the **PScoreReducer** function; in contrast the **mvalue** column computed on lines 16 and 19 of Figure 3 is unused and therefore does not need to be computed and propagated in the **PScoreReducerPost** function of Figure 5. More importantly, if **mvalue** is eliminated, the **market** column does not need to be extracted in Figure 1 by **DefaultTextExtractor** and transmitted during the data-shuffling that is illustrated in Figure 2. Instead, the programmer can define their own specialized **MyTextExtractor** function (top of Figure 5) that does not extract and propagate the **market** column. The market column is also eliminated from Figure 4's declarative code.

## The PeriSCOPE approach

The optimized sample program in Figures 4 and 5 executes more efficiently at the expense of programmability: it can no longer reuse the **DefaultTextExtractor**

```
1  t1 = EXTRACT query:string,clicks:long,~~market:int,~~...
2       FROM    "/users/foo/click_0342342"
3       USING   MyTextExtractor
4       HAVING  IsValidUrl(url) AND clicks != 0
5                               AND GetLength(query) > 4;
6  t2 = PROCESS t1 PRODUCE query, impr, ...
7       USING   PScoreReducerPre;
8  t3 = REDUCE  t2 ON query
9       PRODUCE query, score, ~~mvalue,~~ cvalue
10      USING   PScoreReducerPost
11 t4 = PROCESS t3 PRODUCE query, cscore
12      USING   SigReportProcessor("cvalue")
13      OUTPUT  t4 TO "/users/foo/click/0342342";
```

**Figure 4**: Optimized declarative code of our sample program; strike-through text is original code that is eliminated.

```
1  public class MyTextExtractor : Extractor {
2  List<Row> Extract(StreamReader reader, string[] args){
3    ...
4    string[] columns = line.Split(',');
5    ~~int market = int.Parse(columns[2]);~~
6    ~~outRow[2].Set(market);~~
7    ...
8  }}
9  public class PScoreReducerPre : Processor {
10 List<Row> Process(List<Row> input, string[] args){
11   foreach (Row row in input) {
12     int impr = SmoothImpr(row["clicks"].Long());
13     bool incl = row["ctrls"].Contains("clicks");
14     if (!incl && impr < 0) continue;
15     string[] keys = row["query"].Split(',');
16     outRow["p"] =
17       row["alteredQuery"].ContainsAny(keys);
18     outRow["impr"] = impr;
19     ...
20     yield return outRow;
21 }}
22 public class PScoreReducerPost : Reducer {
23 List<Row> Reduce(List<Row> input, string[] args){
24   maxImpr = 0; score = 0; mvalue = 0; cvalue = 0;
25   foreach (Row row in input) {
26     int impr = row["impr"].Int();
27     ~~bool incl = row["ctrls"].Contains("clicks");~~
28     ~~if (!incl && impr < 0) continue;~~
29     if (row["p"].Boolean()) score += ...;
30     if (impr > maxImpr) maxImpr = impr;
31     if (impr * IMPR_RATIO > maxImpr) continue;
32     ... cvalue += ...
33     ~~... mvalue += ... row["market"] ...~~
34   }
35   outRow["score"] = Normalize(score, ...);
36   ~~outRow["mvalue"] = mvalue;~~
37   outRow["cvalue"] = cvalue;
38   ...
39   yield return outRow;
40 }}
```

**Figure 5**: Optimized procedural code of our sample program.

function, the logic for the **PScoreReducer** function is now distributed into two sections that execute in different pipeline stages, while the optimizations are tedious as the programmer must carefully move code across pipeline stages. These optimizations should be automated but cannot be performed by either logical pipeline optimizers that treat user-defined functions as black boxes or by

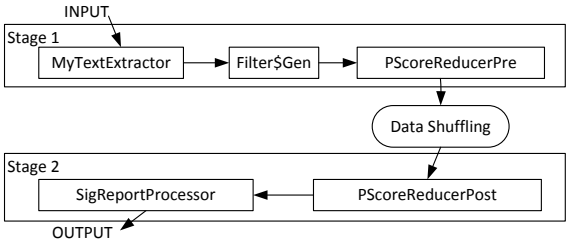



**Figure 6**: Resulting pipeline for the optimized job.

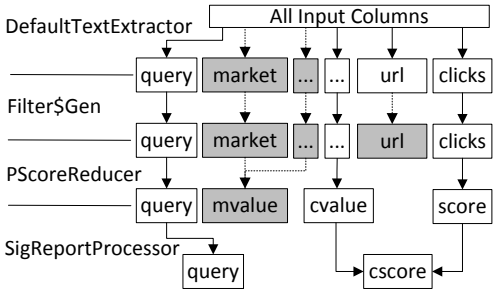



**Figure 7**: A simplified column-dependency graph for column reduction. Columns and computation in the shaded areas are removed by the column reduction optimization.

compilers that are unaware of pipelines.

PeriSCOPE automates such optimizations by considering both user-defined functions and the pipeline of a data-parallel program. In particular, PeriSCOPE reconstructs the data flow across the user-defined functions according to the pipeline topology and applies *column reduction* to remove unused columns along with the computations to compute them from user-defined functions; e.g., PeriSCOPE can eliminate the unused `mvalue` and `market` columns of our sample program. PeriSCOPE next identifies filtering conditions in a user-defined function and moves them earlier in the pipeline through *early filtering*; e.g., the `if` condition on line 7 of Figure 3 is moved earlier to reduce row propagation in the pipeline. Finally, PeriSCOPE applies *smart cut* that finds better boundaries between two stages to minimize data-shuffling I/O by moving size-reducing transformation upstream and size-enlarging transformation downstream in the pipeline. We describe how PeriSCOPE automates these optimizations in Sections 3 and 4.

## 3 Column Reduction

A user-defined function might not use a particular input column that is available to it in a calling pipeline. For example, the `SigReportProcessor` function of Section 2's sample program does not use the `mvalue` column of the pipeline encoded in Figure 1. In distributed data-parallel programs, transferring unused columns during data-shuffling can consume a significant amount of network I/O. As we discuss in Section 6, this problem commonly arises from the reuse of user-defined functions that we observe in production SCOPE jobs.

Column reduction is an optimization in PeriSCOPE that leverages information about how operators are connected together in a pipeline to eliminate unused columns from the program, removing their associated computation and I/O costs. The optimization analyzes the dependency information between the input and output columns of all operators in the pipeline; Figure 7 shows part of the column dependency graph for the example in Figure 1. An input or output column of an operator is represented as a vertex while an edge from a source column $s$ to a destination column $d$ indicates that $d$ has either a data or control dependency on $s$. Only data dependency edges are shown in Figure 7 as control dependency edges are too numerous to illustrate clearly. Because SCOPE allows a column to be accessed by name (e.g., line 6 in Figure 3) or index (e.g., line 18), a column read or write operation may be *unresolved* during compilation, where PeriSCOPE considers that this column could be any column that is visible to the user-defined function, leaving no opportunity for column reduction. Fortunately, as we discuss in Section 5, column accesses that cannot be resolved through simple optimization techniques are relatively rare—at only a 1.9% occurrence in our survey of real SCOPE jobs.

PeriSCOPE applies column reduction by computing a set of "used" output columns for each operator that are used as the input columns of succeeding operators in the pipeline topology. If the operator immediately precedes a data-shuffling phase, the shuffling-key columns are also required as used output columns. Any unused output columns of an operator are removed and, if the operator is a user-defined function, PeriSCOPE also rewrites it to remove all code that only contributes to computing the removed output columns. If any columns were removed, PeriSCOPE removes any input columns that are no longer used because of removed code and repeats column reduction again.

For example, thecolumn `mvalue` is removed from the `PScoreReducer` function because it is not used by the `SigReportProcessor` function listed in Figure 3.

This causes the code that computes `mvalue` to be removed (lines 36 and 33 in Figure 5), which further causes the output column `market` to be removed from the `DefaultTextExtractor` function. Finally, PeriSCOPE creates, through specialization that eliminates code, function whose code is semantically similar to the `MyTextExtractor` function of Figure 5.

## 4 Code Motion

Code motion moves code from user-defined functions across pipeline stages using two techniques: *early filtering* and *smart cut* that respectively reduce the number of rows and the size of each row transmitted during data shuffling. Because such code motion can be done only if safe, i.e., the result of the program is unchanged, we describe the correctness conditions of code motion using the example in Figure 3, with a focus on identifying the domain knowledge that is needed to define correctness.

Moving a statement across a data-shuffling phase is not always safe. For the statements on lines 12–14 in Figure 3, the value of `maxImpr` depends on the processing order of the input rows. Because data shuffling re-orders rows based on a shuffling key, computing `maxImpr` before and after data shuffling would yield different results. PeriSCOPE makes the following two observations on data shuffling, each leading to a safety rule for code motion. First, the data-shuffling phase reads the shuffling-key columns of each row, leaving other columns untouched; i.e.,

RULE 1. *PeriSCOPE must not move a statement after data shuffling if it generates shuffling-key columns.*

Second, the data-shuffling phase can change transparently the number and order of the rows processed on each machine through re-partitioning and grouping. Any computation that relies on the number or order of the rows, which we say is *stateful*, cannot be moved across the data-shuffling phase; i.e.,

RULE 2. *PeriSCOPE must not move a stateful statement across the data shuffling phase.*

PeriSCOPE applies loop dependency analysis [4] to the body of the main loop for each user-defined function to identify stateful statements as those that have loop-carried dependencies. A loop-carried dependency indicates that the destination statement relies on the execution of the source statement from an earlier iteration. For example, the statement on line 12 of Figure 3 relies on its reassignment on line 13 from previous iterations and is therefore stateful; by similar reasoning, the statements on lines 11, and 13–15 are also stateful.

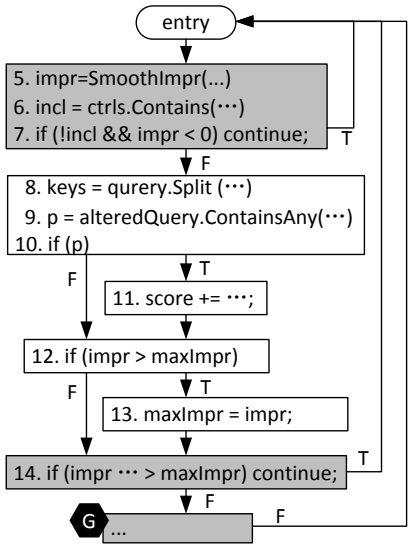



**Figure 8**: Control flow graph for the loop body in `PScore-Reducer` in Figure 3. Edges marked T and F are branches that are taken when the last predicate in the source basic block evaluates to true and false, respectively. The vertices in gray are the basic blocks that contain filtering statements.

Care must be taken in identifying stateful statements. A statement might be stateful if it calls a routine that transparently accesses data; such as by reading and writing a global variable. Also, statefulness is only determined by dependencies on rows that are iterated by the main loop of a user-defined function; PeriSCOPE treats inner loops as single statements.

### Early Filtering

Early filtering is applied to the first user-defined function in a computation stage that executes after a data shuffling phase. PeriSCOPE first identifies filtering statements in the user-defined function's main loop, which are statements that branch back to the beginning of the main loop. Figure 8 shows the control flow graph for the loop body of `PScoreReducer` in Figure 3; statements 7, 14 and the end of basic block `G` are identified as filtering statements. Because earlier filtering will reduce the number of rows that are iterated on later, PeriSCOPE must ensure that moving a filtering statement does not change the cumulative effect of any downstream stateful statements; i.e.,

RULE 3. *PeriSCOPE must not move a filtering statement before a data shuffling phase if the statement is, or is reachable from, a stateful statement.*

This rule excludes statement 14 in Figure 8 because it is stateful, and excludes the last statement in basic block G because it is reachable from statement 14. PeriSCOPE next identifies code that computes the filtering condition by applying backward slicing [35], which starts from the identified filtering statement and collects, as its *backward slice*, the statements that can affect it. The backward slice of statement 7 in Figure 8 includes statements 5–7. PeriSCOPE then copies the entire backward slice upward causing rows to be filtered out before data shuffling occurs. Given rule 3, the slice cannot contain any stateful statements and so this copy is always safe. Finally, the conditions of the moved filter can now be assumed in the original user-defined function, enabling the removal of code through dead code elimination. For the code in Figure 8, statement 7 is removed because `(!incl && impr < 0)` is always false; no row otherwise is permitted past the data-shuffling phase due to early filtering. Statement 6 is then removed because `incl` is not used anymore, causing `ctrls` to become unused in the user-defined function. As a result, early filtering not only reduces the number of rows that are transferred across a data shuffling phase, but can also trigger column reduction (e.g., on `ctrls`).

## Smart Cut

The cross-stage flow of data across the network in a data-parallel program is significantly more expensive than a traditional program whose data flows only through memory. PeriSCOPE therefore aims at re-partitioning the code by finding *smart cuts* as shuffling I/O boundaries that minimize cross-stage data flow. Finding smart-cuts can be formulated as a compiler-like instruction scheduling problem [2, 25]. However, while a compiler usually rearranges instructions to improve instruction-level parallelism on a specific CPU architecture, smart cut reorders statements to reduce the amount data transmitted across the network.

Smart cut is applied to user-defined functions that are immediately adjacent to data-shuffling phases. PeriSCOPE first applies if-conversion [3] to the body of the main loop for a given user-defined function so that the loop body becomes a single basic block, which is necessary because instruction scheduling can only be applied to blocks of non-branching instructions. Figure 9 shows the simplified result for the code segment on lines 5–15 of Figure 3, after lines 6 and 7 are removed according to early filtering. Every statement is now guarded with a predicate that specifies the path condition of its execution; e.g., the statement on line 13 is guarded with predicate `p1` because it is executed only when `p1` is true.

```
5   (T)  impr = SmoothImpr(row["clicks"].Long());

8   (T)  keys = row["query"].String().Split(',');
9   (T)  p = row["alteredQuery"].ContainsAny(keys);

11  (p)  score += ...;
12  (T)  p1 = impr > maxImpr;
13  (p1) maxImpr = impr;
14  (T)  p2 = !(impr * IMPR_RATIO > maxImpr);
15  (p2) ... cvalue += ...
```

**Figure 9**: Simplified if-conversion result for lines 5–15 in Figure 3. `T` stands for `True` which means that the statement always executes.

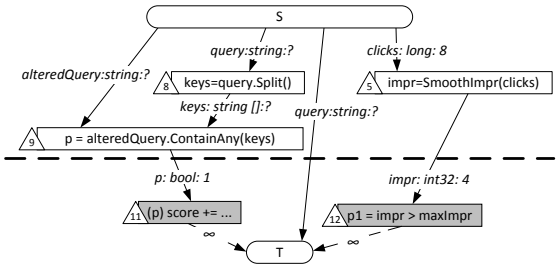



**Figure 10**: Labeled data dependency graph with a smart cut. Statements 13–15 are omitted. Statements in gray are stateful.

PeriSCOPE then builds a data dependency graph [2, 25] for this basic block using the SSA [13] format. Vertices in the data dependency graph are instructions, while directed edges represent read-after-write (RAW) data dependencies where sink instructions use variables defined in the source instructions. PeriSCOPE labels the edges with the name and byte size of the dependent variables, which are either columns or local variables. Figure 10 shows part of the labeled data dependency graph for our example; PeriSCOPE further adds two vertices `S` and `T` to represent the overall input and output of this code snippet, respectively. PeriSCOPE also adds an edge labeled `query` from `S` to `T` as `query` is used as the shuffling key and should always be transmitted. To ensure that rules 1 and 2 are not violated, PeriSCOPE adds directed edges from `S` to any statement that is either stateful or generates shuffling keys before the data-shuffling phase, and adds directed edges from any stateful statement after the data-shuffling phase to `T`; all of these edges have an infinite weight to ensure that those statements are never moved across the data-shuffling phase.

The smart-cut problem is now reduced to one of finding an edge cut between `S` and `T` in the data dependency graph that minimizes the total byte size of all dependent variables on edges across the cut. The problem appears

similar to the minimum cut problem [10] of a directed graph. However, there are two subtle differences between our problem and the standard minimum cut problem:

- Data flows across the data-shuffling phase in only one direction, so all edges must have the same direction across the cut.
- When multiple edges on a cut are labeled with the same variable name, the byte size of that variable is only counted once as it only needs to be transferred once.

Computing an optimal edge cut statically is difficult because the precise weights of some edges depend on dynamic data. For example, it is hard to statically estimate the weights of string-typed columns and variables as their length is unknown. In practice, PeriSCOPE resorts to a simple heuristic-based technique to identify opportunities to move code across data-shuffling phases. Specifically, PeriSCOPE looks for a simple pattern with a variable computed from one or more input columns. If the total size of the input columns that are used only for computing this variable is larger than the size of this variable, this computation should be moved to an earlier stage. Similarly, PeriSCOPE also looks for a reverse pattern where a variable is used to generate one or more output columns. In Figure 10, the input columns `alteredQuery` and `query` from Figure 3 are used to compute variable `p` in the optimize function of Figure 5. Although the `alteredQuery` column is never used elsewhere, the `query` column is used in a later stage. Because the byte size of a string type (`alteredQuery`) is always larger than that of a boolean variable (`p`), the cut should cross the edges labeled with `p`, instead of those labeled with `alteredQuery`. The same reasoning applies to the computation of `impr` from `clicks`. In the end, edges between statements 9 and 11, and between statements 5 and 12, are selected for the smart cut.

Finally, PeriSCOPE applies instruction scheduling according to the selected cut. In our example (Figure 9), statements 5, 8 and 9 are moved before data shuffling. The recorded schema across the data-shuffling phase is changed accordingly where two new columns are added: the boolean-typed `p`, and the integer-typed `impr`, and two old columns (the string-typed `alteredQuery` and the long-typed `clicks`) are deleted. The result is similar to the code shown in Figures 4 and 5 in Section 2.

## 5 Implementation

PeriSCOPE examines a SCOPE program's operators, the definition of the rows used by the operators, and the program's pipeline topology represented as a directed acyclic graph (DAG) in the program's execution plan. The operators and row definitions are extracted from .NET binary executables, while the pipeline topology is represented as an XML file. PeriSCOPE extends ILSpy [32], a de-compiler from .NET byte-code to C# code, and Cecil [36], a library to generate and inspect .NET assemblies, to implement PeriSCOPE as two components. PeriSCOPE's *optimizer* is built on top of ILSpy to specialize all operators in the input execution plan, applying all PeriSCOPE's optimizations to operators (as user-defined functions) as found at the intermediate representation (IR) level. The *generator* emits new bytecode for user-defined functions and generates all utility code for the program, such as new row schemas and their related serialization routines, as well as the new SCOPE description file for the execution plan.

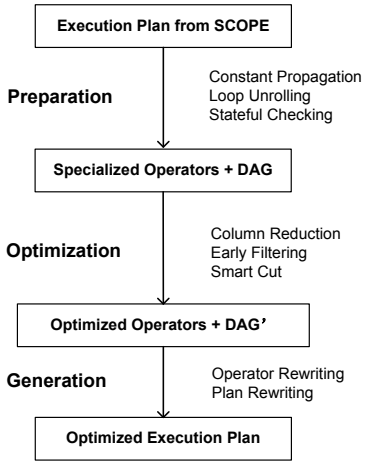



**Figure 11**: Optimization flow in PeriSCOPE.

The optimizer and generator components are both implemented in C# with 7,334 and 2,350 lines of code, respectively. Figure 11 illustrates PeriSCOPE's optimization flow with three major tasks, each containing several steps, where the optimizer performs the first two tasks, while the generator performs the last. Plan rewriting updates the original DAG XML file that describes pipeline topology because some original operators are now split into different computation stages.

SCOPE user-defined functions use a column index or name to access a column. When such index is a variable, program analysis can only make conservative assumption on what column is being manipulated, significantly reducing the opportunity for PeriSCOPE's optimizations. In our survey of SCOPE programs, we found

that 11.4% of the column accesses use variable indices. Fortunately, our investigation shows that many column index variables are determined by the input arguments to the user-defined functions, which is the case on lines 7 and 10 of Figure 1, and constant propagation can be applied to resolve their concrete value. However, not every column index can be resolved by constant propagation only. A user-defined function might enumerate columns using a variable in a loop, which we addressed by standard loop unrolling [2, 25] when the schema describing the data is known. A column index could also be determined by the value of another column, which PeriSCOPE cannot deal with very well. Fortunately, we have found that among the 11.4% of the jobs that use a variable as a column index, PeriSCOPE is able to resolve 83.3% of them, leaving just 1.9% of the jobs containing unresolved column access.

Instead of directly rewriting operator code, PeriSCOPE copies operator code when it needs to be written because a user-defined function can be reused multiple times in a job, each reuse requiring different code transformations. Likewise, row type schema definitions and serialization code are copied and transformed as columns are eliminated from different points in the pipeline.

## 6 Evaluation

We use a real trace of 28,838 jobs from a 10,000 machine SCOPE production cluster to evaluate PeriSCOPE's core I/O reduction optimizations of column reduction, early filtering, and smart cut. Our evaluation focuses on first assessing the overall potential for these optimizations and second evaluating in detail the effectiveness of these optimizations on the end-to-end performance of several real production jobs. With an average analysis time of 3.9 seconds for each job, our current implementation successfully analyzes 19,914 (69%) of the 28,838 jobs. PeriSCOPE fails on the rest of these jobs given limitations in our implementation primarily relating to inconsistent SCOPE versions (18.9%) or outright ILSpy decompilation failures (8.5%), but a minority involve code that cannot be analyzed in general due to unresolved column indices (1.9%) or for reasons that we have yet been unable to determine (1.7%). Table 1 shows that 14.05% of the jobs are eligible for column reduction optimization, 10.47% for early filtering, and 5.35% for smart cut. Some jobs are eligible for multiple types of optimizations, and so the total percentage (22.18%) of jobs that are eligible for those optimizations is lower than the sum of the three.

We next examine the user-defined functions of these jobs. We found that these jobs used only 2,108 unique user-defined functions, meaning many jobs are encoded purely in declarative code that leverages pre-existing user-defined functions. About 16.4% of the user-defined functions are reused more than ten times, where the most popular user-defined function is reused 4,076 times. We suppose that the heavy reuse of user-defined functions creates more opportunities for PeriSCOPE's optimizations. And in fact, about 80.2% of the user-defined functions in jobs eligible for column reduction were reused at least 13 times, confirming our speculation that generic library functions contain a lot of redundancies that can be optimized away. On the other hand, no such correlation is observed for early filtering or smart cut, whose eligibility appear to be unrelated to reuse. Finally, 637 (30.2%) unique user-defined functions used in these jobs have arguments in their function bodies that are used as branch conditions or column names, while 79.1% of the user-defined function invocations in the job scripts contain constant parameters. Specialization of such user-defined functions is a necessary pre-processing step to resolve columns and apply PeriSCOPE's optimizations.

| optimization | eligible jobs |
|---|---|
| column reduction | 4,052 (14.05%) |
| early filtering | 3,020 (10.47%) |
| smart cut | 1,544 ( 5.35%) |
| Total | 6,397 (22.18%) |

**Table 1**: Optimization coverage statistics which lists the number and the percentage of the jobs that are eligible for the given optimization.

### Case Study

To understand the overall effectiveness of PeriSCOPE's optimizations, we compare the performance of the jobs before and after our optimization in terms of both execution time and the amount of I/O used during the data-shuffling phase. Ideally, we would carry out this experiment with representative benchmarks, which unfortunately do not exist. We therefore select eight real and typical SCOPE jobs that are eligible for at least one of PeriSCOPE's optimizations and whose input data is still available on the cluster. The selected jobs are mostly related to web-search scenarios that process crawler traces, search query histories, search clicks, user information, and product bidding logs. Our experiment executes these real production jobs (cases 1–8 in Figure 12) on various number of machines. Specifically, cases 1, 2, and 4 use 1,000 machines, case 3 uses 10 machines, cases 5–7 use 192 machines, while case 8 uses 100 machines.

Figure 12 shows the performance-gain breakdown for

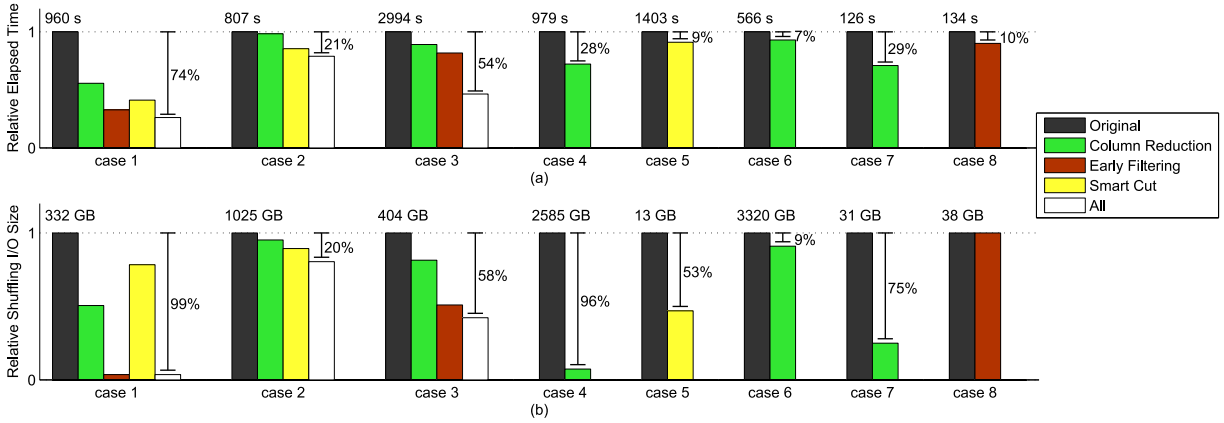



**Figure 12**: Performance gains with PeriSCOPE's column reduction, early filtering, and smart cut optimizations; chart (a) labels unoptimized job time in seconds while chart (b) labels total unoptimized job shuffling I/O size in GB; the bars in each case represent the effectiveness of each optimization relative to unoptimized execution time (a) or shuffling I/O (b); shorter bars indicate more reduction; the "All" bar is only shown for cases that are eligible for more than one PeriSCOPE's optimization; final case reduction of time or I/O is presented as a percentage next to an `I`-bar; both the execution time and the shuffling I/O are average values with a relative standard deviation (RSD) ranging from 7.3% to 23.0% due to the nature of our shared computing environment.

our chosen 8 production jobs in terms of a reduction in both execution time and data-shuffling I/O. The unoptimized and optimized versions of each job are executed three times; we report the average. Due to the nature of our shared computing environment we are using, we see high relative standard deviations (7.3% to 23.0%) in our latency experiments, while the reduction numbers in data-shuffling I/O is a more reliable indicator. In particular, highest standard deviations are seen for cases 5 (23.0% and 22.6%) and 6 (18.0% and 14.9%), indicating that the reductions are insignificant statistically in those cases. The execution time reduction for case 8 (10%) is also statistically insignificant with standard deviations of 13.4% and 7.3%. Case 1 benefits from all three of PeriSCOPE's optimizations, cases 2–3 are eligible for two, while cases 4–8 are only eligible for one each. PeriSCOPE reduces data-shuffling I/O in all cases but the last by between 9% and 99%; the last case incurs no benefit for reasons discussed below. Execution time is reduced by between 7% to 74%, which, beyond data-shuffling I/O, includes other tasks such as executing data-processing code, and reading and writing data to and from storage. Case 4 is particularly sensitive to storage overhead as this job extracts data from a 2.26 TB log file.

Column reduction can be applied six of the eight jobs, yielding I/O reductions ranging from 4.8% up to 96% that depend on how many columns are removed compared to the total byte size of all columns. Column reduction on case 4 removes 18 columns out of 22; the reducer that executes immediately after an extractor uses only 4 of the columns extracted. For case 7, only 2 out of 31 columns are used by its reducer; other columns are consumed by other operators and are not transmitted across the data-shuffling phase.

The effectiveness of early filtering depends highly on the goal of filtering. We have found that filtering conditions simply exclude rows whose columns have invalid values. While such case is rare, early filtering leads only to a negligible I/O reduction; case 8 is exactly this case. The execution time of case 8 is still reduced because PeriSCOPE moved the filtering computation to before the data-shuffling phase, improving the parallelism because more resource (136 CPU cores) are allocated to the stage before shuffling than after (42). When the filtering does not check for invalid values, they usually exclude a large number of rows and early filtering is quite effective. As an extreme case, data-shuffling I/O is reduced by 99% in case 1 because the vast majority of the rows in this job are filtered out and so do not need to be transmitted in the pipeline. The opportunity for early filtering discovered by PeriSCOPE was not obvious: *7* **`if`** conditions, some of them deeply nested, select desired rows for various computations, and manually writing a single filtering condition to replicate these **`if`** conditions is not trivial for a developer.

In contrast to early filtering, smart cut will always deliver I/O reductions when it can be applied. Compu-

tations that trigger smart cut typically involve one column that is mapped to a column of a smaller size, usually via the conversion from string to some arithmetic types, or size-reduction operations such as *Trim* and *Substring*. Binary operations (e.g., $+, *, ==, >$) between two input columns can also trigger smart cut. For example, case 5 contains two string-typed columns as start and end event timestamps; the job parses the two as integer timestamps and computes their delta for the elapsed time of the event, where smart cut causes the delta to be precomputed.

### Discussion

Overall, we found that column reduction and smart cut are always effective in reducing data-shuffling I/O while the effectiveness of early filtering highly depends on the purpose of the filtering. Our experiments have also demonstrated that programmers often write inefficient data-parallel programs; we speculate that they are either unaware of how to optimize these programs or are valuing programmability over performance. In this context, PeriSCOPE's optimizations are valuable as the programmer can be less concerned about I/O performance.

Even experienced programmers who value performance could eventually rely on PeriSCOPE's optimizations to avoid hand-optimizing their code and allow them to reuse more existing code in their programs. In this case, the reliability and predictability of PeriSCOPE's optimizations are as important as the optimizations' effectiveness; we leave an exploration of this topic to future work.

## 7 Related Work

PeriSCOPE is closely related to a large body of research in the areas of data-parallel computation, distributed database systems [16] and query optimizations, and compiler optimizations [2, 4, 25]. Instead of attempting to cover those areas thoroughly, we focus on the most related research that lies in the intersection of those three areas.

### Distributed data-parallel systems

MapReduce [14] has inspired a lot of follow-up research on large-scale distributed data-parallel computation, including Hadoop [5] and Dryad [18]. The MapReduce model has been extended [38] with Merge to support joins and adapted [11] to support pipelining. High-level languages for data-parallel computation have also been proposed for ease of programming. Examples include Sawzall [31], Pig Latin [29, 15], SCOPE [8], Hive [33, 34], and DryadLINQ [40]. In addition, FlumeJava [9] is a Java library for programming and managing MapReduce pipelines that proposes new parallel-collection abstractions, does deferred evaluation, and optimizes the data flow graph of an execution plan internally before executing. Nova [27] is a work-flow manager with support for stateful incremental processing which pushes continually arriving data through graphs of Pig programs executing on Hadoop clusters. Cascading [7] is a Java library built on top of Hadoop for defining and executing complex, scale-free, and fault tolerant data processing work-flows. Bu et al. [6] shows how recursive SQL queries may be translated into iterative Hadoop jobs. Programs in those systems go through a compilation and optimization process to generate code for a low-level execution engine, such as MapReduce and Dryad. All of them support user-defined functions that are treated as black boxes during optimization of the program's pipeline.

PeriSCOPE's optimizations work at the level of bytecode operators and pipeline descriptions, which are typically the result of the existing compilation and optimization process. Conceptually, the approaches taken by the PeriSCOPE's optimizations can be applied to data-parallel systems other than SCOPE, because almost all systems produce a pipeline with operators that call user-defined functions. The coverage and the effectiveness of the concrete optimizations, however, vary due to their different programming models and language runtime implementation. We show two cases where the differences in those systems matter. First, the data models differ, ranging from a relational data model (e.g., SCOPE) or its variations (e.g., Hive, Pig), to the object model (e.g., FlumeJava and DryadLINQ), which introduces different opportunities and difficulties for PeriSCOPE's optimizations. For example, with an object model, PeriSCOPE does not need to resolve the column access index any more, because all fields are accessed explicitly. Also, in an object model, declaring a new schema requires explicit class/object definitions. The resulting inconvenience often cause developers to reuse existing object definitions that contains unneeded fields, offering more opportunities for column reduction. Developers sometimes write custom (de-)serialization functions for an object to achieve better performance, which would pose challenges to PeriSCOPE's optimizations that cause schema changes: those functions must be modified accordingly.

Second, different systems might define different interfaces to their user-defined functions; those interfaces represent different trade offs between expressiveness and ease of analysis. For example, SCOPE exposes a collection of records to a mapper while others usually take a single record as the input to a mapper (e.g., in the

*MapReduce* framework in Hadoop). Other examples include the reducer interface in SCOPE versus the UDAF (user-defined aggregation function) interface in Hive, where the former exposes the record collection and the latter only receives a single value, and is usually applied to a single column. The more restricted the interface and the less expressive the language, the easier it is to analyze. The interface definition also influences where the optimization opportunities lie. For example, if a user-defined function is defined to take a single column as its input, cross-column relationships are now explicitly expressed, reducing the need for program analysis and optimizations.

## Database optimizations

Most of the data-parallel systems adopt a hybrid programming model that combines declarative relational operators with procedural user-defined functions, and are heavily influenced by database systems. The support of relational operators in those systems allows jobs to be specified easily, while at the same time facilitates database optimizations based on relational algebra.

There are interesting similarities between some of the PeriSCOPE's optimizations and the classic database optimizations. Early filtering in PeriSCOPE corresponds naturally to early selection in database optimizations. The counterpart to column reduction in database optimization is early projection that drops unused columns as early as possible. Such logical optimizations [28] have already been proposed for data-parallel programs, but they cannot be readily applied when user-defined functions are involved because they rely on relational operators.

A line of related research focuses on extracting relational queries from user-defined functions. HadoopToSQL [19] transforms MapReduce queries to use the indexing, aggregation, and grouping features provided by SQL databases, taking advantage of advanced storage engines by employing symbolic execution to extract selection and projection conditions. Manimal [20] similarly extracts relational operations such as selection, projection, and data compression from user-defined functions through static data flow analysis. Early filtering and column reduction are possible in Manimal because those optimizations have clear relational interpretations. But PeriSCOPE can also remove unnecessary code from the user-defined functions, while Manimal never rewrites user-defined functions as it can only optimize the relational layer. For example, lines 27, 28, 33, 36 in Figure 5, which are eliminated by PeriSCOPE, would not be removed by Manimal. As a result, PeriSCOPE further removes columns `ctrls` and `market`, as well as the code on lines 5 and 6 in Figure 5 through column reduction, which Manimal cannot do.

Neither Manimal nor HadoopToSQL support smart cut because neither system rewrites any user-defined functions in its optimizations. For smart cut, the closest concept in database optimization that we are aware of is the notion of "virtual columns" from Oracle [30], where a computed column is lazily evaluated when it is used, similar in spirit to moving a computation to a later place in the code. Such lazy evaluation is limited to special cases and cannot be performed on user-defined functions in general.

## Program analysis and optimizations

The need to analyze user-defined functions, by means of techniques such as data flow analysis [2, 4, 25], abstract interpretation [12], and symbolic execution [17], has already been recognized. Ke et al. [21] focuses on data statistics and computational complexity of user-defined functions to cope with data skew. Sameer et al. [1] concludes that certain data and code properties can improve performance of data-parallel jobs, and presents the RoPE system that adaptively re-optimizes jobs by collecting statistics on such code and data properties in a distributed context. Scooby [37] analyzes the data flow relationships of SCOPE's user-defined functions between input and output tables, such as column independence and column equality, by extending the Clousot analysis infrastructure [22]. Yuan et al. [39] define the *associative-decomposable* property of a reducer function to enable partial aggregation automatically after analysis on the reducer functions. Sudo [41] identifies a set of interesting user-defined functions, such as pass-through, one-to-one, and monotonicity, and develops a framework to reason about data-partition properties, functional properties, and data shuffling in order to eliminate unnecessary data shuffling. Sudo analyzes user-defined functions to infer their properties, but never rewrites any user-defined functions.

Compilation of declarative language has huge impact on the efficiency of a high-performance and high-throughput environment. Steno [26] can translate code for declarative LINQ [24, 23] queries both in serial C# programs and DryadLINQ programs to type-specialized, inlined, and loop-based procedural code that is as efficient as hand-optimized code. PeriSCOPE similarly applies those optimizations in program specialization as a preparation step, although differences in the language designs between SCOPE and LINQ lead to different challenges and approaches. Steno can automatically generate

code for operators expressed in LINQ, but has to treat external functions called inside operators as black boxes. PeriSCOPE instead works with compiled user-defined functions, which include such external functions.

## 8 Concluding Remarks

Optimizing distributed data-parallel computation benefits from an inter-disciplinary approach that involves database systems, distributed systems, and program languages. In particular, PeriSCOPE has demonstrated performance gains on real production jobs by applying program analysis and compiler optimizations in the context of the pipelines that these jobs execute in. Much more can be done. We can explore how to enhance the reliability and predictability of PeriSCOPE's optimizations so programmers can reuse existing code as well as write straightforward code without much guilt that performance is being sacrificed. Going further, we can explore how the programming model itself can be enhanced with more guarantees about program behavior, allowing for even more aggressive optimizations that further improve performance.

## Acknowledgement

We thank the anonymous reviewers for their valuable comments. We are particularly grateful to our shepherd, John Wilkes, for his insightful feedback.

## References

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2012. USENIX Association.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 177–189, Austin, Texas, 1983. ACM.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[5] Apache. Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`, March 2012.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *The Proceedings of the VLDB Endowment (PVLDB)*, 3:285–296, 2010.

[7] Cascading. `http://www.cascading.org/`, March 2012.

[8] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *The Proceedings of the VLDB Endowment (PVLDB)*, 1:1265–1276, 2008.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, Toronto, Canada, 2010. ACM.

[10] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 324–333. SIAM, 1997.

[11] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2010. USENIX Association.

[12] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28, June 1996.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems (TOPLAS)*, 13:451–490, 1991. ACM.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–113, San Fancisco, CA, USA, 2004. USENIX Association.

[15] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of MapReduce: The Pig experience. *The Proceedings of the VLDB Endowment (PVLDB)*, 2:1414–1425, 2009.

[16] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3), 1995.

[17] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems (TOPLAS)*, pages 501–538, 1985. ACM.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd European conference on Computer systems (EuroSys)*, pages 59–72, Lisbon, Portugal, 2007. ACM.

[19] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: a mapReduce query optimizer. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, pages 251–264, Paris, France, 2010. ACM.

[20] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *The Proceedings of the VLDB Endowment (PVLDB)*, 4:385–396, 2011.

[21] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *Proceedings of the 13th Workshop on Hot Topics in Operating System (HotOS)*, Napa, CA, USA, 2011. USENIX Association.

[22] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction*, pages 197–212, Budapest, Hungary, 2008. Springer-Verlag.

[23] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 706–706. ACM, 2006.

[24] Microsoft. LINQ. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`, February 2007.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[26] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 121–131, San Jose, CA, USA, 2011. ACM.

[27] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous Pig/Hadoop workflows. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1081–1090, Athens, Greece, 2011. ACM.

[28] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *Annual Technical Conference (ATC)*. USENIX Association, 2008.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, Vancouver, Canada, 2008. ACM.

[30] Oracle. Virtual column. `http://www.oracle-base.com/articles/11g/virtual-columns-11gr1.php`, March 2011.

[31] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 2005.

[32] SharpDevelop. ILSpy. `http://wiki.sharpdevelop.net/ilspy.ashx`, June 2012.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a MapReduce framework. *The Proceedings of the VLDB Endowment (PVLDB)*, 2:1626–1629, 2009.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 996–1005, Long Beach, CA, USA, 2010. IEEE.

[35] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449, San Diego, CA, USA, 1981. IEEE Press.

[36] Xamarin. Mono Cecil. `http://www.mono-project.com/Cecil`.

[37] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, pages 19–35, Los Angeles, CA, USA, 2009. Springer-Verlag.

[38] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1029–1040, Beiing, China, 2007. ACM.

[39] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 247–260. ACM, 2009.

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2008. USENIX Association.

[41] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y.Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2012. USENIX Association.

[42] J. Zhou, N. Bruno, M. chuan Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. In *The VLDB Journal*. Springer-Verlag, 2012.

[43] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 1060–1071. IEEE, 2010.

# MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han[+], Scott Marshall[+], Byung-Gon Chun[*], and Sylvia Ratnasamy[+]

[+]*University of California, Berkeley* [*]*Yahoo! Research*

## Abstract

We present MegaPipe, a new API for efficient, scalable network I/O for message-oriented workloads. The design of MegaPipe centers around the abstraction of a *channel* – a per-core, bidirectional pipe between the kernel and user space, used to exchange both I/O requests and event notifications. On top of the channel abstraction, we introduce three key concepts of MegaPipe: partitioning, lightweight socket (lwsocket), and batching.

We implement MegaPipe in Linux and adapt memcached and nginx. Our results show that, by embracing a clean-slate design approach, MegaPipe is able to exploit new opportunities for improved performance and ease of programmability. In microbenchmarks on an 8-core server with 64 B messages, MegaPipe outperforms baseline Linux between 29% (for long connections) and 582% (for short connections). MegaPipe improves the performance of a modified version of memcached between 15% and 320%. For a workload based on real-world HTTP traces, MegaPipe boosts the throughput of nginx by 75%.

## 1 Introduction

Existing network APIs on multi-core systems have difficulties scaling to high connection rates and are inefficient for "message-oriented" workloads, by which we mean workloads with short connections[1] *and/or* small messages. Such message-oriented workloads include HTTP, RPC, key-value stores with small objects (e.g., RAMCloud [31]), etc. Several research efforts have addressed aspects of these performance problems, proposing new techniques that offer valuable performance improvements. However, they all innovate within the confines of the traditional socket-based networking APIs, by either *i)* modifying the internal implementation but leaving the APIs untouched [20, 33, 35], or *ii)* adding new APIs to complement the existing APIs [1, 8, 10, 16, 29]. While these approaches have the benefit of maintaining backward compatibility for existing applications, the need to maintain the *generality* of the existing API – e.g., its reliance on file descriptors, support for blocking and nonblocking communication, asynchronous I/O, event polling, and so forth – limits the extent to which it can be optimized for performance. In contrast, a clean-slate redesign offers the opportunity to present an API that is specialized for high performance network I/O.

An ideal network API must offer not only high performance but also a simple and intuitive programming abstraction. In modern network servers, achieving high performance requires efficient support for *concurrent I/O* so as to enable scaling to large numbers of connections per thread, multiple cores, etc. The original socket API was not designed to support such concurrency. Consequently, a number of new programming abstractions (e.g., `epoll`, `kqueue`, etc.) have been introduced to support concurrent operation without overhauling the socket API. Thus, even though the basic socket API is simple and easy to use, programmers face the unavoidable and tedious burden of layering several abstractions for the sake of concurrency. Once again, a clean-slate design of network APIs offers the opportunity to design a network API from the ground up with support for concurrent I/O.

Given the central role of networking in modern applications, we posit that it is worthwhile to explore the benefits of a clean-slate design of network APIs aimed at achieving both high performance and ease of programming. In this paper we present MegaPipe, a new API for efficient, scalable network I/O. The core abstraction MegaPipe introduces is that of a *channel* – a per-core, bi-directional pipe between the kernel and user space that is used to exchange both asynchronous I/O requests *and* completion notifications. Using channels, MegaPipe achieves high performance through three design contributions under the roof of a single unified abstraction:

**Partitioned listening sockets**: Instead of a single listening socket shared across cores, MegaPipe allows applications to clone a listening socket and partition its associated queue across cores. Such partitioning improves performance with multiple cores while giving applications control over their use of parallelism.

**Lightweight sockets**: Sockets are represented by file descriptors and hence inherit some unnecessary file-related overheads. MegaPipe instead introduces lwsocket, a lightweight socket abstraction that is not wrapped in file-

[1] We use "short connection" to refer to a connection with a small number of messages exchanged; this is not a reference to the absolute time duration of the connection.

related data structures and thus is free from system-wide synchronization.

**System Call Batching**: MegaPipe amortizes system call overheads by batching asynchronous I/O requests and completion notifications within a channel.

We implemented MegaPipe in Linux and adapted two popular applications – memcached [3] and the nginx [37] – to use MegaPipe. In our microbenchmark tests on an 8-core server with 64 B messages, we show that MegaPipe outperforms the baseline Linux networking stack between 29% (for long connections) and 582% (for short connections). MegaPipe improves the performance of a modified version of memcached between 15% and 320%. For a workload based on real-world HTTP traffic traces, MegaPipe improves the performance of nginx by 75%.

The rest of the paper is organized as follows. We expand on the limitations of existing network stacks in §2, then present the design and implementation of MegaPipe in §3 and §4, respectively. We evaluate MegaPipe with microbenchmarks and macrobenchmarks in §5, and review related work in §6.

## 2 Motivation

Bulk transfer network I/O workloads are known to be inexpensive on modern commodity servers; one can easily saturate a 10 Gigabit (10G) link utilizing only a single CPU core. In contrast, we show that message-oriented network I/O workloads are very CPU-intensive and may significantly degrade throughput. In this section, we discuss limitations of the current BSD socket API (§2.1) and then quantify the performance with message-oriented workloads with a series of RPC-like microbenchmark experiments (§2.2).

### 2.1 Performance Limitations

In what follows, we discuss known sources of inefficiency in the BSD socket API. Some of these inefficiencies are general, in that they occur even in the case of a single core, while others manifest only when scaling to multiple cores – we highlight this distinction in our discussion.

**Contention on Accept Queue** (multi-core): As explained in previous work [20, 33], a single listening socket (with its `accept()` backlog queue and exclusive lock) forces CPU cores to serialize queue access requests; this hotspot negatively impacts the performance of both producers (kernel threads) enqueueing new connections and consumers (application threads) accepting new connections. It also causes CPU cache contention on the shared listening socket.

**Lack of Connection Affinity** (multi-core): In Linux, incoming packets are distributed across CPU cores on a flow basis (hash over the 5-tuple), either by hardware (RSS [5]) or software (RPS [24]); all receive-side processing for the flow is done on a core. On the other hand, the transmit-side processing happens on the core at which the application thread for the flow resides. Because of the serialization in the listening socket, an application thread calling `accept()` may accept a new connection that came through a remote core; RX/TX processing for the flow occurs on two different cores, causing expensive cache bouncing on the TCP control block (TCB) between those cores [33]. While the per-flow redirection mechanism [7] in NICs eventually resolves this core disparity, short connections cannot benefit since the mechanism is based on packet sampling.

**File Descriptors** (single/multi-core): The POSIX standard requires that a newly allocated file descriptor be the lowest integer not currently used by the process [6]. Finding 'the first hole' in a file table is an expensive operation, particularly when the application maintains many connections. Even worse, the search process uses an explicit per-process lock (as files are shared within the process), limiting the scalability of multi-threaded applications. In our `socket()` microbenchmark on an 8-core server, the cost of allocating a single FD is roughly 16% greater when there are 1,000 existing sockets as compared to when there are no existing sockets.

**VFS** (multi-core): In UNIX-like operating systems, network sockets are abstracted in the same way as other file types in the kernel; the Virtual File System (VFS) [27] associates each socket with corresponding file instance, inode, and dentry data structures. For message-oriented workloads with short connections, where sockets are frequently opened as new connections arrive, servers quickly become overloaded since those globally visible objects cause system-wide synchronization cost [20]. In our microbenchmark, the VFS overhead for socket allocation on eight cores was 4.2 times higher than the single-core case.

**System Calls** (single-core): Previous work has shown that system calls are expensive and negatively impact performance, both directly (mode switching) and indirectly (cache pollution) [35]. This performance overhead is exacerbated for message-oriented workloads with small messages that result in a large number of I/O operations.

In parallel with our work, the Affinity-Accept project [33] has recently identified and solved the first two issues, both of which are caused by the shared listening socket (for complete details, please refer to the paper). We discuss our approach (partitioning) and its differences in §3.4.1. To address other issues, we introduce the concept of lwsocket (§3.4.2, for FD and VFS overhead) and batching (§3.4.3, for system call overhead).

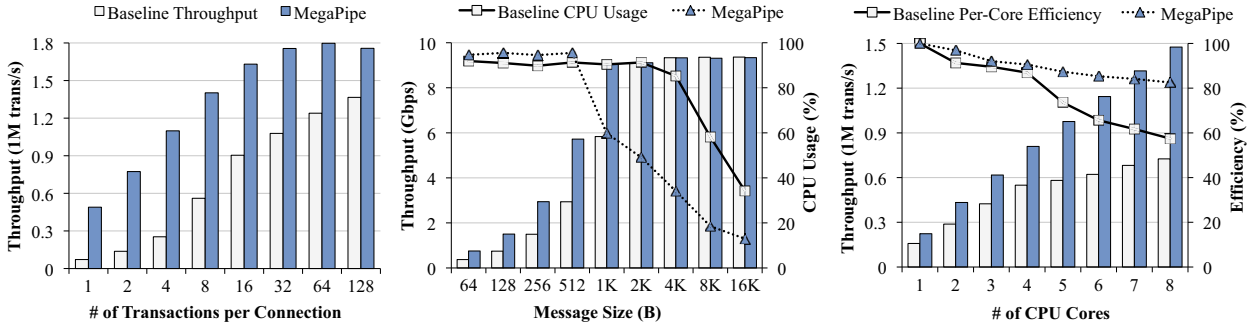



**Figure 1:** (a) the negative impact of connection lifespan (with 64 B messages on eight cores), (b) message size (with ten transactions per connection on eight cores), and (c) increasing number of cores (with 64 B messages and ten transactions per connection).

### 2.2 Performance of Message-Oriented Workloads

While it would be ideal to separate the aforementioned inefficiencies and quantify the cost of each, tight coupling in semantics between those issues and complex dynamics of synchronization/cache make it challenging to isolate individual costs.

Rather, we quantify their *compound* performance impact with a series of microbenchmarks in this work. As we noted, the inefficiencies manifest themselves primarily in workloads that involve *short connections* or *small-sized messages*, particularly with increasing numbers of CPU cores. Our microbenchmark tests thus focus on these problematic scenarios.

**Experimental Setup**: For our tests, we wrote a pair of client and server microbenchmark tools that emulate RPC-like workloads. The client initiates a TCP connection, exchanges multiple request and response messages with the server and then closes the connection.[2] We refer to a single request-response exchange as a *transaction*. Default parameters are 64 B per message and 10 transactions per connection, unless otherwise stated. Each client maintains 256 concurrent connections, and we confirmed that the client is never the bottleneck. The server creates a single listening socket shared by eight threads, with each thread pinned to one CPU core. Each event-driven thread is implemented with `epoll` [8] and the non-blocking socket API.

Although synthetic, this workload lets us focus on the low-level details of network I/O overhead without interference from application-specific logic. We use a single server and three client machines, connected through a dedicated 10G Ethernet switch. All test systems use the Linux 3.1.3 kernel and ixgbe 3.8.21 10G Ethernet device driver [2] (with interrupt coalescing turned on). Each machine has a dual-port Intel 82599 10G NIC, 12 GB of DRAM, and two Intel Xeon X5560 processors, each of which has four 2.80 GHz cores. We enabled the multi-queue feature of the NICs with RSS [5] and FlowDirector [7], and assigned each RX/TX queue to one CPU core.

In this section, we discuss the result of the experiments Figure 1 labeled as "Baseline." For comparison, we also include the results with our new API, labeled as "MegaPipe," from the same experiments.

**Performance with Short Connections**: TCP connection establishment involves a series of time-consuming steps: the 3-way handshake, socket allocation, and interaction with the user-space application. For workloads with short connections, the costs of connection establishment are not amortized by sufficient data transfer and hence this workload serves to highlight the overhead due to costly connection establishment.

We show how connection lifespan affects the throughput by varying the number of transactions per connection in Figure 1(a), measured with eight CPU cores. Total throughput is significantly lower with relatively few (1–8) transactions per connection. The cost of connection establishment eventually becomes insignificant for 128+ transactions per connection, and we observe that throughput in single-transaction connections is roughly 19 times lower than that of long connections!

**Performance with Small Messages**: Small messages result in greater relative network I/O overhead in comparison to larger messages. In fact, the per-message overhead remains roughly constant and thus, independent of message size; in comparison with a 64 B message, a 1 KiB message adds only about 2% overhead due to the copying between user and kernel on our system, despite the large size difference.

To measure this effect, we perform a second microbenchmark with response sizes varying from 64 B to 64 KiB (varying the request size in lieu of or in addition to the response size had almost the same effects). Figure 1(b) shows the measured throughput (in Gbps) and CPU usage for various message sizes. It is clear that connections with

[2] In this experiment, we closed connections with RST, to avoid exhaustion of client ports caused by lingering TIME_WAIT connections.

small-sized messages adversely affect the throughput. For small messages ($\leq$ 1 KiB) the server does not even saturate the 10G link. For medium-sized messages (2–4 KiB), the CPU utilization is extremely high, leaving few CPU cycles for further application processing.

**Performance Scaling with Multiple Cores**: Ideally, throughput for a CPU-intensive system should scale linearly with CPU cores. In reality, throughput is limited by shared hardware (e.g., cache, memory buses) and/or software implementation (e.g., cache locality, serialization). In Figure 1(c), we plot the throughput for increasing numbers of CPU cores. To constrain the number of cores, we adjust the number of server threads and RX/TX queues of the NIC. The lines labeled "Efficiency" represent the measured per-core throughput, normalized to the case of perfect scaling, where *N* cores yield a speedup of *N*.

We see that throughput scales relatively well for up to four cores – the likely reason being that, since each processor has four cores, expensive off-chip communication does not take place up to this point. Beyond four cores, the marginal performance gain with each additional core quickly diminishes, and with eight cores, speedup is only 4.6. Furthermore, it is clear from the growth trend that speedup would not increase much in the presence of additional cores. Finally, it is worth noting that the observed scaling behavior of Linux highly depends on connection duration, further confirming the results in Figure 1(a). With only one transaction per connection (instead of the default 10 used in this experiment), the speedup with eight cores was only 1.3, while longer connections of 128 transactions yielded a speedup of 6.7.

# 3 MegaPipe Design

MegaPipe is a new programming interface for high-performance network I/O that addresses the inefficiencies highlighted in the previous section and provides an easy and intuitive approach to programming high concurrency network servers. In this section, we present the design goals, approach, and contributions of MegaPipe.

## 3.1 Scope and Design Goals

MegaPipe aims to accelerate the performance of message-oriented workloads, where connections are short and/or message sizes are small. Some possible approaches to this problem would be to extend the BSD Socket API or to improve its internal implementation. It is hard to achieve optimal performance with these approaches, as many optimization opportunities can be limited by the legacy abstractions. For instance: *i)* sockets represented as files inherit the overheads of files in the kernel; *ii)* it is difficult to aggregate BSD socket operations from concurrent connections to amortize system call overheads. We leave optimizing the message-oriented workloads with those dirty-slate (minimally disruptive to existing API semantics and legacy applications) alternatives as an open problem. Instead, we take a clean-slate approach in this work by designing a new API from scratch.

We design MegaPipe to be conceptually simple, self-contained, and applicable to existing event-driven server applications with moderate efforts. The MegaPipe API provides a unified interface for various I/O types, such as TCP connections, UNIX domain sockets, pipes, and disk files, based on the completion notification model (§3.2) We particularly focus on the performance of network I/O in this paper. We introduce three key design concepts of MegaPipe for high-performance network I/O: partitioning (§3.4.1), lwsocket (§3.4.2), and batching (§3.4.3), for reduced per-message overheads and near-linear multi-core scalability.

## 3.2 Completion Notification Model

The current best practice for event-driven server programming is based on the readiness model. Applications poll the readiness of interested sockets with `select/poll/epoll` and issue non-blocking I/O commands on the those sockets. The alternative is the completion notification model. In this model, applications issue asynchronous I/O commands, and the kernel notifies the applications when the commands are complete. This model has rarely been used for network servers in practice, though, mainly because of the lack of socket-specific operations such as `accept/connect/shutdown` (e.g., POSIX AIO [6]) or poor mechanisms for notification delivery (e.g., SIGIO signals).

MegaPipe adopts the completion notification model over the readiness model for three reasons. First, it allows transparent batching of I/O commands and their notifications. Batching of non-blocking I/O commands in the readiness model is very difficult without the explicit assistance from applications. Second, it is compatible with not only sockets but also disk files, allowing a unified interface for any type of I/O. Lastly, it greatly simplifies the complexity of I/O multiplexing. Since the kernel controls the rate of I/O with completion events, applications can blindly issue I/O operations without tracking the readiness of sockets.

## 3.3 Architectural Overview

MegaPipe involves both a user-space library and Linux kernel modifications. Figure 2 illustrates the architecture and highlights key abstractions of the MegaPipe design. The left side of the figure shows how a multi-threaded application interacts with the kernel via MegaPipe *channels*. With MegaPipe, an application thread running on each core opens a separate channel for communication

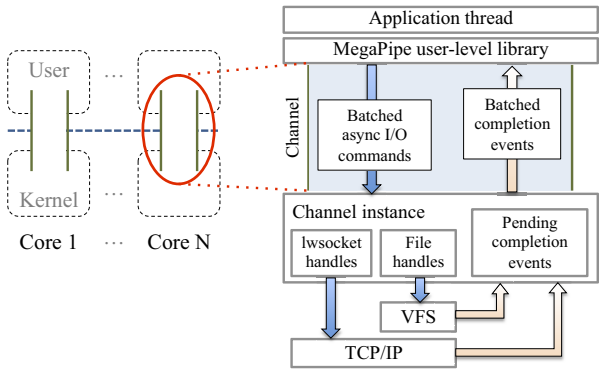



**Figure 2:** MegaPipe architecture

between the kernel and user-space. The application thread registers a *handle* (socket or other file type) to the channel, and each channel multiplexes its own set of handles for their asynchronous I/O requests and completion notification events.

When a listening socket is registered, MegaPipe internally spawns an independent accept queue for the channel, which is responsible for incoming connections to the core. In this way, the listening socket is not shared by all threads, but *partitioned* (§3.4.1) to avoid serialization and remote cache access.

A handle can be either a regular file descriptor or a lightweight socket, *lwsocket* (§3.4.2). lwsocket provides a direct shortcut to the TCB in the kernel, to avoid the VFS overhead of traditional sockets; thus lwsockets are only visible within the associated channel.

Each channel is composed of two message streams: a request stream and a completion stream. User-level applications issue asynchronous I/O requests to the kernel via the request stream. Once the asynchronous I/O request is done, the completion notification of the request is delivered to user-space via the completion stream. This process is done in a *batched* (§3.4.3) manner, to minimize the context switch between user and kernel. The MegaPipe user-level library is fully responsible for transparent batching; MegaPipe does not need to be aware of batching.

## 3.4 Design Components

### 3.4.1 Listening Socket Partitioning

As discussed in §2.1, the shared listening socket causes two issues in the multi-core context: *i)* contention on the accept queue and *ii)* cache bouncing between RX and TX cores for a flow. Affinity-Accept [33] proposes two key ideas to solve these issues. First, a listening socket has per-core accept queues instead of the shared one. Second, application threads that call `accept()` prioritize their local accept queue. In this way, connection establishment becomes completely parallelizable and independent, and all the connection establishment, data transfer, and application logic for a flow are contained in the same core.

In MegaPipe, we achieve essentially the same goals but with a more controlled approach. When an application thread associates a listening socket to a channel, MegaPipe spawns a separate listening socket. The new listening socket has its own accept queue which is only responsible for connections established on a particular subset of cores that are explicitly specified by an optional `cpu_mask` parameter.[3] After a shared listening socket is registered to MegaPipe channels with disjoint `cpu_mask` parameters, all channels (and thus cores) have completely partitioned backlog queues. Upon receipt of an incoming TCP handshaking packet, which is distributed across cores either by RSS [5] or RPS [24], the kernel finds a "local" accept queue among the partitioned set, whose `cpu_mask` includes the current core. On the application side, an application thread accepts pending connections from its local queue. In this way, cores no longer contend for the shared accept queue, and connection establishment is vertically partitioned (from the TCP/IP stack up to the application layer).

We briefly discuss the main difference between our technique and that of Affinity-Accept. Our technique requires user-level applications to partition a listening socket explicitly, rather than transparently. The downside is that legacy applications do not benefit. However, explicit partitioning provides more flexibility for user applications (e.g., to forgo partitioning for single-thread applications, to establish one accept queue for each physical core in SMT systems, etc.) Our approach follows the design philosophy of the Corey operating system, in a way that "applications should control sharing" [19].

Partitioning of a listening socket may cause potential load imbalance between cores [33]. Affinity-Accept solves two cases of load imbalance. For a short-term load imbalance, a non-busy core running `accept()` may steal a connection from the remote accept queue on a busy CPU core. For a long-term load imbalance, the flow group migration mechanism lets the NIC to distribute more flows to non-busy cores. While the current implementation of MegaPipe does not support load balancing of incoming connections between cores, the techniques made in Affinity-Accept are complementary to MegaPipe. We leave the implementation and evaluation of connection load balancing as future work.

### 3.4.2 lwsocket: Lightweight Socket

`accept()`ing an established connection is an expensive process in the context of the VFS layer. In Unix-like op-

[3] MegaPipe currently does not support runtime reconfiguration of `cpu_mask` after it is initially set, but we believe that this is easy to add.

erating systems, many different types of open files (disk files, sockets, pipes, devices, etc.) are identified by a *file descriptor*. A file descriptor is an integer identifier used as an indirect reference to an opened *file instance*, which maintains the status (e.g., access mode, offset, and flags such as `O_DIRECT` and `O_SYNC`) of the opened file. Multiple file instances may point to the same *inode*, which represents a unique, permanent file object. An inode points to an actual type-specific kernel object, such as TCB.

These layers of abstraction offer clear advantages. The kernel can seamlessly support various file systems and file types, while retaining a unified interface (e.g., `read()` and `write()`) to user-level applications. The CPU overhead that comes with the abstraction is tolerable for regular disk files, as file I/O is typically bound by low disk bandwidth or high seek latency. For network sockets, however, we claim that these layers of abstraction could be overkill for the following reasons:

*(1) Sockets are rarely shared.* For disk files, it is common that multiple processes share the same open file or independently open the same permanent file. The layer of indirection that file objects offer between the file table and inodes is useful in such cases. In contrast, since network sockets are rarely shared by multiple processes (HTTP socket redirected to a CGI process is such an exception) and not opened multiple times, this indirection is typically unnecessary.

*(2) Sockets are ephemeral.* Unlike permanent disk-backed files, the lifetime of network sockets ends when they are closed. Every time a new connection is established or torn down, its FD, file instance, inode, and dentry are newly allocated and freed. In contrast to disk files whose inode and dentry objects are cached [27], socket inode and dentry cannot benefit from caching since sockets are ephemeral. The cost of frequent (de)allocation of those objects is exacerbated on multi-core systems since the kernel maintains the inode and dentry as globally visible data structures [20].

To address the above issues, we propose lightweight sockets – *lwsocket*. Unlike regular files, a lwsocket is identified by an arbitrary integer within the channel, not the lowest possible integer within the process. The lwsocket is a common-case optimization for network connections; it does not create a corresponding file instance, inode, or dentry, but provides a straight shortcut to the TCB in the kernel. A lwsocket is only locally visible within the associated MegaPipe channel, which avoids global synchronization between cores.

In MegaPipe, applications can choose whether to fetch a new connection as a regular socket or as a lwsocket. Since a lwsocket is associated with a specific channel, one cannot use it with other channels or for general system calls, such as `sendmsg()`. In cases where applications need the full generality of file descriptors, MegaPipe provides a fall-back API function to convert a lwsocket into a regular file descriptor.

### 3.4.3 System Call Batching

Recent research efforts report that system calls are expensive not only due to the cost of mode switching, but also because of the negative effect on cache locality in both user and kernel space [35]. To amortize system call costs, MegaPipe batches multiple I/O requests and their completion notifications into a single system call. The key observation here is that batching can exploit connection-level parallelism, extracting multiple independent requests and notifications from concurrent connections.

Batching is transparently done by the MegaPipe user-level library for both directions user $\rightarrow$ kernel and kernel $\rightarrow$ user. Application programmers need not be aware of batching. Instead, application threads issue one request at a time, and the user-level library accumulates them. When *i)* the number of accumulated requests reaches the batching threshold, *ii)* there are not any more pending completion events from the kernel, or *iii)* the application explicitly asks to flush, then the collected requests are flushed to the kernel in a batch through the channel. Similarly, application threads dispatch a completion notification from the user-level library one by one. When the user-level library has no more completion notifications to feed the application thread, it fetches multiple pending notifications from kernel in a batch. We set the default batching threshold to 32 (adjustable), as we found that the marginal performance gain beyond that point is negligible.

## 3.5 API

The MegaPipe user-level library provides a set of API functions to hide the complexity of batching and the internal implementation details. Table 1 presents a partial list of MegaPipe API functions. Due to lack of space, we highlight some interesting aspects of some functions rather than enumerating all of them.

The application associates a handle (either a regular file descriptor or a lwsocket) with the specified channel with `mp_register()`. All further I/O commands and completion notifications for the registered handle are done through only the associated channel. A cookie, an opaque pointer for developer use, is also passed to the kernel with handle registration. This cookie is attached in the completion events for the handle, so the application can easily identify which handle fired each event. The application calls `mp_unregister()` to end the membership. Once unregistered, the application can continue to use the regular FD with general system calls. In contrast, lwsockets

| Function | Parameters | Description |
|---|---|---|
| `mp_create()` | | Create a new MegaPipe channel instance. |
| `mp_register()` | channel, fd, cookie, cpu_mask | Create a MegaPipe handle for the specified file descriptor (either regular or lightweight) in the given channel. If a given file descriptor is a listening socket, an optional CPU mask parameter can be used to designate the set of CPU cores which will respond to incoming connections for that handle. |
| `mp_unregister()` | handle | Remove the target handle from the channel. All pending completion notifications for the handle are canceled. |
| `mp_accept()` | handle, count, is_lwsocket | Accept one or more new connections from a given listening handle asynchronously. The application specifies whether to accept a connection as a regular socket or a lwsocket. The completion event will report a new FD/lwsocket and the number of pending connections in the accept queue. |
| `mp_read()` `mp_write()` | handle, buf, size | Issue an asynchronous I/O request. The completion event will report the number of bytes actually read/written. |
| `mp_disconnect()` | handle | Close a connection in a similar way with `shutdown()`. It does not deallocate or unregister the handle. |
| `mp_dispatch()` | channel, timeout | Retrieve a single completion notification for the given channel. If there is no pending notification event, the call blocks until the specified timer expires. |

**Table 1:** MegaPipe API functions (not exhaustive).

are immediately deallocated from the kernel memory.

When a listening TCP socket is registered with the `cpu_mask` parameter, MegaPipe internally spawns an accept queue for incoming connections on the specified set of CPU cores. The original listening socket (now responsible for the remaining CPU cores) can be registered to other MegaPipe channels with a disjoint set of cores – so each thread can have a completely partitioned view of the listening socket.

`mp_read()` and `mp_write()` issue asynchronous I/O commands. The application should not use the provided buffer for any other purpose until the completion event, as the ownership of the buffer has been delegated to the kernel, like in other asynchronous I/O APIs. The completion notification is fired when the I/O is actually completed, i.e., all data has been copied from the receive queue for read or copied to the send queue for write. In adapting nginx and memcached, we found that vectored I/O operations (multiple buffers for a single I/O operation) are helpful for optimal performance. For example, the unmodified version of nginx invokes the `writev()` system call to transmit separate buffers for a HTTP header and body at once. MegaPipe supports the counterpart, `mp_writev()`, to avoid issuing multiple `mp_write()` calls or aggregating scattered buffers into one contiguous buffer.

`mp_dispatch()` returns one completion event as a `struct mp_event`. This data structure contains: *i)* a completed command type (e.g., read/write/accept/etc.), *ii)* a cookie, *iii)* a result field that indicates success or failure (such as broken pipe or connection reset) with the corresponding `errno` value, and *iv)* a `union` of command-specific return values.

Listing 1 presents simplified pseudocode of a ping-pong server to illustrate how applications use MegaPipe. An application thread initially creates a MegaPipe channel and registers a listening socket (`listen_sd` in this example) with `cpu_mask` 0x01 (first bit is set) which means that the handle is only interested in new connections established on the first core (core 0). The application then invokes `mp_accept()` and is ready to accept new connections. The body of the event loop is fairly simple; given an event, the server performs any appropriate tasks (barely anything in this ping-pong example) and then fires new I/O operations.

```
ch = mp_create()
handle = mp_register(ch, listen_sd, mask=0x01)
mp_accept(handle)

while true:
  ev = mp_dispatch(ch)
  conn = ev.cookie
  if ev.cmd == ACCEPT:
    mp_accept(conn.handle)
    conn = new Connection()
    conn.handle = mp_register(ch, ev.fd,
        cookie=conn)
    mp_read(conn.handle, conn.buf, READSIZE)
  elif ev.cmd == READ:
    mp_write(conn.handle, conn.buf, ev.size)
  elif ev.cmd == WRITE:
    mp_read(conn.handle, conn.buf, READSIZE)
  elif ev.cmd == DISCONNECT:
    mp_unregister(ch, conn.handle)
    delete conn
```

**Listing 1:** Pseudocode for ping-pong server event loop

### 3.6 Discussion: Thread-Based Servers

The current MegaPipe design naturally fits event-driven servers based on callback or event-loop mechanisms [32, 40]. We mostly focus on event-driven servers in this work. On the other hand, MegaPipe is also applicable to thread-based servers, by having one channel for each thread, thus each connection. In this case the application cannot take advantage of batching (§3.4.3), since batching exploits the parallelism of independent connections that are

multiplexed through a channel. However, the application still can benefit from partitioning (§3.4.1) and lwsocket (§3.4.2) for better scalability on multi-core servers.

There is an interesting spectrum between pure event-driven servers and pure thread-based servers. Some frameworks expose thread-like environments to user applications to retain the advantages of thread-based architectures, while looking like event-driven servers to the kernel to avoid the overhead of threading. Such functionality is implemented in various ways: lightweight user-level threading [23, 39], closures or coroutines [4, 18, 28], and language runtime [14]. Those frameworks intercept I/O calls issued by user threads to keep the kernel thread from blocking, and manage the outstanding I/O requests with polling mechanisms, such as `epoll`. These frameworks can leverage MegaPipe for higher network I/O performance without requiring modifications to applications themselves. We leave the evaluation of effectiveness of MegaPipe for these frameworks as future work.

# 4 Implementation

We begin this section with how we implemented MegaPipe in the Linux kernel and the associated user-level library. To verify the applicability of MegaPipe, we show how we adapted two applications (memcached and nginx) to benefit from MegaPipe.

## 4.1 MegaPipe API Implementation

As briefly described in §3.3, MegaPipe consists of two parts: the kernel module and the user-level library. In this section, we denote them by MP-K and MP-L, respectively, for clear distinction between the two.

**Kernel Implementation**: MP-K interacts with MP-L through a special device, `/dev/megapipe`. MP-L opens this file to create a channel, and invokes `ioctl()` system calls on the file to issue I/O requests and dispatch completion notifications for that channel.

MP-K maintains a set of handles for both regular FDs and lwsockets in a red-black tree[4] for each channel. Unlike a per-process file table, each channel is only accessed by one thread, avoiding data sharing between threads (thus cores). MP-K identifies a handle by an integer unique to the owning channel. For regular FDs, the existing integer value is used as an identifier, but for lwsockets, an integer of $2^{30}$ or higher value is issued to distinguish lwsockets from regular FDs. This range is used since it is unlikely to conflict with regular FD numbers, as the POSIX standard allocates the lowest unused integer for FDs [6].

[4]It was mainly for ease of implementation, as Linux provides the template of red-black trees. We have not yet evaluated alternatives, such as a hash table, which supports $O(1)$ lookup rather than $O(logN)$.

MP-K currently supports the following file types: sockets, pipes, FIFOs, signals (via `signalfd`), and timers (via `timerfd`). MP-K handles asynchronous I/O requests differently depending on the file type. For sockets (such as TCP, UDP, and UNIX domain), MegaPipe utilizes the native callback interface, which fires upon state changes, supported by kernel sockets for optimal performance. For other file types, MP-K internally emulates asynchronous I/O with `epoll` and non-blocking VFS operations within kernel. MP-K currently does not support disk files, since the Linux file system does not natively support asynchronous or non-blocking disk I/O, unlike other modern operating systems. To work around this issue, we plan to adopt a lightweight technique presented in FlexSC [35] to emulate asynchronous I/O. When a disk I/O operation is about to block, MP-K can spawn a new thread on demand while the current thread continues.

Upon receiving batched I/O commands from MP-L through a channel, MP-K first examines if each request can be processed immediately (e.g., there is pending data in the TCP receive queue, or there is free space in the TCP send queue). If so, MP-K processes the request and issues a completion notification immediately, without incurring the callback registration or `epoll` overhead. This idea of opportunistic shortcut is adopted from LAIO [22], where the authors claim that the 73–86% of I/O operations are readily available. For I/O commands that are not readily available, MP-K needs some bookkeeping; it registers a callback to the socket or declares an epoll interest for other file types. When MP-K is notified that the I/O operation has become ready, it processes the operation.

MP-K enqueues I/O completion notifications in the per-channel event queue. Those notifications are dispatched in a batch upon the request of MP-L. Each handle maintains a linked list to its pending notification events, so that they can be easily removed when the handle is unregistered (and thus not of interest anymore).

We implemented MP-K in the Linux 3.1.3 kernel with 2,200 lines of code in total. The majority was implemented as a Linux kernel module, such that the module can be used for other Linux kernel versions as well. However, we did have to make three minor modifications (about 400 lines of code of the 2,200) to the Linux kernel itself, due to the following issues: *i)* we modified `epoll` to expose its API to not only user space but also to MP-K; *ii)* we modified the Linux kernel to allow multiple sockets (partitioned) to listen on the same address/port concurrently, which traditionally is not allowed; and *iii)* we also enhanced the socket lookup process for incoming TCP handshake packets to consider `cpu_mask` when choosing a destination listening socket among a partitioned set.

**User-Level Library**: MP-L is essentially a simple wrap-

| Application | Total | Changed |
|---|---|---|
| memcached | 9442 | 602 (6.4%) |
| nginx | 86774 | 447 (0.5%) |

**Table 2:** Lines of code for application adaptations

per of the kernel module, and it is written in about 400 lines of code. MP-L performs two main roles: *i)* it transparently provides batching for asynchronous I/O requests and their completion notifications, *ii)* it performs communication with MP-K via the `ioctl()` system call.

The current implementation uses copying to transfer commands (24 B for each) and notifications (40 B for each) between MP-L and MP-K. This copy overhead, roughly 3–5% of total CPU cycles (depending on workloads) in our evaluation, can be eliminated with virtual memory mapping for the command/notification queues, as introduced in Mach Port [11]. We leave the implementation and evaluation of this idea as future work.

### 4.2 Application Integration

We adapted two popular event-driven servers, memcached 1.4.13 [3] (an in-memory key-value store) and nginx 1.0.15 [37] (a web server), to verify the applicability of MegaPipe. As quantitatively indicated in Table 2, the code changes required to use MegaPipe were manageable, on the order of hundreds of lines of code. However, these two applications presented different levels of effort during the adaptation process. We briefly introduce our experiences here, and show the performance benefits in Section 5.

**memcached**: memcached uses the libevent [30] framework which is based on the readiness model (e.g., `epoll` on Linux). The server consists of a main thread and a collection of worker threads. The main thread accepts new client connections and distributes them among the worker threads. The worker threads run event loops which dispatch events for client connections.

Modifying memcached to use MegaPipe in place of libevent involved three steps[5]:

*(1) Decoupling from libevent*: We began by removing libevent-specific data structures from memcached. We also made the drop-in replacement of `mp_dispatch()` for the libevent event dispatch loop.

*(2) Parallelizing accept*: Rather than having a single thread that accepts all new connections, we modified worker threads to accept connections in parallel by partitioning the shared listening socket.

*(3) State machine adjustment*: Finally, we replaced calls

[5]In addition, we pinned each worker thread to a CPU core for the MegaPipe adaptation, which is considered a best practice and is necessary for MegaPipe. We made the same modification to stock memcached for a fair comparison.

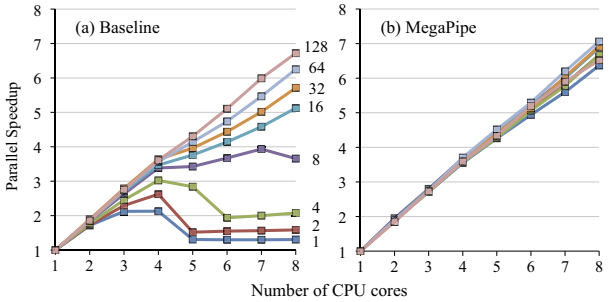



**Figure 3:** Comparison of parallel speedup for varying numbers of transactions per connection (labeled) over a range of CPU cores (x-axis) with 64 B messages.

to `read()` with `mp_read()` and calls to `sendmsg()` with `mp_writev()`. Due to the semantic gap between the readiness model and the completion notification model, each state of the memcached state machine that invokes a MegaPipe function was split into two states: actions prior to a MegaPipe function call, and actions that follow the MegaPipe function call and depend on its result. We believe this additional overhead could be eliminated if memcached did not have the strong assumption of the readiness model.

**nginx**: Compared to memcached, nginx modifications were much more straightforward due to three reasons: *i)* the custom event-driven I/O of nginx does not use an external I/O framework that has a strong assumption of the readiness model, such as libevent [30]; *ii)* nginx was designed to support not only the readiness model (by default with `epoll` in Linux), but also the completion notification model (for POSIX AIO [6] and signal-based AIO), which nicely fits with MegaPipe; and *iii)* all worker processes already accept new connections in parallel, but from the shared listening socket.

nginx has an extensible *event module* architecture, which enables easy replacement for its underlying event-driven mechanisms. Under this architecture, we implemented a MegaPipe event module and registered `mp_read()` and `mp_writev()` as the actual I/O functions. We also adapted the worker threads to accept new connections from the partitioned listening socket.

## 5 Evaluation

We evaluated the performance gains yielded by MegaPipe both through a collection of microbenchmarks, akin to those presented in §2.2, and a collection of application-level macrobenchmarks. Unless otherwise noted, all benchmarks were completed with the same experimental setup (same software versions and hardware platforms as described in §2.2.

| | Number of transactions per connection | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| +P | 211.6 | 207.5 | 181.3 | 83.5 | 38.9 | 29.5 | 17.2 | 8.8 |
| P +B | 18.8 | 22.8 | 72.4 | 44.6 | 31.8 | 30.4 | 27.3 | 19.8 |
| PB +L | 352.1 | 230.5 | 79.3 | 22.0 | 9.7 | 2.9 | 0.4 | 0.1 |
| Total | 582.4 | 460.8 | 333.1 | 150.1 | 80.4 | 62.8 | 45.0 | 28.7 |

**Table 3:** Accumulation of throughput improvement (%) over baseline, from three contributions of MegaPipe.

### 5.1 Microbenchmarks

The purpose of the microbenchmark results is three-fold. First, utilization of the same benchmark strategy as in §2 allows for direct evaluation of the low-level limitations we previously highlighted. Figure 1 shows the performance of MegaPipe measured for the same experiments. Second, these microbenchmarks give us the opportunity to measure an upper-bound on performance, as the minimal benchmark program effectively rules out any complex effects from application-specific behaviors. Third, microbenchmarks allow us to illuminate the performance contributions of each of MegaPipe's individual design components.

We begin with the impact of MegaPipe on multi-core scalability. Figure 3 provides a side-by-side comparison of parallel speedup (compared to the single core case of each) for a variety of transaction lengths. The baseline case on the left clearly shows that the scalability highly depends on the length of connections. For short connections, the throughput stagnates as core count grows due to the serialization at the shared accept queue, then suddenly collapses with more cores. We attribute the performance collapse to increased cache congestion and non-scalable locks [21]; note that the connection establishment process happens more frequently with short flows in our test, increasing the level of contention.

In contrast, the throughput of MegaPipe scales almost linearly regardless of connection length, showing speedup of 6.4 (for single-transaction connections) or higher. This improved scaling behavior of MegaPipe is mostly from the multi-core related optimizations techniques, namely partitioning and lwsocket. We observed similar speedup without batching, which enhances per-core throughput.

In Table 3, we present the incremental improvements (in percent over baseline) that Partitioning (P), Batching (B), and lwsocket (L) contribute to overall throughput, by accumulating each technique in that order. In this experiment, we used all eight cores, with 64 B messages (1 KiB messages yielded similar results). Both partitioning and lwsocket significantly improve the throughput of short connections, and their performance gain diminishes for longer connections since the both techniques act only at the connection establishment stage. For longer connections (not shown in the table), the gain from batching converged around 15%. Note that the case with partitioning alone (+P in the table) can be seen as *sockets with Affinity-Accept* [33], as the both address the shared accept queue and connection affinity issues. lwsocket further contributes the performance of short connections, helping to achieve near-linear scalability as shown in Figure 3(b).

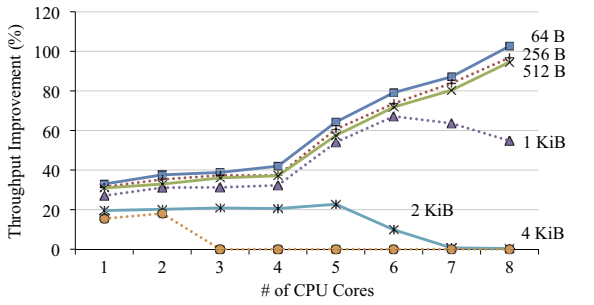



**Figure 4:** Relative performance improvement for varying message sizes over a range of CPU cores.

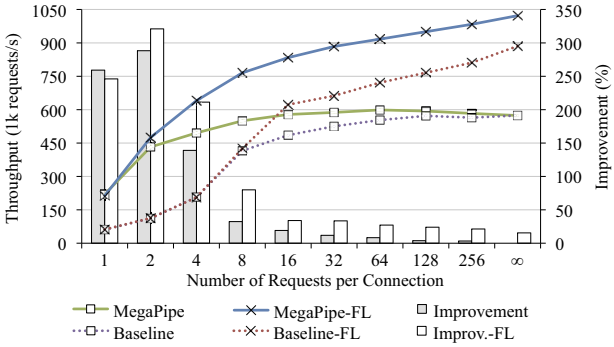



**Figure 5:** memcached throughput comparison with eight cores, by varying the number of requests per connection. $\infty$ indicates persistent connections. Lines with "X" markers (-FL) represent fine-grained-lock-only versions.

Lastly, we examine how the improvement changes by varying message sizes. Figure 4 depicts the relative throughput improvement, measured with 10-transaction connections. For the single-core case, where the improvement comes mostly from batching, MegaPipe outperforms the baseline case by 15–33%, showing higher effectiveness for small ($\leq$ 1 KiB) messages. The improvement goes higher as we have five or more cores, since the baseline case experiences more expensive off-chip cache and remote memory access, while MegaPipe effectively mitigates them with partitioning and lwsocket. The degradation of relative improvement from large messages with many cores reflects that the server was able to saturate the 10 G link. MegaPipe saturated the link with seven, five, and three cores for 1, 2, and 4 KiB messages, respectively. The baseline Linux saturated the link with seven and three cores for 2 and 4 KiB messages, respectively.

### 5.2 Macrobenchmark: memcached

We perform application-level macrobenchmarks of memcached, comparing the baseline performance to that of memcached adapted for MegaPipe as previously described. For baseline measurements, we used a patched[6] version of the stock memcached 1.4.13 release.

We used the `memaslap` [12] tool from libmemcached 1.0.6 to perform the benchmarks. We patched `memaslap` to accept a parameter designating the maximum number of requests to issue for a given TCP connection (upon which it closes the connection and reconnects to the server). Note that the typical usage of memcached is to use persistent connections to servers or UDP sockets, so the performance result from short connections may not be representative of memcached; rather, it should be interpreted as what-if scenarios for event-driven server applications with non-persistent connections.

The key-value workload used during our tests is the default `memaslap` workload: 64 B keys, 1 KiB values, and a get/set ratio of 9:1. For these benchmarks, each of three client machines established 256 concurrent connections to the server. On the server side, we set the memory size to 4 GiB. We also set the initial hash table size to $2^{22}$ (enough for 4 GiB memory with 1 KiB objects), so that the server would not exhibit performance fluctuations due to dynamic hash table expansion during the experiments.

Figure 5 compares the throughput between the baseline and MegaPipe versions of memcached (we discuss the "-FL" versions below), measured with all eight cores. We can see that MegaPipe greatly improves the throughput for short connections, mostly due to partitioning and lwsocket as we confirmed with the microbenchmark. However, the improvement quickly diminishes for longer connections, and for persistent connections, MegaPipe does not improve the throughput at all. Since the MegaPipe case shows about 16% higher throughput for the single-core case (not shown in the graph), it is clear that there is a performance-limiting bottleneck for the multi-core case. Profiling reveals that spin-lock contention takes roughly 50% of CPU cycles of the eight cores, highly limiting the scalability.

In memcached, normal get/set operations involve two locks: `item_locks` and a global lock `cache_lock`. The fine-grained `item_locks` (the number is dynamic, 8,192 locks on eight cores) keep the consistency of the object store from concurrent accesses by worker threads. On the other hand, the global `cache_lock` ensures that the hash table expansion process by the *maintenance thread* does not interfere with worker threads. While this global lock is inherently not scalable, it is unnecessary for our experiments since we configured the hash table expansion to not happen by giving a sufficiently large initial size.

[6]We discovered a performance bug in the stock memcached release as a consequence of unfairness towards servicing new connections, and we corrected this fairness bug.

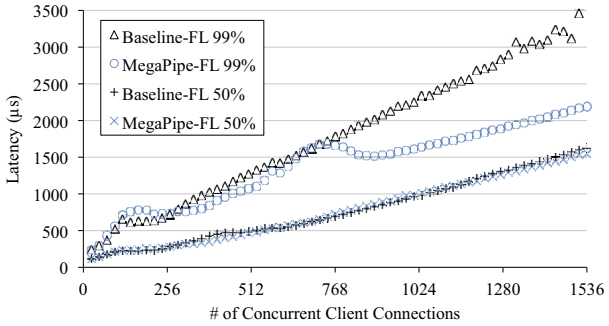



**Figure 6:** 50th and 99th percentile memcached latency.

We conducted experiments to see what would happen if we rule out the global lock, thus relying on the fine-grained locks (`item_locks`) only. We provide the results (with the suffix "-FL") also in Figure 5. Without the global lock, the both MegaPipe and baseline cases perform much better for long or persistent connections. For the persistent connection case, batching improved the throughput by 15% (note that only batching among techniques in §3 affects the performance of persistent connections). We can conclude two things from these experiments. First, MegaPipe improves the throughput of applications with short flows, and the improvement is fairly insensitive to the scalability of applications themselves. Second, MegaPipe might not be effective for poorly scalable applications, especially with long connections.

Lastly, we discuss how MegaPipe affects the latency of memcached. One potential concern with latency is that MegaPipe may add additional delay due to batching of I/O commands and notification events. To study the impact of MegaPipe on latency, we measured median and tail (99th percentile) latency observed by the clients, with varying numbers of persistent connections, and plotted these results in Figure 6. The results show that MegaPipe does not adversely affect the median latency. Interestingly, for the tail latency, MegaPipe slightly increases it with low concurrency (between 72–264) but greatly reduces it with high concurrency ($\geq$ 768). We do not fully understand these tail behaviors yet. One likely explanation for the latter is that batching becomes more effective with high concurrency; since that batching exploits parallelism from independent connections, high concurrency yields larger batch sizes.

In this paper, we conduct all experiments with the interrupt coalescing feature of the NIC. We briefly describe the impact of disabling it, to investigate if MegaPipe favorably or adversely interfere with interrupt coalescing. When disabled, the server yielded up to 50$\mu$s (median)

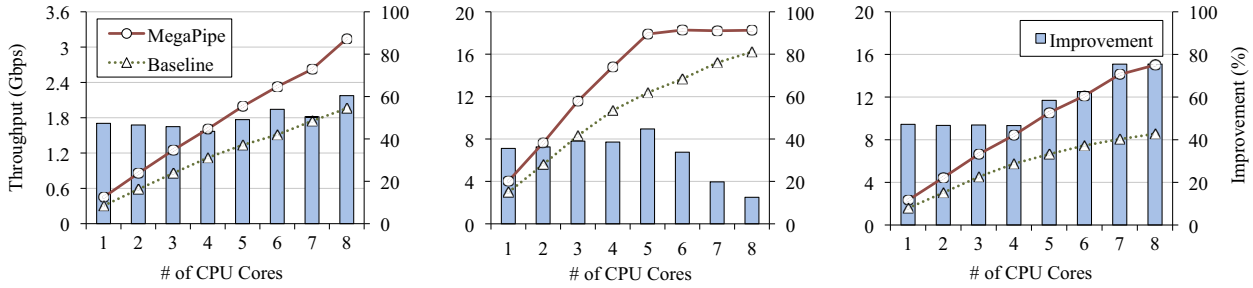



**Figure 7:** Evaluation of nginx throughput for the (a) SpecWeb, (b) Yahoo, and (c) Yahoo/2 workloads.

and 200$\mu$s (tail) lower latency with low concurrency (thus underloaded). On the other hand, beyond near saturation point, disabling interrupt coalescing incurred significantly higher latency due to about 30% maximum throughput degradation, which causes high queueing delay. We observed these behaviors for both MegaPipe and baseline; we could not find any MegaPipe-specific behavior with interrupt coalescing in our experiments.

### 5.3 Macrobenchmark: nginx

Unlike memcached, the architecture of nginx is highly scalable on multi-core servers. Each worker process has an independent address space, and nothing is shared by the workers, so the performance-critical path is completely lockless. The only potential factor that limits scalability is the interface between the kernel and user, and we examine how MegaPipe improves the performance of nginx with such characteristics.

For the nginx HTTP benchmark, we conduct experiments with three workloads with static content, namely `SpecWeb`, `Yahoo`, and `Yahoo/2`. For all workloads, we configured nginx to serve files from memory rather than disks, to avoid disks being a bottleneck. We used weighttp[7] as a workload generator, and we modified it to support variable number of requests per connection.

**SpecWeb**: We test the same HTTP workload used in Affinity-Accept [33]. In this workload, each client connection initiates six HTTP requests. The content size ranges from 30 to 5,670 B (704 B on average), which is adopted from the static file set of SpecWeb 2009 Support Workload [9].

**Yahoo**: We used the HTTP trace collected from the Yahoo! CDN [13]. In this workload, the number of HTTP requests per connection ranges between 1 and 1,597. The distribution is heavily skewed towards short connections (98% of connections have ten or less requests, 2.3 on average), following the Zipf-like distribution. Content sizes range between 1 B and 253 MiB (12.5 KiB on average). HTTP responses larger than 60 KiB contribute roughly 50% of the total traffic.

**Yahoo/2**: Due to the large object size of the `Yahoo` workload, MegaPipe with only five cores saturates the two 10G links we used. For the `Yahoo/2` workload, we change the size of all files by half, to avoid the link bottleneck and observe the multi-core scalability behavior more clearly.

Web servers can be seen as one of the most promising applications of MegaPipe, since typical HTTP connections are short and carry small messages [13]. We present the measurement result in Figure 7 for each workload. For all three workloads, MegaPipe significantly improves the performance of both single-core and multi-core cases. MegaPipe with the `Yahoo/2` workload, for instance, improves the performance by 47% (single core) and 75% (eight cores), with a better parallel speedup (from 5.4 to 6.5) with eight cores. The small difference of improvement between the `Yahoo` and `Yahoo/2` cases, both of which have the same connection length, shows that MegaPipe is more beneficial with small message sizes.

## 6 Related Work

**Scaling with Concurrency**: Stateless event multiplexing APIs, such as `select()` or `poll()`, scale poorly as the number of concurrent connections grows since applications must declare the entire *interest set* of file descriptors to the kernel repeatedly. Banga et al. address this issue by introducing stateful interest sets with incremental updates [16], and we follow the same approach in this work with `mp_(un)register()`. The idea was realized with with `epoll` [8] in Linux (also used as the baseline in our evaluation) and `kqueue` [29] in FreeBSD. Note that this scalability issue in event delivery is orthogonal to the other scalability issue in the kernel: VFS overhead, which is addressed by lwsocket in MegaPipe.

**Asynchronous I/O**: Like MegaPipe, Lazy Asynchronous I/O (LAIO) [22] provides an interface with completion notifications, based on "continuation". LAIO achieves low overhead by exploiting the fact that most I/O operations do not block. MegaPipe adopts this idea, by processing non-blocking I/O operations immediately as explained

[7] `http://redmine.lighttpd.net/projects/weighttp/wiki`

in §4.1.

POSIX AIO defines functions for asynchronous I/O in UNIX [6]. POSIX AIO is not particularly designed for sockets, but rather, general files. For instance, it does not have an equivalent of `accept()` or `shutdown()`. Interestingly, it also supports a form of I/O batching: `lio_listio()` for AIO commands and `aio_suspend()` for their completion notifications. This batching must be explicitly arranged by programmers, while MegaPipe supports transparent batching.

Event Completion Framework [1] in Solaris and `kqueue` [29] in BSD expose similar interfaces (completion notification through a completion port) to MegaPipe (through a channel), when they are used in conjunction with POSIX AIO. These APIs associate individual AIO operations, not handles, with a channel to be notified. In contrast, a MegaPipe handle is a member of a particular channel for explicit partitioning between CPU cores. Windows IOCP [10] also has the concept of completion port and membership of handles. In IOCP, I/O commands are not batched, and handles are still shared by all CPU cores, rather than partitioned as lwsockets.

**System Call Batching**: While MegaPipe's batching was inspired by FlexSC [35, 36], the main focus of MegaPipe is I/O, not general system calls. FlexSC batches synchronous system call requests via asynchronous channels (syscall pages), while MegaPipe batches asynchronous I/O requests via synchronous channels (with traditional exception-based system calls). Loose coupling between system call invocation and its execution in FlexSC may lead poor cache locality on multi-core systems; for example, the `send()` system call invoked from one core may be executed on another, inducing expensive cache migration during the copy of the message buffer from user to kernel space. Compared with FlexSC, MegaPipe explicitly partitions cores to make sure that all processing of a flow is contained within a single core.

netmap [34] extensively use batching to amortize the cost of system calls, for high-performance, user-level packet I/O. MegaPipe follows the same approach, but its focus is generic I/O rather than raw sockets for low-level packet I/O.

**Kernel-Level Network Applications**: Some network applications are partly implemented in the kernel, tightly coupling performance-critical sections to the TCP/IP stack [25]. While this improves performance, it comes at a price of limited security, reliability, programmability, and portability. MegaPipe gives user applications lightweight mechanisms to interact with the TCP/IP stack for similar performance advantages, while retaining the benefits of user-level programming.

**Multi-Core Scalability**: Past research has shown that partitioning cores is critical for linear scalability of network I/O on multi-core systems [19, 20, 33, 38]. The main ideas are to maintain flow affinity and minimize unnecessary sharing between cores. In §3.4.1, we addressed the similarities and differences between Affinity-Accept [33] and MegaPipe. In [20], the authors address the scalability issues in VFS, namely inode and dentry, in the general context. We showed in §3.4.2 that the VFS overhead can be completely bypassed for network sockets in most cases.

The Chronos [26] work explores the case of direct coupling between NIC queues and application threads, in the context of multi-queue NIC and multi-core CPU environments. Unlike MegaPipe, Chronos bypasses the kernel, exposing NIC queues directly to user-space memory. While this does avoid in-kernel latency/scalability issues, it also loses the generality of TCP connection handling which is traditionally provided by the kernel.

**Similarities in Abstraction**: Common Communication Interface (CCI) [15] defines a portable interface to support various transports and network technologies, such as Infiniband and Cray's Gemini. While CCI and MegaPipe have different contexts in mind (user-level message-passing in HPC vs. general sockets via the kernel network stack), both have very similar interfaces. For example, CCI provides the *endpoint* abstraction as a channel between a virtual network instance and an application. Asynchronous I/O commands and notifications are passed through the channel with similar API semantics (e.g., `cci_get_event()`/`cci_send()` corresponding to `mp_dispatch()`/`mp_write()`).

The channel abstraction of MegaPipe shares some similarities with Mach port [11] and other IPC mechanisms in microkernel designs, as it forms queues for typed messages (I/O commands and notifications in MegaPipe) between subsystems. Especially, Barrelfish [17] leverages message passing (rather than sharing) based on event-driven programming model to solve scalability issues, while its focus is mostly on inter-core communication rather than strict intra-core communication in MegaPipe.

## 7 Conclusion

Message-oriented network workloads, where connections are short and/or message sizes are small, are CPU-intensive and scale poorly on multi-core systems with the BSD Socket API. In this paper, we introduced MegaPipe, a new programming interface for high-performance networking I/O. MegaPipe exploits many performance optimization opportunities that were previously hindered by existing network API semantics, while being still simple and applicable to existing event-driven servers with moderate efforts. Evaluation through microbenchmarks,

memcached, and nginx showed significant improvements, in terms of both single-core performance and parallel speedup on an eight-core system.

## Acknowledgements

We thank Luca Niccolini, members of NetSys Lab at UC Berkeley, anonymous OSDI reviewers, and our shepherd Jeffrey Mogul for their help and invaluable feedback. The early stage of this work was done in collaboration with Keon Jang, Sue Moon, and KyoungSoo Park, when the first author was affiliated with KAIST.

## References

[1] Event Completion Framework for Solaris. `http://developers.sun.com/solaris/articles/event_completion.html`.

[2] Intel 10 Gigabit Ethernet Adapter. `http://e1000.sourceforge.net/`.

[3] memcached - a distributed memory object caching system. `http://memcached.org/`.

[4] Node.js: an event-driven I/O server-side JavaScript environment. `http://nodejs.org`.

[5] Receive-Side Scaling. `http://www.microsoft.com/whdc/device/network/ndis_rss.mspx`, 2008.

[6] The Open Group Base Specifications Issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2008.

[7] Intel 8259x 10G Ethernet Controller. Intel 82599 10 GbE Controller Datasheet, 2009.

[8] epoll - I/O event notification facility. `http://www.kernel.org/doc/man-pages/online/pages/man4/epoll.4.html`, 2010.

[9] SPECweb2009 Release 1.20 Support Workload Design Document. `http://www.spec.org/web2009/docs/design/SupportDesign.html`, 2010.

[10] Windows I/O Completion Ports. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx`, 2012.

[11] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation For UNIX Development. In *Proc. of USENIX Summer* (1986).

[12] AKER, B. memaslap: Load testing and benchmarking a server. `http://docs.libmemcached.org/memaslap.html`, 2012.

[13] AL-FARES, M., ELMELEEGY, K., REED, B., AND GASHINSKY, I. Overclocking the Yahoo! CDN for Faster Web Page Loads. In *Proc. of ACM IMC* (2011).

[14] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, second ed. Prentice Hall, 1996.

[15] ATCHLEY, S., DILLOW, D., SHIPMAN, G., GEOFFRAY, P., SQUYRES, J. M., BOSILCA, G., AND MINNICH, R. The Common Communication Interface (CCI). In *Proc. of IEEE HOTI* (2011).

[16] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proc. of USENIX ATC* (1999).

[17] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. of ACM SOSP* (2009).

[18] BILENKO, D. gevent: A coroutine-based network library for Python. `http://www.gevent.org/`.

[19] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proc. of USENIX OSDI* (2008).

[20] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proc. of USENIX OSDI* (2010).

[21] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable locks are dangerous. In *Proc. of the Linux Symposium* (July 2012).

[22] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy Asynchronous I/O For Event-Driven Servers. In *Proc. of USENIX ATC* (2004).

[23] ENGELSCHALL, R. S. Portable Multithreading - The Signal Stack Trick for User-Space Thread Creation. In *Proc. of USENIX ATC* (2000).

[24] HERBERT, T. RPS: Receive Packet Steering. `http://lwn.net/Articles/361440/`, 2009.

[25] JOUBERT, P., KING, R. B., NEVES, R., RUSSINOVICH, M., AND TRACEY, J. M. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *Proc. of USENIX ATC* (2001).

[26] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND AMIN, V. Reducing Datacenter Application Latency with Endhost NIC Support. Tech. Rep. CS2012-0977, UCSD, April 2012.

[27] KLEIMAN, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of USENIX Summer* (1986).

[28] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events Can Make Sense. In *Proc. of USENIX ATC* (2007).

[29] LEMON, J. Kqueue: A generic and scalable event notification facility. In *Proc. of USENIX ATC* (2001).

[30] MATHEWSON, N., AND PROVOS, N. libevent - an event notification library. `http://libevent.org`.

[31] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND RYAN, S. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[32] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An Efficient and Portable Web Server. In *Proc. of USENIX ATC* (1999).

[33] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving Network Connection Locality on Multicore Systems. In *Proc. of ACM EuroSys* (2012).

[34] RIZZO, L. netmap: a novel framework for fast packet I/O. In *Proc. of USENIX ATC* (2012).

[35] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of USENIX OSDI* (2010).

[36] SOARES, L., AND STUMM, M. Exception-Less System Calls for Event-Driven Servers. In *Proc. of USENIX ATC* (2011).

[37] SYSOEV, I. nginx web server. `http://nginx.org/`.

[38] VEAL, B., AND FOONG, A. Performance Scalability of a Multi-Core Web Server. In *Proc. of ACM/IEEE ANCS* (2007).

[39] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable Threads for Internet Services. In *Proc. of ACM SOSP* (2003).

[40] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of ACM SOSP* (2001).

# DJoin: Differentially Private Join Queries over Distributed Databases

Arjun Narayan
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

## Abstract

In this paper, we study the problem of answering queries about private data that is spread across multiple different databases. For instance, a medical researcher may want to study a possible correlation between travel patterns and certain types of illnesses. The necessary information exists today – e.g., in airline reservation systems and hospital records – but it is maintained by two separate companies who are prevented by law from sharing this information with each other, or with a third party. This separation prevents the processing of such queries, even if the final answer, e.g., a correlation coefficient, would be safe to release.

We present DJoin, a system that can process such distributed queries and can give strong differential privacy guarantees on the result. DJoin can support many SQL-style queries, including joins of databases maintained by different entities, as long as they can be expressed using DJoin's two novel primitives: BN-PSI-CA, a differentially private form of private set intersection cardinality, and DCR, a multi-party combination operator that can aggregate noised cardinalities without compounding the individual noise terms. Our experimental evaluation shows that DJoin can process realistic queries at practical timescales: simple queries on three databases with 15,000 rows each take between 1 and 7.5 hours.

## 1 Introduction

A vast amount of information is constantly accumulating in databases (social networks, hospital records, airline reservation systems, etc.) all around the world. There are many good uses to which this data could potentially be put; however, much of this data is sensitive and cannot safely be released because of privacy concerns. Simple solutions, such as anonymizing or aggregating the data before release, are not reliable; experience with cases like the Netflix prize [3] or the AOL search data [2] shows that such data can sometimes be de-anonymized with auxiliary information [26].

Differential privacy [7] has been proposed as a way to solve this problem. By disallowing certain queries, and by adding a carefully chosen amount of noise to the result of others, it is possible to give a strong upper bound on how much an adversary could learn about an individual person's data, even under worst-case assumptions. Several differentially private query processors, including PINQ [23], Airavat [32], Fuzz [16], and PDDP [6], have been developed and are available today.

However, existing query processors assume either that all the data is available in a *single* database [16, 23, 32] or that distributed queries can be broken into several subqueries that can each be answered using only one of the databases [6, 10, 15, 31]. In practice, this is not necessarily the case. For instance, suppose a medical researcher wanted to study how a certain illness is correlated with travel to a particular region. This data may be available, e.g., in a hospital database $H$ and an airline reservation system $R$, but to determine the correlation, it is necessary to *join the two databases together* – for instance, we must count the individuals who have been treated for the illness (according to $H$) *and* have traveled to the region (according to $R$).

We are not aware of any existing method or query processor that can efficiently support join queries with differential privacy guarantees. Joins cannot be broken into smaller subqueries on individual databases because, in order to match up the same persons' data in the two databases, such queries would have to ask about individual rows, which is exactly what differential privacy is designed to prevent. In principle, one could process joins using secure multi-party computation (MPC) [38], but MPC is only practical for small computational tasks, and differential privacy only works well for large databases. The cost of an entire join under MPC would be truly spectacular.

DJoin, the system we present in this paper, is a solution to this problem. DJoin can support SQL-style queries across multiple databases, including common forms of joins. The key insight behind DJoin is that the distributed parts of many queries can be expressed as intersections of sets or multisets. For instance, we can rewrite the query from above to locally select all patients with the illness from $H$ and all travelers to the relevant region from $R$, then intersect the resulting sets, and finally count the number of elements in the intersection. Not all SQL queries can be rewritten in this way, but

many counting queries can: conjunctions and disjunctions of equality tests directly correspond to unions and intersections of data elements. As we will show, a number of additional operations, such as inequalities and numeric comparisons, can be expressed in terms of multi-set operations.

Protocols for private set operations have been studied by cryptographers for some time [14, 17, 37], but existing solutions compute exact set elements or exact cardinalities, which is not compatible with differential privacy. We present *blinded, noised private set intersection cardinality (BN-PSI-CA)*, an extension of the set-intersection protocol from [17] that supports private noising, as well as *denoise-combine-renoise (DCR)*, an operator that can add or subtract multiple noised subset cardinalities without compounding the corresponding noise terms. DCR relies on MPC to remove the noise terms on its inputs and to re-noise the output, but DCR's complexity grows with the number of parties and not with the number of elements in the sets. For the queries we tried, this step never took more than 20 seconds.

We have implemented and evaluated a prototype of DJoin. Our results show that the costs are substantial but typically feasible. For instance, the elements in a simple two-way join on databases with 32,000 rows each can be evaluated in about 1.8 hours, with 83 MB of traffic, using a single commodity workstation for each database. This is orders of magnitude faster than general MPC. DJoin's cost is too high for interactive use, but it seems practical for applications that can tolerate a certain amount of latency, such as research studies. Our algorithms are easy to parallelize, so the speed could be improved by increasing the number of cores.

To summarize, this paper makes the following four contributions:

- two new primitives, BN-PSI-CA and DCR, for distributed private query processing (Section 4);
- a query planner that rewrites SQL-style queries to take advantage of those two primitives (Section 5);
- the design of DJoin, an engine for distributed, differentially private queries (Section 6); and
- an experimental evaluation of DJoin, based on a prototype implementation (Section 7).

## 2 Related work

DJoin provides differential privacy [7, 8, 9, 11], which is one of the strongest privacy guarantees that have been proposed so far. Alternatives include randomization [1], k-anonymity [34], and l-diversity [21], which are generally less restrictive but can be vulnerable to certain attacks on privacy [12, 20]. Differential privacy offers a provable bound on the amount of information that an attacker can learn about any individual, even with access to auxiliary information.

**Differentially private query processors:** PINQ [23], Airavat [32], and Fuzz [16] are query processors that support differential privacy, but they assume a centralized setting in which a single entity has access to the entire data. We are aware of five solutions for distributed settings [6, 10, 15, 31, 33], but these assume that the data is horizontally partitioned (i.e., each individual's data is completely contained in one of the databases), and that the query can be factored into subqueries that are each local to a single database. For instance, [10] computes queries of the form $\sum_i f(d_i)$, i.e., the sum over all rows $i$ in the database after applying a function $f$ to each row. DJoin's data model is more general: multiple databases may contain data for a given individual, and queries can contain joins. We note that some of the other systems have far more sophisticated query languages, but we speculate that DJoin's rewriting and execution engine could be integrated with existing systems, e.g., with PINQ or Fuzz.

**Private set operations:** The first protocols for private two-party set intersection and set intersection cardinality were proposed by Freedman et al. [14]. Since then, a number of improvements have been proposed; for instance, Kissner and Song [17] extended the protocols to multiple parties, and Vaidya and Clifton [37] reduced the computational overhead. These protocols produce exact results, and are thus not directly suitable for differential privacy. There are specialized protocols for other private multi-party operations, e.g., for decision-tree learning [29], and some of these have been adapted for differential privacy, e.g., [39].

**Computational differential privacy:** The standard definition of differential privacy is information-theoretic, i.e., it holds even against a computationally unbounded adversary. In contrast, DJoin provides computational differential privacy [25]: it relies on a homomorphic cryptosystem and thus depends on certain computational hardness assumptions. Mironov et al. [25] demonstrated a protocol for this model that privately approximates the Hamming distance between two vectors in a two-party setting. This problem is closely related to that of computing the cardinality of set intersections, which is solved by BN-PSI-CA.

**Untrusted servers:** Several existing systems enable clients to use an untrusted server without exposing private information to that server. In SUNDR [19], SPORC [13], and Depot [22], the server provides storage; in CryptDB [30], it implements a database and SQL-style queries. This approach is complementary to ours: DJoin's goal is to reveal *some* useful information about the data it stores, but with an upper bound on how much can be learned about a single individual.

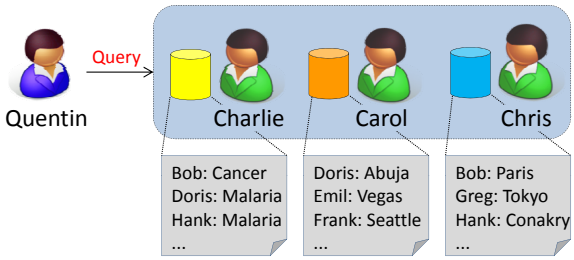



Figure 1: Motivating scenario. Charlie is a physician, and Carol and Chris are travel agents. Quentin would like to know the correlation between treatment for malaria and travel to high-risk areas.

# 3 Background and overview

## 3.1 Motivating scenario

Figure 1 shows our motivating scenario. Charlie, Carol, and Chris each have a database with confidential information about individuals; for instance, Charlie could be a physician, and Carol and Chris could be travel agents. We will refer to these three as the *curators*. Quentin asks a question that combines data from each of the databases; for instance, he might want to know the correlation between treatment for malaria and travel to areas with a high risk of malaria infections. We will refer to Quentin as the *querier*.

Our goal is to build a system that can give an (at least approximate) answer to Quentin's question while offering strong privacy guarantee to the individuals whose data is in the databases. In particular, we would like to establish an upper bound on how much *additional* information any participant of the system (queriers or curators) can learn about any individual in the database. The word 'additional' is crucial here, since the curators each have full access to their respective databases. For instance, since Charlie has treated Bob for cancer, our system cannot prevent him from learning this fact, but it can prevent him from learning whether or not Bob has recently traveled to Paris.

## 3.2 Differential privacy

To formally define the privacy guarantee we want to provide, we rely on *differential privacy* [7]. Differential privacy is a property of randomized queries that take a database as input and return a result that is typically some form of aggregate (such as a number representing a count, a histogram, etc). The database is seen as a collection of rows, and each row contains the data from one individual.

Informally, a randomized function is differentially private if arbitrary changes to a single individual's row (while keeping the other rows constant) result in only statistically insignificant changes in the function's output distribution. Thus, the presence or absence of any individual has a statistically negligible effect. Formally [11], differential privacy is parametrized by a real number $\varepsilon$, which corresponds to the strength of the privacy guarantee; smaller values of $\varepsilon$ yield better privacy. Two databases $b$ and $b'$ are considered *similar*, written $b \sim b'$, if they differ in only one row. We then say that a randomized function $f$ with range $R$ is *$\varepsilon$-differentially private* if, for all possible sets of outputs $S \subseteq R$, and for all similar databases $b, b'$, we have

$$Pr[f(b) \in S] \leq e^{\varepsilon} \cdot Pr[f(b') \in S]$$

That is, when the input database is changed in one row, there is at most a small multiplicative difference ($e^{\varepsilon}$) in the probability of *any* set of outcomes $S$. A slightly weaker variant of this privacy definition is $(\varepsilon, \delta)$-differential privacy [10], where $\delta$ is a bound on the maximum *additive* (not multiplicative) difference between the probabilities of a given output with and without a particular input row.

Practical solutions for achieving differential privacy typically rely on adding a carefully chosen amount of noise to the result. The required amount of noise depends on the *sensitivity* of the query, i.e., how much the result can change in response to changing the data in a single row [11]. More formally, if $q$ is a function that computes the (exact) result of the query and $|q(b) - q(b')| \leq s$ for any pair of similar databases $b \sim b'$, the query is $s$-sensitive, and we can construct an $\varepsilon$-differentially private function $f$ by adding noise to $s$ that is drawn from a Laplace distribution with parameter $\lambda = s/\varepsilon$. This corresponds to the intuition that more sensitive queries need more noise to conceal the contributions of any given individual.

## 3.3 Challenge: Distribution

Answering differentially private queries over a *single* database is a well-studied problem, and several systems [16, 23, 32] are already available for this purpose. In principle, these systems can also be used to answer queries across multiple databases, but this requires that all curators turn over their data to a single trusted entity (e.g., one of the curators), who evaluates the query on their behalf. However, there may not always be a single entity that is sufficiently trusted by all the curators, so it seems useful to have an alternative solution that does not require a trusted entity.

In some cases, distributed queries can be factored into several subqueries that can each be executed on an individual database. For instance, a group of doctors can count the number of male patients in their respective

databases by counting the number of patients in *each* database separately, and then add up the (individually noised) results. This type of distributed query is supported by several existing systems [6, 10, 15, 31, 33]. However, not all queries can be factored in this way. For instance, the above approach will double-count male patients that have been treated by more than one doctor, but a union query (which would avoid this problem) cannot be expressed as a sum of counts. Similarly, any query that involves joining several databases (such as our motivating example) cannot be expressed in this way.

Joins *could* be supported via general-purpose multiparty computation (MPC) [38], but the required runtimes would be gigantic: state-of-the-art MPC solutions, such as FairplayMP [4], need about 10 seconds to evaluate (very simple) functions that can be expressed with 1,024 logic gates. Since the number of gates needed for a join would be at least quadratic in the number of input rows, and since differential privacy only works well for large databases, this approach does not seem practical.

### 3.4 Approach

The key insight behind our solution is that joins are rarely used to compute full cross products of different databases; rather, they are often used to 'match up' elements from different databases. For instance, in our running example, we can first select all the individuals in $R$ who have traveled to the region of interest, then select all the individuals in $H$ who have been treated for the illness, and finally count the number of individuals who appear in both sets. Thus, the problem of privately answering the overall query is reduced to 1) some local operations on each database, and 2) privately computing the cardinality of the intersection of multiple sets. Not all queries can be decomposed in this way, but, as we will show in Section 5, there is a substantial class of queries that can.

Protocols for private multiset operations (such as intersection and union) are available [14, 17, 37], but they tend to compute *exact* sets or set cardinalities. If we naïvely used these algorithms, Charlie could compute the intersection of the set of the malaria patients in his database with the sets of customers in Carol's and Chris' databases who have traveled to high-risk areas, and then add noise in a collaborative fashion [10]. This would prevent Quentin from learning anything other than the (differentially private) output of the query — but Charlie could learn where his patents have traveled, and Carol and Chris could learn which of their customers have been treated for malaria. Hence, our first challenge is to extend these set-intersection operations to support noising between the data curators.

A second challenge arises because some queries involve multiple set operations. If Charlie simply added the two cardinalities together, the noise terms would compound, and thus (unnecessarily) degrade the quality of the overall result. To avoid this problem, we need a way to de-noise, combine, and re-noise intermediate results without compromising privacy.

## 4 Building blocks: BN-PSI-CA and DCR

Next, we describe two key building blocks that enable private processing of distributed queries. Each building block performs only one, very specific operation. In Section 5, we will describe how these building blocks can be used in a larger query plan to answer a variety of different queries.

### 4.1 Background: PSI-CA

Our first building block is related to a primitive called *private set-intersection cardinality (PSI-CA)*, which allows a group of $k$ curators with multisets $S_1, \ldots, S_k$ to privately compute $|\bigcap_i S_i|$, i.e., the (exact) number of elements they have in common, but *not* the specific elements in $\bigcap_i S_i$. PSI-CA is a well-studied primitive [14, 17, 37], albeit not in the context of differential privacy. To explain the intuition, we describe one simple PSI-CA primitive [14] for only two curators with simple sets in the honest-but-curious (HbC) model. The primitive uses a homomorphic encryption scheme that preserves addition and allows multiplication by a constant. Paillier's cryptosystem [28] is an example of a scheme that has this property.

Suppose the two curators are $C_1$ and $C_2$ and their sets are $S_1 := \{x_1, \ldots\}$ and $S_2 := \{y_1, \ldots\}$. $C_1$ defines a polynomial $P(z)$ over a finite field whose roots are his set elements $x_i$:

$$P(z) := (x_1 - z)(x_2 - z) \cdots = \sum_u \alpha_u z^u$$

Next, $C_1$ sends homomorphic encryptions of the coefficients $\alpha_u$ to $C_2$, along with the public key. For each element $y_i \in S_2$, $C_2$ then computes $\mathrm{Enc}(rP(y_i) + 0^+)$, i.e., she evaluates the polynomial at each of her inputs, multiplies each result by a fresh random number $r$, and finally adds a special string $0^+$, e.g., a string of zeroes. Since the cryptosystem is homomorphic, $C_2$ can do this even though she does not know $C_1$'s private key. Finally, $C_2$ sends a random permutation of the results back to $C_1$, who decrypts them and counts the occurrences of the special string $0^+$, which is exactly $|S_1 \bigcap S_2|$.

At first glance, the cost of this algorithm appears to be quadratic: $C_2$ must compute $\mathrm{Enc}(rP(y_i) + 0^+)$ for each of her $|S_2|$ inputs, which involves computing $\mathrm{Enc}(P(y_i))$ along the way. If this is naïvely evaluated as $\mathrm{Enc}(\sum_{u=0}^{|S_1|} \alpha_u y_i^u)$, $C_2$ must multiply each of the $|S_1| + 1$

encrypted coefficients with an unencrypted constant ($y_i^u$), which requires an exponentiation each time, for a total of $O(|S_1| \cdot |S_2|)$ exponentiations. However, [14] describes several optimizations that can reduce this overhead, including an application of Horner's rule and the use of hashing to replace the single high-degree polynomial with several low-degree polynomials. This reduces the computational overhead to $O(|S_1| + |S_2| \ln \ln |S_1|)$ exponentiations.

## 4.2 BN-PSI-CA: Two-party case

The basic PSI-CA primitive is not compatible with differential privacy because $C_1$ learns the *exact*, un-noised size of $|S_1 \bigcap S_2|$; moreover, each curator can learn the size of the other curator's set by observing the number of encrypted coefficients, or encrypted return values, that are received from that curator. However, we can extend the primitive to avoid both problems.

First, we need to make the number of coefficients and return values independent of the set sizes. We can do this by adding some extra elements that cannot appear in either of the sets. As long as we can ensure that $C_1$ and $C_2$ are adding different elements (e.g., by setting some bit to zero on $C_1$ and to one on $C_2$), this will not affect the size of the intersection. In DJoin, we assume that a rough upper bound on the size of each curator's database is known, and we add enough elements to fill up both sets to that upper bound.

Second, we need to add some noise $n$ to the result that is revealed to $C_1$. We observe that $C_2$ can increase the apparent size of the intersection by $n$ if she adds $n$ different[1] encodings of the special string $0^+$. However, to guarantee $\varepsilon$-differential privacy, we would have to draw $n$ from a Laplace distribution $\mathrm{Lap}(1/\varepsilon)$, and this would sometimes yield $n < 0$ – but $C_2$ cannot *remove* encodings of $0^+$ because she does not have $C_1$'s private key, and thus cannot tell them apart from encodings of other values. Instead, we require $C_2$ to draw $n$ from $X_2 + \mathrm{Lap}(1/\varepsilon)$ and we cut $n$ at 0 and $2 \cdot X_2$; thus, $C_2$ can add $n$ encodings of $0^+$ and $2 \cdot X_2 - n$ encodings of a random value to keep the overall size independent of $n$. (Cutting the Laplace distribution can leak a small amount of information when the extremal values are drawn, and thus changes the privacy guarantee to $(\varepsilon, \delta)$-differential privacy [10]; however, by increasing $X_2$, we can make $\delta$ arbitrarily small, at the expense of a higher overhead.) We call the resulting primitive *blinded noised PSI-CA (BN-PSI-CA)*.

Note that at the end, $C_2$ knows the noise term $n$ and $C_1$ the noised cardinality $|S_1 \bigcap S_2| + n$. Thus, if the latter is used in further computations, we have an opportunity to remove the noise again, as long as we can ensure that neither curator learns both values. This prevents the noise terms from compounding, and it enables us to use a *very* high noise level (and thus a low value of $\varepsilon$) because the noise will not affect the final result.

[1] The Paillier cryptosystem can construct many different ciphertexts for the same plaintext.

## 4.3 BN-PSI-CA: Multi-party case

Since Freedman's initial work, cryptographers have considerably extended the range of private multiset operations. For instance, the protocol by Kissner and Song [17] also supports set unions, as well as set intersections with more than two parties, and it is *compositional*: the result of a set union or set intersection can be unioned or intersected with further sets, without decrypting it first. [17] can evaluate any function on multisets that can be described by the following grammar:

$$\Upsilon ::= s \,|\, \Upsilon \cap \Upsilon \,|\, s \cup \Upsilon \,|\, \Upsilon \cup s$$

where $s$ is a multiset that is known to some curator $C_i$.

The protocol from [17] computes $|\bigcap_{i=1,\ldots,k} S_i|$ as follows. First, the $k$ curators use a homomorphic threshold cryptosystem to share a secret key $sk$ amongst themselves, while the corresponding public key $pk$ is known to all curators. Each curator $C_i$ now encrypts a polynomial $P_i$ whose roots are the elements of its local set $S_i$. The encrypted polynomials are then essentially added together, yielding a polynomial $P$ whose roots are the elements in the intersection. Each curator $C_i$ now evaluates $P$ on the elements $e_{ij}$ of his local set $S_i$, yielding values $v_{ij} := P(e_{ij})$; however, recall that, because $sk$ is shared, no individual curator can decrypt the $v_{ij}$. The curators then securely re-randomize and shuffle [27] the $v_{ij}$, such that each curator learns all the $v_{ij}$ but cannot tell which curator it came from. Finally, the curators jointly decrypt the $v_{ij}$. If there are $n$ elements in the intersection, this yields $n \cdot k$ zeroes; hence, each curator can compute the final result by dividing the number of zeroes by $k$.

We can use the same blinding technique as in Section 4.2 to construct a multi-party version of BN-PSI-CA. After computing the $v_{ij}$, but before the shuffle, each curator draws a noise term $n_i$ as above and adds $2 \cdot X_i$ extra values, $n_i$ of which are $0^+$. As above, this adds $\sum_i n_i$ to the resulting cardinality, but the noise can be removed again via DCR, which we discuss next.

## 4.4 DCR: Adding cardinalities

BN-PSI-CA is sufficient to answer queries that require a single distributed multiset operation. However, in Section 5.2 we will see that some queries require multiple operations, and that the result is then a linear combination of the different cardinalities. In principle, we could

designate a single curator $C$ that collects all cardinalities and computes the overall result; however, this would a) compound all the noise terms and thus decrease the quality of the result, and b) reveal all the intermediate results to $C$ and thus (unnecessarily) reveal some private information.

Instead, we can combine the various cardinalities using secure multi-party computation (MPC) [38]. If we have a number of players with private inputs $x_i$ that are each known to only one of the players, MPC allows the players to collectively compute a function $f(x_1, x_2, \ldots)$ *without* revealing the inputs to each other. Even after decades of research, MPC remains impractical for complex functions or large inputs, but modern implementations, such as [4], can process simple functions in a few seconds or less. Thus, while MPC may be too expensive to evaluate the entire query, we can certainly use it to combine a small number of subquery results.

For instance, suppose the query is for $|S_1 \bigcap S_2| + |S_3 \bigcap S_4|$, and that there are four curators involved: $C_1$ and $C_3$ learn the noised results $R_1$ and $R_2$ for the first and the second term, respectively, and $C_2$ and $C_4$ learn the corresponding noise terms $n_1$ and $n_2$. Then we can compute the query result under MPC as

$$q = R_1 + R_2 - (n_1 + n_2) + N$$

where each of the four curators contributes one of the private inputs $R_i$ and $n_i$, and $N$ is a new, global noise term. Next, we describe how $N$ is computed.

## 4.5 DCR: Cooperative noising

MPC enables us to safely remove the noise that was added to the individual cardinalities by BN-PSI-CA, but we must add back a sufficient amount of noise $N$ *as part of the MPC*, i.e., before the result is revealed. To prevent information leakage, the new noise $N$ must be such that no individual curator can control it or predict its value.

We follow the algorithm in [10] to generate the noise $N$, with some implementation modifications. Each curator chooses a random bitstring $v_i$ uniformly at random and contributes it as an input to the MPC. The MPC computes $v := v_1 \oplus v_2 \oplus \cdots$. As long as a curator honestly chooses $v_i$ uniformly at random and does not share this with any other party, she can be certain that no other curator can know anything else about the computed noise string $v$, even if every single other curator colludes. Finally, the MPC uses the fundamental transformation law of probabilities to change the distribution of $v$ to a Laplace distribution $\text{Lap}(1/\varepsilon)$. This yields the noise term $N$, which is then added to the query result. We call this primitive *denoise-combine-renoise (DCR)*.

```
query     := SELECT output FROM union
             WHERE predicate
output    := NOISY COUNT(field)
union     := rows | union UNION ALL rows
rows      := join | subquery
join      := db{,db}*
subquery  := SELECT fields FROM join
             WHERE predicate
predicate := term | predicate OR term |
             predicate AND term
term      := val = val | val != val |
             val < val
val       := number | string | db.field
```

Figure 2: DJoin's query language.

# 5 Distributed query processing

So far, we have described BN-PSI-CA, which can compute differentially private set intersection cardinalities, and DCR, which can privately combine multiple cardinalities. Next, we describe how DJoin integrates these two primitives into larger query plans that can answer SQL-style queries.

## 5.1 Query language: SPJU

For ease of presentation, we describe our approach using the simple query language in Figure 2, which consists of SQL-style operators for selection, projection, a cross join, and union (SPJU). This query language is obviously much simpler than SQL itself, but it is rich enough to capture many interesting distributed operations. We note that many of the missing features of SQL can easily be added back, as long as queries do not use them to access more than one database at a time.

Each query in our language can be translated into relational algebra, specifically, in a combination of selections ($\sigma$), projections ($\pi$), joins ($\bowtie$), unions ($\bigcup$), and counts ($|\cdot|$). For instance, the query

```
SELECT COUNT(A.id) FROM A,B
WHERE (A.ssn=B.ssn OR A.id=B.id)
  AND A.diagnosis='malaria'
```

could be written (with abbreviations) as:

$$|\,\sigma_{(\text{A.ssn=B.ssn} \vee \text{A.id=B.id}) \wedge \text{A.diag="malaria"}}(A \bowtie B)\,|$$

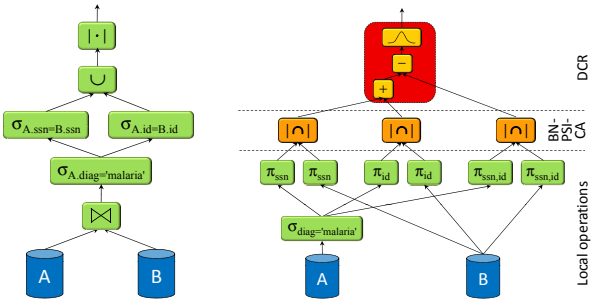
Figure 3(a) shows a graphical illustration of this query.

## 5.2 Query rewriting

Most distributed queries cannot be executed natively by DJoin because they contain operators (such as $\bowtie$ or $<$) that our system cannot support. Therefore, such queries

| | | From | To |
|---|---|---|---|
| R1 | Local sel. | $\sigma_{P(X)\wedge Q}(X \bowtie Y)$ | $\sigma_Q(\sigma_P(X) \bowtie Y)$ |
| R2 | Disjunction | $\sigma_{P\vee Q}(X \bowtie Y)$ | $\sigma_P(X \bowtie Y) \bigcup \sigma_Q(X \bowtie Y)$ |
| R3 | Split | $\lvert\sigma_{X.a=Y.b\wedge(P(X)\vee Q(Y))}(X \bowtie Y)\rvert$ | $\lvert\sigma_{X.a=Y.b}(\sigma_P(X) \bowtie Y)\rvert + \lvert\sigma_{X.a=Y.b}(\sigma_{\neg P}(X) \bowtie \sigma_Q(Y))\rvert$ |
| R4 | Union | $\lvert X \bigcup Y\rvert$ | $\lvert X\rvert + \lvert Y\rvert - \lvert X \bigcap Y\rvert$ |
| R5 | Not equal | $\lvert\sigma_{X.a=Y.b\wedge X.c\neq Y.d}(X \bowtie Y)\rvert$ | $\lvert\sigma_{X.a=Y.b}(X \bowtie Y)\rvert - \lvert\sigma_{X.a=Y.b\wedge X.c=Y.d}(X \bowtie Y)\rvert$ |
| R6 | Comparison | $\lvert\sigma_{X.a=Y.b\wedge X.c>Y.d}(X \bowtie Y)\rvert$ | $\sum_{i=0..k-1} \lvert\pi_{\text{a}\,\lvert\lvert\,\text{pre(c,i)}}(\sigma_{\text{bit(c,i)=1}}(X)) \bigcap \pi_{\text{b}\,\lvert\lvert\,\text{pre(d,i)}}(\sigma_{\text{bit(d,i)=0}}(Y))\rvert$ |
| R7 | Equality | $\lvert\sigma_{X.a=Y.b\wedge X.c=Y.d}(X \bowtie Y)\rvert$ | $\lvert\sigma_{(X.a\lvert\lvert\,\text{pad}\lvert\lvert X.c)=(Y.b\lvert\lvert\,\text{pad}\lvert\lvert Y.d)}(X \bowtie Y)\rvert$ |
| R8 | Join | $\lvert\sigma_{X.a=Y.b}(X \bowtie Y)\rvert$ | $\lvert\pi_a(X) \bigcap \pi_b(Y)\rvert$ |

Table 1: DJoin's rewrite rules. These rules are used to transform a query (written in the language from Figure 2) into the intermediate query language from Figure 4, which can be executed natively.




(a) Original plan (b) Rewritten for DJoin

Figure 3: Query example. The original plan (left) cannot be executed without compromising privacy. The rewritten plan (right) consists of three tiers: a local tier, a BN-PSI-CA tier, and a DCR tier.

must be transformed into other queries that are semantically equivalent but contain only operators that our system *can* support, which are a) any SQL queries on a single database that produce a noisy count or a multiset; b) BN-PSI-CA; and c) DCR. Figure 4 shows the language that can be supported natively. DJoin uses a number of rewrite rules to perform this transformation. The most interesting rules are shown in Table 1; some trivial rules, e.g., for transforming boolean predicates, have been omitted.

**Local selects:** We try to perform as many operations as possible locally at each database, e.g., via rule R1 for selects that involve only columns from one database.

**Disjunctions:** We use basic boolean transformations to move any disjunctions in the join predicates to the outermost level, where they can be replaced by set unions using rule R2, or split off using rule R3.

**Unions:** Rule R4 (which is basically De Morgan's law) replaces all the set unions with additions, subtractions, and set intersections.

**Inequalities:** Rule R5 replaces the $\neq$ operators with an equality test and a subtraction; rule R6 encodes integer comparisons as a sum of equalities. Both rules assume that there is a nearby equality test for matching rows.

```
query    := cardex | cardex + cardex |
            cardex - cardex
cardex   := |setex ∩ setex {∩ setex}*|
setex    := π_fields(sigmaex) | sigmaex
sigmaex  := σ_local_predicate(db) | db
```

Figure 4: DJoin's intermediate language.

**Equalities:** Once all non-local operations in the join predicates are conjunctions of equality tests, we can use rule R7 to reduce these to a single equality test, simply by concatenating the relevant columns in each database (with appropriate padding to separate columns).

**Joins:** Once a join cardinality has only one equality test left, rule R8 replaces it with an intersection cardinality.

## 5.3 Result: Three-tier query plan

If the rewriting process has completed successfully, the rewritten query should now conform to our intermediate language from Figure 4, which implies a three-tier structure: the first tier (`sigmaex` and `setex`) consists of local selections and projections that involve only a single database; the second tier (`cardex`) consists of set intersection cardinalities, and the third tier (`query`) consists of arithmetic operations applied to cardinalities. We refer to the rewritten query as a *query plan*. Figure 3(b) shows a query plan for the query from Figure 3(a) as an illustration.

A query plan with this three-tier structure can be executed in a privacy-preserving way. The first tier can be evaluated using classical database operations on the individual databases; the second tier can be evaluated using BN-PSI-CA (Section 4.2 and 4.3), and the third tier can be evaluated using DCR (Section 4.4 and 4.5).

## 5.4 Limitations

DJoin has only two distributed operators: BN-PSI-CA and DCR. If a query cannot be rewritten into a query

plan that uses only those operators (and some purely local ones), it cannot be supported by DJoin. For instance, DJoin currently cannot process the query

```
SELECT COUNT(A.id) FROM A,B,C
WHERE ((A.x*B.y)<C.z)
```

because we know of no efficient way to rewrite the predicate into set intersections. Rewriting is generally difficult for predicates that involve computations across fields from multiple databases. The predicates DJoin *can* support include 1) predicates that use only fields from a single database, 2) equality tests between fields from different databases, and 3) conjunctions and disjunctions of such predicates. In addition, DJoin supports operators for which it has an explicit rewrite rule, such as inequalities and numeric comparisons (rules R5 and R6). We do not claim that we have found all possible rewrite rules; if rules for additional operators are discovered, DJoin could be extended to support them as well.

DJoin is currently limited to counting queries: it does not support sum queries, or queries with non-numeric results. Differential privacy can in principle support such queries, e.g., via the exponential mechanism [24], but we have not yet found a way to express them in terms of set intersections.

# 6 DJoin design

In this section, we present the design of DJoin, our system for processing distributed differentially-private queries using the mechanisms explained so far.

## 6.1 Assumptions

Our design is based on the following assumptions:

1. All queriers know the schema and a rough upper bound on the total size of each curator's database.
2. The curators are "honest but curious", i.e., they will learn whatever information they can, but they will not deviate from the protocol.
3. Each curator has a "privacy budget" that represents to amount of private information he or she is willing to release through queries.
4. The curators can authenticate each querier.

Assumption 1 is necessary to make BN-PSI-CA and query planning work. Assumption 2 is not inherent (PSI-CA can work in an adversarial model [17]) but helps with efficiency and does not seem unreasonable in practice. Assumption 3 is common for differentially private query processors [16, 23, 32], and assumption 4 can be satisfied, e.g., using cryptographic signatures.

## 6.2 Overview and roadmap

DJoin consists of a number of *servers*, which run on the curators' machines, as well as at least one *client*, which runs the querier's machine and communicates with the servers to execute queries. Each server has a privacy budget (Section 6.3) and a local database with a schema (Section 6.4) that is known to all clients and servers.

Users can interact with DJoin by issuing a query $q$ and a requested accuracy level $\nu$ to their local client. ($\nu$ is the parameter of the Laplace distribution from which DCR will draw the final noise term.) The user's client attempts to rewrite the query according to the rules from Section 5.2. If this succeeds, the result is a different query $q'$ that is equivalent to $q$ but can be executed entirely with local queries, BN-PSI-CA, and DCR. The client then submits the query to the servers, and each server performs an analysis (Section 6.5) to determine the sensitivity $S(q, db_i)$ of the query $q$ in that server's local data $db_i$. In combination with the accuracy level $\nu$, the sensitivity yields the privacy cost $\varepsilon_i$ that this server will incur for answering the query.

Next, the client then uses a distributed commit protocol (Section 6.6) to assign an identifier to the query and to ensure that all the servers agree which query is being executed. Once the query is committed, the servers execute the query in three stages (Section 6.7): first, each server completes any subqueries that involve only its local database; next, the servers jointly complete each of the BN-PSI-CA operations; and finally, the servers execute DCR to combine and re-noise their results. The overall result is then revealed to the client.

## 6.3 Privacy budget

Each server maintains three pieces of local information: A local database, a privacy budget, and a table of pending queries, which is initially empty.

The *privacy budget* is essentially an upper bound on the amount of private information about any individual that the curator owning the server is willing to release through answering queries. It is well known [7] that, if $q_1$ and $q_2$ are two queries that are $\varepsilon_1$- and $\varepsilon_2$-differentially private, respectively, the sequential composition of both is $(\varepsilon_1 + \varepsilon_2)$-differentially private. Because of this, servers can simply deduct each query's "privacy cost" from the budget separately, without having to remember previous queries. A similar construction is used in other differentially private query processors, including PINQ [23], Airavat [32], and Fuzz [16]. In the appendix, we briefly sketch a possible approach to choosing the privacy budget.

Recall from Section 4 that DJoin must charge the privacy budget both for intermediate results from BN-PSI-CA operations and for the final result that is revealed by

DCR. To avoid confusion, we use the symbol $\varepsilon_p$ to denote the cost of a BN-PSI-CA operation and $\varepsilon_r$ to denote the cost of the final result. The total cost of a query with several BN-PSI-CAs is thus $\varepsilon_r + \sum_j \varepsilon_{p,j}$.

## 6.4 Schemata and multiplicities

The local database is a relational database that can be maintained in a classical, non-distributed DBMS, e.g., mySQL. For simplicity, we will assume that the data from each individual user is collected in a single row of the database; if this is not the case already, a normalization step (e.g., a `GROUP BY`) must be performed first. The database schema may assign an arbitrary type $\tau(c)$ to each column $c$; however, to make our sensitivity analysis work, we additionally allow each column to be annotated with a *multiplicity* $m(c)$ that indicates how often any individual value can appear in that column (for instance, $m(c) = 1$ indicates a column of unique keys). If no annotation is present, DJoin assumes $m(c) = \infty$.

Multiplicities are important to determine an upper bound on sensitivity of a query. Recall from Section 3.2 that the sensitivity $S(q, db_i)$ of a counting query $q$ in a database $db_i$ is the largest number of rows that a change to a single row in $D$ can cause to be added or removed from the result of $q$. For instance, consider the query

```
SELECT COUNT(A.x) FROM A,B
WHERE A.x=B.y
```

If the multiplicities are $m(A.x) = 3$ and $m(B.y) = 5$, then a change to a single row in $A$ can add at most five rows to the result – hence, whatever the new value of $A.x$ is, we know that $B$ can contain at most five rows whose $y$-column matches that value. (The argument for disappearing rows is analogous.) Conversely, the query's sensitivity in $B$ is three because at most three rows in $A$ can have the value $B.y$ in column $x$. Note that processing such queries as intersections requires an extra encoding step; see the appendix for details.

Clearly, the use of a column with unbounded multiplicity can cause the sensitivity to become unbounded as well. However, it is safe to use such columns in conjunction with others; for instance, the query

```
SELECT COUNT(A.x) FROM A,B
WHERE A.x=B.y AND A.p=B.q
```

has sensitivity 5 in A even if $m(A.p) = m(B.q) = \infty$.

It may seem tempting to let DJoin choose the multiplicity itself, based on how often elements *actually* occur in the database. However, this would create a side channel: queriers could learn private facts about the database by observing, e.g., how much is deducted from the privacy budget after running certain queries. To avoid this problem, DJoin follows the approach from [16] and determines the multiplicity statically, without looking at the data.

## 6.5 Sensitivity analysis

We now describe how to infer the sensitivity of more complex queries, and specifically on the question how much the number of rows output by a query $\sigma_{\text{pred}}(db_1 \bowtie \cdots \bowtie db_k)$ can change if a single row in one of the $db_i$ is changed.

To explain the intuition behind our analysis, we begin with a few simple examples:

1. $A \bowtie B \bowtie C$
2. $\sigma_{\text{A.x=B.y}}(A \bowtie B \bowtie C)$
3. $\sigma_{\text{A.x=B.y} \wedge \text{ B.y=C.z}}(A \bowtie B \bowtie C)$
4. $\sigma_{\text{A.x=B.y} \wedge \text{ A.p=B.q}}(A \bowtie B \bowtie C)$
5. $\sigma_{\text{A.x=B.y} \wedge \text{ B.y=C.z} \wedge \text{ A.x=C.q}}(A \bowtie B \bowtie C)$

Since query (1) has no predicates, its sensitivity in $A$ is simply $|B| \cdot |C|$. The addition of the constraint $A.x = B.y$ changes the sensitivity to $m(B.y) \cdot |C|$, since each row in $A$ can now join with at most $m(B.y)$ rows in $B$; similarly, adding $B.y = C.z$ in query (3) reduces the sensitivity to $m(B.y) \cdot m(C.z)$. When there is a conjunction of multiple constraints between the same databases, the most selective one 'wins'; hence, the sensitivity of query (4) is $\min(m(B.y), m(B.q)) \cdot |C|$. When there are multiple 'join paths', the most restrictive one wins. For instance, in query (5), the third constraint reduces the sensitivity in $A$ only if $m(C.q) < m(B.y) \cdot m(C.z)$; otherwise, the sensitivity is the same as for query (3).

To solve this problem in the general case, we adapt a classical algorithm from the database literature [18] that was originally intended for query optimization in the presence of joins. This algorithm builds a *join graph* $G$ that contains a vertex for each database that participates in the join, and a directed edge between each pair $(db_1, db_2)$ of vertices that is initially annotated with $|db_2|$, the size of the database $db_2$. We then consider each of the predicates in turn and update the edges. Specifically, for each predicate $db_i.f_1 = db_j.f_2$ with $db_i \neq db_j$, we change the annotation $w_{i,j}$ on the edge $(db_i, db_j)$ to $\min(w_{i,j}, m(db_j.f_2))$ and, correspondingly, the annotation $w_{j,i}$ on $(db_j, db_i)$ to $\min(w_{j,i}, m(db_i.f_1))$. Then we can obtain an upper bound on the sensitivity $S(q, db_i)$ of $q$ in some database $db_i$ by finding the min-cost spanning tree that is rooted at $db_i$, using the product of the edge annotations as the cost function.

If the predicate contains disjunctions, we can rewrite it into DNF and then add up the sensitivity bounds. This is sound because $\sigma_{p \vee q}(X) = \sigma_p(X) \bigcup \sigma_q(X)$. If a row is removed from $X$ and the sensitivities of $p$ and $q$ are $s_p$ and $s_q$, this can change the cardinalities of the two sets by at most $s_p$ and $s_q$, and thus the cardinality of the union by at most $s_p + s_q$. The same approach also works for unions of subqueries.

## 6.6 Distributed commit

Next, we describe how the client submits the query to the servers. It is important to ensure that the servers agree on which query they are executing; without this, a malicious client could trick a server into believing that it is executing a low-sensitivity query, and thus cause an insufficient amount of noise to be added to the result. Note that there is no need to agree on an ordering because all queries are read-only.

When the client accepts a query $q$ with requested noise level $\nu$ from the user, it first calculates the sensitivity of $q$ and the corresponding $\varepsilon$; then it tries to rewrite $q$ into an equivalent query $q'$ that uses only the language from Figure 4. If this succeeds, the client chooses a random identifier $I$ and sends a signed PREPARE$(I, q, q', \nu)$ message to each server. What follows is essentially a variant of the classical two-phase commit protocol.

Upon receiving the PREPARE message, the server at each $C_i$ verifies that $q$ can be rewritten into $q'$, and that it does not already have a pending query with identifier $I$. If either test fails, the server responds with a NAK immediately. Otherwise, $C_i$'s server calculates its privacy cost $\varepsilon_i := \varepsilon_{r,i} + \sum_j \varepsilon_{p,ij}$ that it would incur by executing its part of $q'$. This cost consists of the base cost $\varepsilon_{r,i} := S(q, db_i)/\nu$, which depends on the query's sensitivity in $C_i$'s local data, and an additional charge $\varepsilon_{p,ij}$ for each PSI-CA operation that $C_i$ must participate in to execute $q'$. If $C_i$'s privacy budget can cover $\varepsilon_i$, its server deducts $\varepsilon_i$ from the budget, adds $(I, q', \nu, \varepsilon_i)$ to its pending table, and sends a signed response ACK$(I, q, q', \nu)$ back to the client. Otherwise, the server responds with a NAK. This might occur, for instance, if the sensitivity of $q$ is too high or the requested noise level $\nu$ is too low.

If the client receives at least one NAK, it sends a signed ABORT$(I)$ message to each server that has responded with an ACK, which causes the reserved parts of the privacy budget to be released. Otherwise the client combines the received ACK messages to form a certificate $\Gamma$, and it sends COMMIT$(I, \Gamma)$ to the servers. The servers verify that all required ACKs are present; if so, they begin executing the query.

## 6.7 Query execution

Each query is executed in three stages. First, upon receiving the COMMIT message, the server at each $C_i$ computes the parts of the query that require only data from its local database $db_i$. For some queries, this will yield part of the result directly (e.g., in $|\sigma_{x=0}(A) \bigcup \sigma_{x=1}(B)|$), but more typically the first stage will produce a number of sets on each server that will be used as inputs in the second stage.

The second stage consists of a number of BN-PSI-CA instances. Since all servers agree on the query $q'$, each server can independently determine which BN-PSI-CA instances it should be involved in, and what role in the protocol it should play in each instance. Ties are broken deterministically, and the instances are numbered in order to distinguish different instances that involve the same set of servers. At the end of the second stage, each server has learned a number of noised results and/or noise terms, which are used as inputs to the third stage.

The third stage consists of an invocation of DCR, which de-noises the results from the second stage, combines them as required by $q'$, and then re-noises the combined result using the protocol from Section 4.4. Recall that the re-noising requires an additional input from each server that must be chosen uniformly at random. At the end of the third stage, each server learns the result of the multi-party computation and forwards it back to the client, which displays it to the user.

# 7 Evaluation

In this section, we report results from an experimental evaluation of DJoin. Our goal is to show that 1) DJoin is powerful enough to support useful queries; and that 2) DJoin's communication and computation overheads are low enough to be practical.

## 7.1 Prototype implementation

We have built a prototype implementation of DJoin for our experiments. Our prototype uses mySQL to store each curator's data and to execute the purely local parts of each query, and it relies on FairplayMP [4] to execute the secure multi-party computation. We implemented the two-party BN-PSI-CA primitive from Section 4.2, based on the `thep` library [35] for the Paillier cryptosystem. Our implementation includes the optimizations from [14] that were already briefly described in Section 4.1, including the use of bucket hashing to replace the single high-degree polynomial $P$ with a number of lower-degree polynomials. This reduces BN-PSI-CA's $O(|S_1| \cdot |S_2|)$ time complexity to $O(|S_1| + |S_2| \ln\ln |S_1|)$ and makes it highly parallelizable, with synchronization required only for the few elements that hash to the same bucket. Our prototype also supports multi-party BN-PSI-CA based on the protocol from Kissner and Song [17] and the UTD Paillier Threshold Encryption Toolbox [36], but we do not include multi-party results here due to lack of space.

We also built a query planner that implements the rewrite rules from Section 5.2, as well as a backend for FairplayMP that outputs code for DCR (Section 4.5). To our knowledge, DCR is the first implementation of the shared noise generation algorithm described in [10]. Altogether, our prototype consists of 3,560 lines of Java code for the runtime engine, 249 lines of code in FairplayMP's custom language for the DCR primitive, and 6,776 lines of C++ code for the query planner.

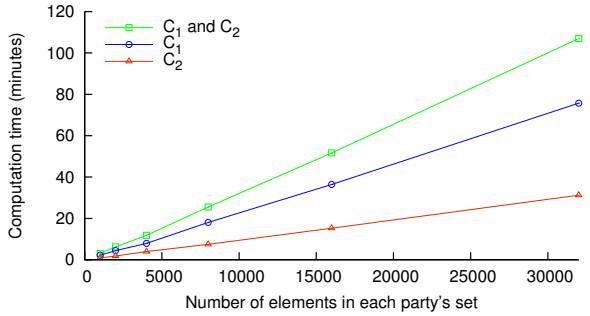



Figure 5: Computation time for PSI-CA. The time is approximately linear in the number of set elements.

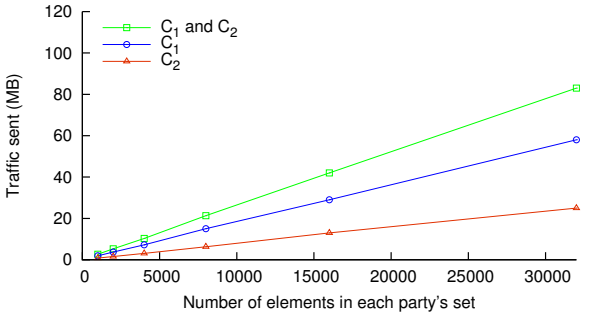



Figure 6: Network traffic sent by the two parties in a BN-PSI-CA run.

## 7.2 Experimental setup

For our experiments, we used five Dell PowerEdge R410 machines, each with a Xeon E5530 2.4 GHz CPU, 12 GB of memory, and four 250 GB SATA disks. The machines were connected by Gbit Ethernet. Following the recommendations in [5], we used 1,024-bit keys for the Paillier cryptosystem. We chose $\varepsilon_r = 0.0212$ to ensure that the noise for a query with sensitivity $s = 1$ is within $\pm 100$ with probability 95%; we set $\varepsilon_p = 1/8 \cdot \varepsilon_r$, and we chose $\delta = 1/N = 6.67 \cdot 10^{-5}$.

Our experiments use synthetic data rather than 'real' confidential data because our cryptographic primitives operate on hashes of the data anyway, so the actual content has no influence on the overall performance. Therefore, we generated synthetic databases. Each database had $N = 15,000$ rows.

## 7.3 Microbenchmarks: BN-PSI-CA

First, we quantified the cost of our two main cryptographic primitives. To measure the cost of BN-PSI-CA, we generated two random sets with $N$ elements each, and we ran two-party BN-PSI-CA on them, varying $N$ between 1,000 and 32,000 elements. We measured the computation time on each party and the amount of traffic that was exchanged between the two parties.

Figure 5 shows the time taken by the servers at $C_1$ and $C_2$, respectively, to execute BN-PSI-CA using a single core. The time increases almost linearly with the size of the sets; recall from Section 7.1 that the optimizations we applied reduce the computational overhead to $O(|S_1| + |S_2| \ln \ln |S_1|)$. Note that the two servers cannot run in parallel; the total runtime is the sum of the two servers' runtimes. Most of the computation is performed by $C_1$: 49% of the total time was spent constructing the polynomials at $C_1$; 29% of the time was spent evaluating the polynomials at $C_2$; and the remaining 21% were spent decrypting the resulting evaluations at $C_1$.

Figure 6 shows the total amount of traffic sent by $C_1$ and $C_2$. The traffic is roughly proportional to the set sizes. For large sets, approximately 70% of the traffic consists of polynomials sent from $C_1$ to $C_2$, and the remaining 30% consists of evaluation results sent back to $C_1$ for decryption.

To quantify BN-PSI-CA's scalability in the number of cores, we performed a 15,000-element intersection with one, two, and four cores. (This was done on a different machine with a 2.67 GHz Intel X3450 CPU, since our E5530s have only two cores.) The additional cores resulted in speedups of 1.99 and 3.98, respectively. This is expected because BN-PSI-CA is trivially scalable: encryptions, polynomial construction, evaluations, and decryptions can all proceed in parallel on multiple cores, or even multiple machines. Thus, DJoin should be able to handle databases much larger than 32,000 elements, as long as the computation can be spread over a sufficient number of machines.

## 7.4 Microbenchmarks: DCR

Next, we quantified the cost of the DCR operator. Recall from Section 4.4 that DCR internally consists of two stages: first, the inputs (cardinalities and inverted noise terms) from the various servers are added together, and then a new noise term is drawn from a Laplace distribution and added to the result. To separate the two stages, we measured the time to execute DCR twice, with and without the second stage, and we varied the number of parties from two to four.

Figure 7 shows our results. The times grow superlinearly with the number of parties ([4] reports a quadratic dependency) but are all below 20 seconds. Although MPC is generally expensive, DJoin performs most of its work using a specialized primitive (BN-PSI-CA), so the functionality that remains for DCR to perform is fairly simple. Note that neither the size nor the number of sets affect DCR's runtime because each server inputs just a single number: the sum of all the cardinalities and noise terms it has computed.

| | Query | #PSI-CA |
|---|---|---|
| Q1 | `SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.y`<br>$\lvert(\pi_x(A) \bigcap \pi_y(B))\rvert$ | 1 |
| Q2 | `SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.x AND (A.y!=B.y)`<br>$\lvert(\pi_x(A) \bigcap \pi_x(B))\rvert - \lvert(\pi_{x,y}(A) \bigcap \pi_{x,y}(B))\rvert$ | 2 |
| Q3 | `SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.y AND (A.z="x" OR B.p="y")`<br>$\lvert(\pi_x(A) \bigcap \pi_y(\sigma_{p=\text{"}y\text{"}}(B)))\rvert + \lvert(\pi_x(\sigma_{z=\text{"}x\text{"}}(A)) \bigcap \pi_y(\sigma_{p\neq\text{"}y\text{"}}(B)))\rvert$ | 2 |
| Q4 | `SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.x OR A.y=B.y`<br>$\lvert(\pi_x(A) \bigcap \pi_x(B))\rvert + \lvert(\pi_y(A) \bigcap \pi_y(B))\rvert - \lvert(\pi_{x,y}(A) \bigcap \pi_{x,y}(B))\rvert$ | 3 |
| Q5 | `SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x LIKE "%xyz%" AND A.w=B.w AND (B.y+B.z>10) AND (A.y>B.y)`<br>$\sum_{i=0..7} \lvert(\pi_{w,(y>>i+1)}(\sigma_{(x \text{ like '\%xyz\%'}) \wedge (y\&2^i=1)}(A)) \bigcap \pi_{w,(y>>i+1)}(\sigma_{((y+z)>10) \wedge (y\&2^i=0)}(B)))\rvert$ | 8 |

Table 2: Example queries and the corresponding query plans. The number of BN-PSI-CA operations, which is a rough measure for the complexity of the query, is shown on the right.

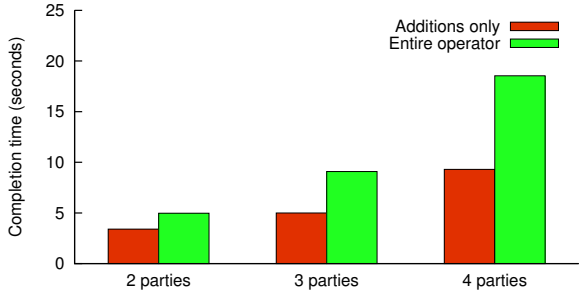



Figure 7: Computation time for DCR with and without the renoising step.

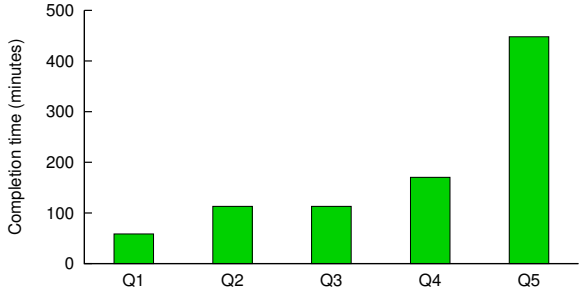



Figure 8: Total query execution time for each of the example queries from Table 2.

## 7.5 Example queries

To demonstrate that DJoin can execute nontrivial and potentially useful queries, we chose five example queries, which are shown in Table 2 along with the query plan they are rewritten into. Each query illustrates a different aspect of DJoin's capabilities:

- **Q1** is an example of a basic join between two databases, which is transformed into a PSI-CA using rule R8.
- **Q2** adds an inequality, which is rewritten as a difference between two intersections via rule R5.
- **Q3** contains a disjunction with two local predicates, which can be split using rule R3.
- **Q4** contains another disjunction, but with remote predicates; this is rewritten via rule R2.
- **Q5** contains an equality and a numeric comparison between columns in different databases, which can be split via rule R6, as well as several other predicates that can be evaluated locally.

For Q5, the `y` column in both databases contained numbers between 0 and 255. The table also shows the number of BN-PSI-CA operations in each query plan, which (in conjunction with the set sizes) is a rough measure of the effort it takes to evaluate it. The more complex a query is, the more BN-PSI-CAs it requires. Q1 is the least complex query because it translates straight into a BN-PSI-CA; Q5 is the most complex one because the inequality requires one intersection per bit.

## 7.6 Query execution cost

To quantify the end-to-end cost of DJoin, we ran each of our five example queries over a synthetic dataset of 15,000 rows per database, and we measured the completion time and the overall amount of network traffic that was sent.

Figures 8 and 9 show our results. The simplest query (Q1) took 58 minutes, and the most complex query (Q5) took 448 minutes, or slightly less than seven and a half hours; the traffic was between 42.7 MB and 340 MB. Both metrics should scale roughly linearly with the size of the sets and the number of set intersections in the

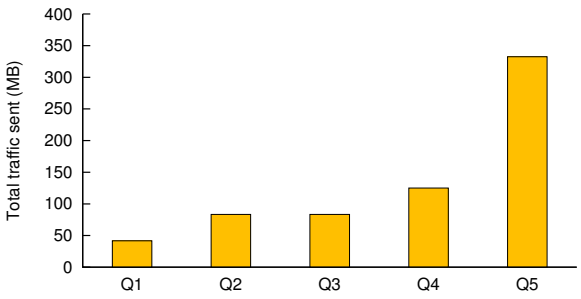



Figure 9: Total network traffic for each of the example queries from Table 2.

query, and a comparison with our microbenchmarks from Section 7.3 confirms this.

The completion times are much higher than the completion times one would expect from a traditional DBMS, but recall that DJoin is not meant for interactive use, but rather for occasional analysis tasks or research studies. For those purposes, an hour or two should be acceptable. Also, recall that the best previously known method for executing such queries is general MPC, which is impractical at this scale.

To illustrate how much DJoin improves performance over straightforward MPC, we implemented our simplest query (Q1) directly in FairplayMP. A version for two databases of just eight (!) rows had 9,700 gates and took 40 seconds to run; we were unable to test larger databases because this produced crashes in FairplayMP. The runtimes we observed increased quadratically with the number of rows, which suggests that this approach is not realistic for the database sizes we consider.

## 8 Conclusion

In this paper, we have introduced two new primitives, BN-PSI-CA and DCR, that can be used to answer queries over distributed databases with differential privacy guarantees, and we have presented a system called DJoin that can execute SQL-style queries using these two primitives. Unlike prior solutions, DJoin is not restricted to horizontally partitioned databases; it supports queries that join databases from different curators together. The key insight behind DJoin is that many distributed join queries can be rewritten in terms of operations on multisets. Not all SQL queries can be transformed in this way, but many can, including counting queries with conjunctions and disjunctions of equality tests, as well as certain inequalities.

DJoin is not fast enough for interactive use, but, to the best of our knowledge, the only known alternative for distributed differentially private join queries is secure multi-party computation, which is orders of magnitude slower. Also, most of the computational cost is due to BN-PSI-CA, which is trivially scalable and can thus benefit from additional cores.

## Acknowledgments

We thank our shepherd, Nickolai Zeldovich, and the anonymous reviewers for their comments and suggestions. We also thank Marco Gaboardi, Benjamin Pierce, Aaron Roth, and Andre Scedrov for helpful comments on earlier drafts of this paper. This work was supported by NSF grants CNS-1065060 and CNS-1054229, ONR grants N00014-09-1-0770 and N00014-12-1-0757, and by a gift from Google.

## References

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. SIGMOD*, May 2000.

[2] M. Barbaro and T. Zeller. A face is exposed for AOL searcher No. 4417749. *The New York Times*, Aug. 2006. `http://nytimes.com/2006/08/09/technology/09aol.html`.

[3] R. M. Bell and Y. Koren. Lessons from the Netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2):75–79, Dec. 2007.

[4] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *Proc. CCS*, Oct. 2008.

[5] J. Bethencourt, D. Song, and B. Waters. New constructions and practical applications for private stream searching (extended abstract). In *Proc. IEEE S&P*, May 2006.

[6] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, Apr. 2012.

[7] C. Dwork. Differential privacy. In *Proc. ICALP*, July 2006.

[8] C. Dwork. Differential privacy: A survey of results. In *Proc. TAMC*, Apr. 2008.

[9] C. Dwork. The differential privacy frontier (extended abstract). In *Proc. IACR TCC*, Mar. 2009.

[10] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. EUROCRYPT*, May 2006.

[11] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, Mar. 2006.

[12] A. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke. Privacy preserving mining of association rules. In *Proc. KDD*, July 2002.

[13] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. OSDI*, Oct. 2010.

[14] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Proc. EUROCRYPT*, May 2004.

[15] M. Götz and S. Nath. Privacy-aware personalization for mobile advertising. Technical Report MSR-TR-2011-92, Microsoft Research, Aug. 2011.

[16] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, Aug. 2011.

[17] L. Kissner and D. Song. Privacy-preserving set operations. In *Proc. CRYPTO*, Aug. 2005.

[18] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. VLDB*, Aug. 1986.

[19] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.

[20] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Proc. ICDE*, Apr. 2007.

[21] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: privacy beyond k-anonymity. In *Proc. ICDE*, Apr. 2006.

[22] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.

[23] F. McSherry. Privacy integrated queries. In *Proc. SIGMOD*, June 2009.

[24] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proc. FOCS*, Oct. 2007.

[25] I. Mironov, O. Pandey, O. Reingold, and S. P. Vadhan. Computational differential privacy. In *Proc. CRYPTO*, Aug. 2009.

[26] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE S&P*, May 2008.

[27] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *Proc. CCS*, Nov. 2001.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT*, May 1999.

[29] B. Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explor. Newsl.*, 4(2):12–19, Dec. 2002.

[30] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. SOSP*, Oct. 2011.

[31] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proc. SIGMOD*, June 2010.

[32] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, Apr. 2010.

[33] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, Feb. 2011.

[34] L. Sweeney. k-anonymity: A model for protecting privacy. *Int. J. Uncert. Fuzzin. Knowl.-Based Syst.*, 10(5):557–570, Oct. 2002.

[35] The Homomorphic Encryption Project. `http://code.google.com/p/thep/`.

[36] UTD Paillier threshold encryption toolbox. Available from `http://utdallas.edu/~mxk093120/paillier/`.

[37] J. Vaidya and C. Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4):593–622, Nov. 2005.

[38] A. Yao. Protocols for secure computations (extended abstract). In *Proc. FOCS*, Nov. 1982.

[39] N. Zhang, M. Li, and W. Lou. Distributed data mining with differential privacy. In *Proc. ICC*, June 2011.

## Appendix

**Choosing $\varepsilon$:** The choice of $\varepsilon$ is essentially a social question and beyond the scope of this paper; however, we briefly sketch one possible approach. Suppose Alice is considering whether or not to allow her data to be included in a database that can later be queried via DJoin, and suppose she is concerned that an adversary might then be able to learn a certain fact about her – for instance, that she has cancer. From Alice's perspective, the worst-case scenario is that the adversary 1) already knows *all* the data in the database (!), except Alice's, that he 2) manages to get access to DJoin, and that he 3) burns the entire privacy budget on a single query $q$ – say, "how many people in the database have cancer?".

Consider the situation from the adversary's perspective. Since we have (very conservatively) assumed that the adversary already knows *all* the data except Alice's, he can construct two "possible worlds": one database $b_1$ where Alice has cancer, and another database $b_2$ where she does not. He does not know whether the real database is $b_1$ or $b_2$, but he can compute the conditional probability $P_i := P(q(db) = r \,|\, db = b_i)$ that $q$ will return $r$ if the real database is $b_i$. Thus, once he observes the actual result, he can use Bayes' formula to update his belief that Alice has cancer.

Now recall that, according to the definition of differential privacy from Section 3.2, $P_1/P_2$ is bounded by $e^{\varepsilon}$. Thus, $\varepsilon$ controls how much more confident the adversary can become about Alice's cancer status. If Alice is comfortable with $P_1/P_2 \leq 2$, she can accept values of $\varepsilon$ up to $\ln 2 \approx 0.69$. If a benign querier wants to ask queries with sensitivity $s = 1$ and $\sum_j \varepsilon_{p,j} = \varepsilon_r$ on a database with $100,000$ entries and have $c = 95\%$ confidence that the error due to noise is less than $E = 1,000$ (1%), we have

$$N_{max} = \frac{\varepsilon_{max} \cdot \lambda_{max}}{2 \cdot s} = \frac{\varepsilon_{max} \cdot E}{-2 \cdot s \cdot \ln(1-c)} \approx 115$$

In other words, a privacy budget of $\varepsilon_{max} = 0.69$ would be enough to answer up to 115 queries of this type.

**Multiset encodings:** In some instances, it is necessary to encode the input sets before they can be processed as intersections. For instance, if the underlying PSI-CA primitive supports sets but not multisets, we can encode an element $e$ that appears $n$ times as $\{e||1,\ldots,e||n\}$, with each element included only once [17]. Another example are joins with multiplicities greater than one. Suppose two curators want to evaluate $|\sigma_x(A) \bowtie \sigma_y(B)|$, and $A.x$ and $B.y$ contain $n_A$ and $n_B$ copies of some element $e$, respectively. Then $A$'s curator can add, $\forall 1 \leq k \leq m(B.y)$, $k \cdot n_A$ encoded elements $e||n_A||k$, and $B$'s curator can add, $\forall 1 \leq k \leq m(A.x)$, $k \cdot n_B$ elements $e||k||n_B$. The intersection then consists of $n_A \cdot n_B$ encoded elements $e||n_A||n_B$.

# Improving Integer Security for Systems with KINT

Xi Wang Haogang Chen Zhihao Jia† Nickolai Zeldovich M. Frans Kaashoek
*MIT CSAIL* †*Tsinghua IIIS*

## Abstract

Integer errors have emerged as an important threat to systems security, because they allow exploits such as buffer overflow and privilege escalation. This paper presents KINT, a tool that uses scalable static analysis to detect integer errors in C programs. KINT generates constraints from source code and user annotations, and feeds them into a constraint solver for deciding whether an integer error can occur. KINT introduces a number of techniques to reduce the number of false error reports. KINT identified more than 100 integer errors in the Linux kernel, the lighttpd web server, and OpenSSH, which were confirmed and fixed by the developers. Based on the experience with KINT, the paper further proposes a new integer family with NaN semantics to help developers avoid integer errors in C programs.

## 1 Introduction

It is well known that integer errors, including arithmetic overflow, division-by-zero, oversized shift, lossy truncation, and sign misinterpretation, can be exploited by adversaries. Recently integer errors have emerged as one of the main threats to systems security. One reason is that it is difficult for programmers to reason about integer semantics [15]. A 2007 study of the Common Vulnerabilities and Exposures (CVE) [1] suggests that they are already "number 2 for OS vendor advisories" [12], second only to buffer overflows. A recent survey [9] reviews the Linux kernel vulnerabilities in CVE from 2010 to early 2011, and confirms the finding that integer errors account for more than one third of the vulnerabilities that can be misused to corrupt the kernel and gain root privilege.

Although integer errors are a known source of problems, there are no detailed studies of integer errors in large systems. This paper's first contribution is a detailed study of integer errors in the Linux kernel, using a new static analysis tool called KINT that we will present in the rest of this paper. We conclude that integer errors are prevalent in all of the subsystems in Linux. We also found 105 new errors for which our patches have been accepted by the Linux kernel community. Finally, we found that two integer errors previously reported in the CVE database were fixed incorrectly, highlighting the difficulty of reasoning about integer semantics.

```
off_t j, pg_start = /* from user space */;
size_t i, page_count = ...;
int num_entries = ...;
if ((pg_start + page_count > num_entries) ||
    (pg_start + page_count < pg_start))
    return -EINVAL;
...
for (i = 0, j = pg_start; i < page_count; i++, j++)
    /* write to some address with offset j */;
```

**Figure 1**: Patched code for the CVE-2011-1745 vulnerability in the Linux AGP driver. The original code did not have the overflow check `pg_start + page_count < pg_start`. In that case, an adversary could provide a large `pg_start` value from user space to bypass the check `pg_start + page_count > num_entries`, since `pg_start + page_count` wraps around. This leads to out-of-bounds memory writes in later code.

In applying the tool to the Linux kernel, we found that the state-of-the-art in static analysis tools for finding integer errors have trouble achieving high coverage and avoiding false error reports when applied to large software systems. For example, both PREfix+Z3 [27] and SmartFuzz [25] use symbolic execution to explore possible paths, but large systems have an exponentially large number of potential paths to explore, making it infeasible to achieve high coverage. Moreover, previous tools (e.g., PREfix+Z3) generate many error reports that do not correspond to actual integer errors [27].

This paper introduces a scalable static analysis for finding integer errors, along with a number of automated and programmer-driven techniques to reduce the number of generated reports, implemented in a tool called KINT. Similar to previous analysis tools, KINT generates a constraint to represent the condition under which an integer error may occur, and uses an off-the-shelf solver to see if it is possible to satisfy the constraint and thus trigger the integer error. Unlike previous tools based on symbolic execution, KINT statically generates a constraint capturing the path condition leading to an integer error, as in Saturn [37], which allows KINT to scale to large systems while maintaining high coverage.

A problem for symbolic execution tools and KINT is the large number of error reports that can be generated for a complex system. To illustrate why it is necessary to reduce the number of error reports, consider the code snippet shown in Figure 1. This example illustrates a correct and widely used pattern for guarding against addition overflow by performing the addition and checking whether the result overflowed. Such checks are prevalent in systems code, including most parts of the Linux kernel.

However, a tool that signaled an error for every integer operation that goes out of bounds would incorrectly flag the overflow check itself as an error, because the check's addition can overflow. In addition to common overflow check idioms, there are a number of other sources for false error reports, such as complex invariants that hold across the entire program which are difficult for an automated tool to infer, and external invariants that programmers assume, such as a number of CPUs not overflowing $2^{32}$.

This paper provides several contributions to help developers effectively find and deal with integer errors, as follows. First, we provide a pragmatic definition of integer errors that avoids reporting common idioms for overflow checking. Second, we introduce a whole-program analysis for KINT that can capture certain invariants in a way that scales to large programs and reduces the number of false errors. Third, because our automated analysis still produces a large number of error reports for Linux (125,172), we introduce range annotations that allow programmers to inform KINT of more complex invariants that are difficult to infer automatically, and thus help reduce false error reports from KINT. Fourth, we introduce a family of overflow-checked integers for C that help programmers write correct code. Finally, we contribute a less error-prone API for memory allocation in the Linux kernel that avoids a common source of integer errors, inspired by Andrew Morton.

Although we focus on the Linux kernel, we believe KINT's ideas are quite general. We also applied KINT to the lighttpd web server and OpenSSH, and found bugs in those systems too.

The rest of this paper is organized as follows. §2 differentiates KINT from previous work on integer error detection. §3 presents a case study of integer errors in the Linux kernel. §4 outlines several approaches to dealing with integer errors. §5 presents KINT's design for generating constraints, including KINT's integer semantics. §6 evaluates KINT using the Linux kernel and known CVE cases. §7 proposes the NaN integer family. §8 summarizes our conclusions.

## 2 Related work

There are a number of approaches taken by prior work to address integer errors, and the rest of this section outlines the relation between this paper and previous work by considering each of these approaches in turn.

*Static analysis.* Static analysis tools are appealing to find integer errors, because they do not require the availability of test inputs that tickle an integer error, which often involve subtle corner cases. One general problem with static analysis is reports of errors that cannot be triggered in practice, termed *false positives*.

One class of static analysis tools is symbolic model checking, which systematically explores code paths for integer errors by treating input as symbolic values and pruning infeasible paths via constraint solving. Examples include PREfix+Z3 [27], KLEE [7], LLBMC [24], SmartFuzz [25], and IntScope [34]. While these tools are effective for exploring all code paths through a small program, they suffer from path explosion when applied to the entire Linux kernel.

KINT is carefully designed to avoid path explosion on large systems, by performing costly constraint solving at the level of individual functions, and by statically generating a single path constraint for each integer operation. This approach is inspired by Saturn [37].

PREfix+Z3 [27], a tool from Microsoft Research, combines the PREfix symbolic execution engine [6] with the Z3 constraint solver to find integer errors in large systems. PREfix+Z3 proposed checking precise out-of-bounds conditions for integer operations using a solver. PREfix, however, explores a limited number of paths in practice [6], and the authors of PREfix+Z3 confirmed to us that their tool similarly stopped exploring paths after a fixed threshold, possibly missing errors. The authors used some techniques to reduce the number of false positives, such as ignoring reports involving explicit casts and conversions between unsigned and signed. Despite these techniques, when applying their tool to 10 million lines of production code, the authors found that the tool generated a large number of false error reports, such as the overflow check described in the introduction.

Verification tools such as ArC (now eCv) [13] and Frame-C's Jessie plugin [26] can catch integer errors, but they accept only a restrictive subset of C (e.g., no function pointers) and cannot apply to systems like the Linux kernel.

Static analysis tools that do not keep track of sanity checks cannot precisely pinpoint integer errors. For example, a simple taint analysis that warns about untrusted integers used in sensitive sinks (e.g., allocation) [8, 16] would report false errors on correctly fixed code, such as the code shown in Figure 1.

The range checker from Stanford's metacompiler [3] eliminates cases where a user-controlled value is checked against *some* bounds, and reports unchecked integer uses. A similar heuristic is used in a PREfast-based tool from Microsoft [30]. This approach will miss integer errors due to incorrect bounds checking since it does not perform reasoning on the actual values of the bounds. KINT avoids these issues by carefully generating constraints that include path conditions.

*Runtime detection.* An advantage of runtime detection for integer errors is fewer false positives. Runtime integer error detection tools insert checks when generating

code; any violation in these checks will cause a trap at run time. Examples include GCC's `-ftrapv`, RICH [4], Archerr [11], IOC [15], blip [19], IntPatch [38], and PaX's overflow GCC plugin [29]. An alternative approach is to instrument binary executable files, such as IntFinder [10] and UQBTng [36]. Using such tools to find integer errors, however, requires carefully chosen inputs to trigger them. Because integer errors typically involve corner cases, the dynamic approaches tend to have low coverage. Because KINT is a static checker, it does not have this limitation.

*Library and language support.* To avoid integer errors, developers can adopt an integer library with error checks, such as CERT's IntegerLib [32, INT03-C] and SafeInt [22]. For example, developers change their code by calling a library function `addsl`$(x,y)$ to add two signed long integers $x$ and $y$. The library then performs sanity checks at run time, and invokes a preset error handler if an integer error occurs.

These integer libraries are trusted and supposed to be implemented correctly. Unfortunately, integer errors have recently been discovered in both IntegerLib and SafeInt [15].

Ada provides language support to define a ranged subtype (e.g., integers from 0 to 9). The runtime will raise an exception on any attempt to store an out-of-bounds value to variables of that subtype, and developers are responsible for handling the exception. There is a similar proposal that adds ranged integers to the C language [17]. Our NaN integer family is inspired by these designs.

*Case studies.* Integer overflows are a well-known problem, and a number of security vulnerabilities have been discovered due to integer errors. Many of the tools above find or mitigate integer errors, and have noted the complexity involved in reasoning about integer errors [15]. In particular, PREfix+Z3 was applied to over 10 millions lines of production code, and the authors of that tool found 31 errors, but provided few details. We are not aware of any detailed study of integer errors and their consequences for a complete OS kernel, which we provide in the next section.

# 3 Case study

A naïve programmer may expect the result of an $n$-bit arithmetic operation to be equal to that of the corresponding mathematical ($\infty$-bit) operation—in other words, the result should fall within the bounds of the $n$-bit integer. Integer errors, therefore, are bugs that arise when the programmer does not properly handle the cases when $n$-bit arithmetic diverges from the mathematically expected result. However, not every integer overflow is an integer error, as described in §1 and shown in Figure 1.

In this section, we present a case study of integer errors in the Linux kernel, which will help motivate the rest of this paper. Figure 2 summarizes the integer errors we discovered in the Linux kernel as part of this case study. Each line represents a patch that fixes one or more integer errors; the number is shown in the "Error" column if it is more than one. An operation may have a subscript $s$ or $u$ to indicate whether it operates on signed or unsigned integers, respectively. As we will describe in the rest of this section, this case study shows that integer errors are a significant problem, and that finding and fixing integer errors is subtle and difficult.

## 3.1 Methodology

To find the integer errors shown in Figure 2, we applied KINT (which we describe later in this paper) to Linux, analyzed the results, and submitted reports and/or patches to the kernel developers. KINT generated 125,172 error reports for the Linux kernel. To determine whether a report was legitimate required careful analysis of the surrounding code to understand whether it can be exploited or not. We could not perform this detailed analysis for each of the reports, but we tried a number of approaches for finding real errors among these reports, as described in §5.6, including several ad-hoc ranking techniques. Thus, our case study is incomplete: there may be many more integer errors in the Linux kernel. However, we report only integer errors that were acknowledged and fixed by kernel developers.

## 3.2 Distribution

As can be seen in Figure 2, the integer errors found in this case study span a wide range of kernel subsystems, including the core kernel, device drivers, file systems, and network protocols. 78 out of the 114 errors affect both 32-bit and 64-bit architectures; 31 errors are specific to 32-bit architecture, and the other 5 are specific to 64-bit architecture.

## 3.3 Incorrect fixes for integer errors

As part of our case study, we discovered that preventing integer errors is surprisingly tricky. Using the log of changes in the kernel repository, the "# of prev. checks" column in Figure 2 reports the number of previous sanity checks that were incorrect or insufficient. The fact that this column is non-zero for many errors shows that although developers realized the need to validate those values, it was still non-trivial to write correct checks. One of the cases, sctp's autoclose timer, was fixed three times before we submitted a correct patch. We will now describe several interesting such cases.

### 3.3.1 Incorrect bounds

Figure 3 shows an example using a magic number $2^{30}$ as the upper bound for `count`, a value from user space. Unfortunately, $2^{30}$ is insufficient to limit the value of `count` on a 32-bit system: `sizeof(struct rps_dev_flow)` is 8,

| Subsystem | Module | Error | Arch | | Impact | Attack vector | # of prev. checks |
|---|---|---|---|---|---|---|---|
| drivers:drm | crtc | $\times_u$ | 32 | 64 | OOB write | user space | – |
| | nouveau | cmp | 32 | 64 | logic error | – | 1 |
| | vmwgfx | $\times_u$ | 32 | | OOB read | user space | – |
| | | $\times_u$ | 32 | 64 | OOB write | user space | 2 |
| | i915 | $\times_u$ (2) | 32 | | OOB write | user space | 1 (2) |
| | savage | $\times_u$ (2) | 32 | | OOB read | user space | – |
| drivers:input | cma3000_d0x | cmp | 32 | 64 | logic error | – | 1 |
| drivers:media | lgdt330x | cmp | 32 | 64 | logic error | – | 1 |
| | uvc | $\times_u$ | 32 | | OOB read | user space | – |
| | wl128x | cmp (36) | 32 | 64 | logic error | – | 1 (36) |
| | v4l2-ctrls | $\times_u$ (2) | 32 | | OOB write | user space | 1 (2) |
| | zoran | $\times_u$ $+_s$ | 32 | 64 | OOB write | user space | 1 |
| drivers:mtd | pmc551 | cmp | 32 | 64 | logic error | – | 1 |
| drivers:scsi | iscsi_tcp | $\times_u$ | 32 | 64 | OOB write | network | – |
| drivers:usb | usbtest | $\times_u$ | 32 | 64 | OOB write | user space | – |
| | | $\times_u$ | 32 | | logic error | user space | 1 |
| drivers:platform | panasonic-laptop | $\times_u$ | 32 | 64 | OOB write | user space | – |
| drivers:staging | comedi | $\times_u$ | 32 | | OOB write | user space | 1 |
| | olpc_dcon | cmp | 32 | 64 | logic error | – | 1 |
| | vt6655 / vt6656 | $\times_u$ $+_u$ (4) | 32 | | OOB write | user space | – |
| drivers:xen | gntdev † | $\times_u$ (5) | 32 | | OOB write | user space | 1 |
| | xenbus | $+_u$ | | 64 | N/A | not exploitable | – |
| block | rbd | $\times_u$ $+_u$ | 32 | | OOB write | disk | – |
| fs | ext4 † | cmp | 32 | 64 | logic error | – | 1 |
| | | << | 32 | 64 | DoS | disk | 1 (CVE-2009-4307) |
| | | $\times_u$ | 32 | | OOB write | disk | – |
| | nilfs2 | $\times_u$ (2) | 32 | | OOB read | user space | 1 |
| | | $\times_u$ | 32 | 64 | logic error | disk | – |
| | xfs | $\times_u$ | 32 | 64 | OOB write | disk | 1 |
| | ceph | index (2) | 32 | 64 | OOB read | network | 1 (2) |
| | | $\times_u$ | 32 | | OOB write | network | 1 |
| | | $+_u$ | 32 | 64 | DoS | network | 1 |
| | jffs2 | $+_u$ | 32 | 64 | OOB write | disk | – |
| kernel | auditsc | cmp | 32 | 64 | logic error | – | – |
| | relayfs † | $\times_u$ (2) | 32 | 64 | OOB write | user space | – |
| mm | vmscan † | cmp | 32 | 64 | logic error | – | 1 |
| net | ax25 | $\times_u$ (8) | 32 | 64 | timer | user space | – |
| | | $\times_u$ (4) | | 64 | timer | user space | 1 (4) |
| | can | cmp | 32 | 64 | logic error | – | – |
| | ceph | $\times_u$ $+_u$ (2) | 32 | | OOB write | network | 1 |
| | | $+_u$ | 32 | 64 | OOB write | network | – |
| | | $\times_u, +_u$ | 32 | | OOB write | network | – |
| | irda | $\times_s$ | 32 | 64 | timer | user space | – |
| | netfilter | $-_u$ (2) | 32 | 64 | wrong output | – | 1 (2) |
| | netrom | $\times_u$ (4) | 32 | 64 | timer | user space | – |
| | rps | $\times_u$ $+_u$ | 32 | | OOB write | user space | 1 |
| | sctp | $\times_u$ | 32 | | timer | user space | 3 |
| | | $+_u$ | 32 | 64 | N/A | – | 1 (CVE-2008-3526) |
| | unix | $+_s$ | 32 | 64 | logic error | user space | – |
| sound | usb | $\times_u$ | 32 | | OOB write | usb | – |
| | | $\times_u$ | 32 | | OOB read | usb | – |

**Figure 2**: Integer errors discovered by our case study in the Linux kernel. Each line is a patch that tries to fix one or more bugs (the number is in the "Error" column if more than one). For each patch, we list the corresponding subsystem, the error operation with the number of bugs, the affected architectures (32-bit and/or 64-bit), the security impact, a description of the attack vector and affected values, and the number of previous sanity checks from the history of the Linux kernel repository that attempt to address the same problem incorrectly or insufficiently. Numbers in parentheses indicate multiple occurrences represented by a single row in the table. Nine bugs marked with † were concurrently found and patched by others.

```
unsigned long count = /* from user space */;
if (count > 1<<30)
    return -EINVAL;
table = vmalloc(sizeof(struct rps_dev_flow_table) +
                count * sizeof(struct rps_dev_flow));
...
for (i = 0; i < count; i++)
    table->flow[i] = ...;
```

**Figure 3**: Incorrect bounds in the receive flow steering (RPS) implementation. The magic number 1 << 30 (i.e., $2^{30}$) cannot prevent integer overflow in the computation of the argument to `vmalloc`.

```
int opt = /* from user space */;
if (opt < 0 || opt > ULONG_MAX / (60 * HZ))
    return -EINVAL;
... = opt * 60 * HZ;
```

**Figure 4**: A type mismatch between the variable `opt`, of type `int` and the bounds `ULONG / (60 * HZ)`, of type `unsigned long`. This mismatch voids the checks intended to prevent integer overflow in the computation `opt * 60 * HZ`.

```
u32 yes = /* from network */;
if (yes > ULONG_MAX / sizeof(struct crush_rule_step))
    goto bad;
... = kmalloc(sizeof(*r) +
              yes * sizeof(struct crush_rule_step),
              GFP_NOFS);
```

**Figure 5**: A malformed check in the form $x > \text{uintmax}_n / b$ from the Ceph file system.

```
if (num > ULONG_MAX / sizeof(u64) - sizeof(*snapc))
    goto fail;
... = kzalloc(sizeof(*snapc) + num * sizeof(u64),
              GFP_NOFS);
```

**Figure 6**: A malformed check in the form $x > \text{uintmax}_n / b - a$ from the Ceph file system.

and multiplying it with a `count` holding the value $2^{30}$ overflows 32 bits. In that case, the allocation size for `vmalloc` wraps around to a small number, leading to buffer overflows later in the loop.

Using magic numbers for sanity checks is not only error-prone, but also makes code hard to maintain: developers need to check and update all such magic numbers if they want to add a new field to `struct rps_dev_flow`, which increases its size. A better practice is to use explicit arithmetic bounds. In this case, the allocation size is in the form of $a + \texttt{count} \times b$; a correct bounds check is $\texttt{count} > (\texttt{ULONG\_MAX} - a)/b$.

In addition, one needs to ensure that the type of the bounds check matches that of the variable to be checked, otherwise a mismatch may void the check. Figure 4 shows one such example. Since `opt` is read from user space, the code checks if the computation of `opt * 60 * HZ` overflows, but the check is incorrect. On a 64-bit system, `opt` of type `int` is a 32-bit integer, while `ULONG_MAX` of type `unsigned long` is a 64-bit integer, with value $2^{64} - 1$. Therefore, the upper bound `ULONG_MAX / (60 * HZ)` fails to prevent a 32-bit multiplication overflow, voiding the check. A correct fix is to change the type of `opt` to `unsigned long`, to match `ULONG_MAX`'s type.

```
struct dcon_platform_data { ...
    u8 (*read_status)(void);
};
/* ->read_status() implementation */
static u8 dcon_read_status_xo_1_5(void)
{
    if (!dcon_was_irq())
        return -1;
    ...
}
static struct dcon_platform_data *pdata = ...;
irqreturn_t dcon_interrupt(...)
{
    int status = pdata->read_status();
    if (status == -1)
        return IRQ_NONE;
    ...
}
```

**Figure 7**: An integer error in the OLPC secondary display controller driver of the Linux kernel. Since `->read_status()` returns an unsigned 8-bit integer, the value of `status` is in the range of $[0, 255]$, due to zero extension. Comparing `status` with $-1$ will always be false, which breaks the error handling.

### 3.3.2 Malformed checks

As discussed in §3.3.1, the correct bounds check to avoid overflow in the expression $a + x \times b$ is:

$$x >_u (\text{uintmax}_n - a)/_u b.$$

where $a$ and $b$ are constants, $x$ is an $n$-bit unsigned integers, and $\text{uintmax}_n$ denotes the maximum unsigned $n$-bit integer $2^n - 1$.

One common mistake is to check for $x > \text{uintmax}_n / b$, which an adversary can bypass with a large $x$. As shown in Figure 5, `yes` is read from network, and a crafted value can bypass the broken check and overflow the addition in the `kmalloc` allocation size, leading to further buffer overflows. Another common broken form is $x > \text{uintmax}_n / b - a$, as shown in Figure 6.

Both forms also appeared when a developer tried to fix a similar integer error in the Linux `perf` tools; the developer wrote three broken checks before coming up with a correct version [31]. We will use this fix from `perf` as an example to demonstrate how to simplify bounds checking using NaN integers in §7.

### 3.3.3 Sign misinterpretation

C provides both signed and unsigned integer types, which are subject to different type conversion rules. Inconsistent choice of signedness often breaks sanity checks. For example, in Figure 7, the intent of the comparison `status == -1` was to check whether `read_status` returns $-1$ on error. However, since the function returns an unsigned 8-bit integer, which is zero-extended to `int` according to C's conversion rules, `status` is non-negative. Consequently, the comparison always evaluates to false (i.e., a *tautological comparison*), which disables the error handling. Using signed `int` for error handling fixes the bug.

```
u32 len = ...;
if (INT_MAX - len < sizeof(struct sctp_auth_bytes))
    return NULL;
... = kmalloc(sizeof(struct sctp_auth_bytes)
              + len, gfp);
```

**Figure 8**: An incorrect fix for CVE-2008-3526 from the sctp network protocol implementation.

```
sbi->s_log_groups_per_flex = /* from disk */;
groups_per_flex = 1 << sbi->s_log_groups_per_flex;
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

**Figure 9**: An incorrect fix to CVE-2009-4307 in the ext4 file system [2], because oversized shifting is undefined behavior in C.

Tautological comparisons are often indicative of signedness errors. Surprisingly, a simple tautological expression that compares an unsigned integer $x$ with 0 (i.e., $x <_u 0$) affected several subsystems. The wl128x driver alone contained 36 such bugs, effectively disabling most of its error handling paths.

Figure 8 shows another example, the fix for CVE-2008-3526, where a sanity check tries to reject a large `len` and avoid overflowing the allocation size. However, the check does not work. Consider `len` = `0xffffffff`. Since `INT_MAX` is `0x7fffffff`, the result of the left-hand side of the check is then `0x80000000`. Note that `len` is unsigned, the left-hand side result is also treated as unsigned (i.e., $2^{31}$), which bypasses the check. A correct check is `len` > `INT_MAX` − `sizeof(struct sctp_auth_bytes)`.

After discussion with the kernel developers, we came to the conclusion that `len` could not become that large. Therefore, CVE-2008-3526 is not exploitable, and the fix is unnecessary. Our patch was nonetheless applied to clarify the code.

### 3.3.4 Undefined behavior

Figure 9 shows a fix for CVE-2009-4307. A developer discovered a division-by-zero bug, which an adversary could trigger by mounting a corrupted file system with a large `s_log_groups_per_flex`. To reject such illegal input, the developer added a zero check against `groups_per_flex`, the result of a shift operation [2].

However, this check turns out to be incorrect. Shifting an $n$-bit integer by $n$ or more bits is undefined behavior in C, and the actual result varies across architectures and compilers. For example, when `s_log_groups_per_flex` is set to 36, which is an illegal value, 1 << 36 is essentially 1 << 4 = 16 on x86, since x86's shifting instruction truncates the amount to 5 bits. This will bypass the check, leaving the two values `s_log_groups_per_flex` (36) and `groups_per_flex` (16) inconsistent. Some C compilers even optimize away the check because they conclude that left-shifting the value 1 never produces zero [35], which effectively eliminates the fix.

Another kernel developer later revised the zero check to `groups_per_flex < 2`, which still suffers from the same problem. This issue was re-assigned CVE-2012-2100 after we reported it. A correct fix is to check `s_log_groups_per_flex` before the shift, so as to avoid undefined behavior.

In general, it is unsafe to check the result of an integer operation that may involve undefined behavior, such as shifts, divisions, and signed integer operations [35]. One should instead check the operands *before* the operation.

## 3.4 Impact

Integer errors that allow out-of-bounds writes (i.e., buffer overflow) can break the integrity of the kernel and potentially enable privilege escalation attacks. They can be exploited via network, local access, or malformed file systems on disk. Figure 3 shows a typical example of an integer error that allows out-of-bounds writes. We found a large number of such errors in `ioctl`, an infamous error-prone interface. There are also two interesting vulnerabilities in the sound subsystem; an adversary can exploit them by plugging in a malicious USB audio device that responds with bogus sampling rates, leading to a kernel hang, DoS, or buffer overflow.

Integer errors cause timing bugs in several network protocol implementations. For example, when a user-space application provides a large timeout argument, the internal timer can wrap around to a smaller timeout value.

Most logic related integer errors are due to tautological comparisons. These bugs would effectively disable error handling, or make the kernel behave in unanticipated ways. One example from the CAN network protocol implementation is as follows:

```
if (((errc & 0x7f) >> 8) > 127) ...
```

The intent of the code is to test whether the error counter `errc` has reached certain level. However, this comparison will never be true because the left-hand side of the test, which extracts 7 bits from `errc`, is at most $2^7 - 1 = 127$. The fix is to check the right bit according to the specification, using `errc` & `0x80`.

# 4 Problems and approaches

As the previous section illustrated, integer errors are common, can lead to serious problems, and are difficult to fix even for experts. Thus, it is important both to find integer errors and to help developers verify their patches or write correct code in the first place.

One approach to prevent integer errors is to avoid the fixed-width arithmetic that leads to integer operations deviating from the mathematically expected semantics. Many languages, such as Python and Haskell, take this approach. However, this is not always feasible because there is a performance penalty for using infinite-precision

arithmetic. Moreover, the runtime and libraries of these languages are often implemented in C, and can have integer errors as well (e.g., CVE-2011-0188 in Ruby's integer implementation). As a result, this paper focuses on helping developers find or avoid integer errors in the presence of fixed-width arithmetic, such as in C code.

Another approach to dealing with integer errors is to find them using static analysis. The key challenges in making this approach work well lie in scaling the analysis to large systems while achieving good coverage and minimizing the number of false error reports. Minimizing false errors is particularly important for verifying correctness of patches. We describe the design of our scalable static analysis tool for finding integer errors, and techniques for reducing the number of false positives, in §5.

Based on the case study, we find that many integer errors occur when computing the number of bytes to allocate for a variable-sized data structure, such as an array of fixed-sized elements. Better APIs that perform overflow-checked multiplication for the caller, similar to the `calloc` function, can help avoid this class of integer errors. To help developers avoid this common problem, we contributed `kmalloc_array(n, size)` for array allocation to the Linux kernel, which checks overflow for $\texttt{n} \times_u \texttt{size}$, as suggested by Andrew Morton. This function has been incorporated in the Linux kernel since v3.4-rc1.

Finally, as illustrated by the case study, programmers can make mistakes in writing overflow checks for integer operations. One approach taken by prior work is to raise an exception every time the value of an integer expression goes out of bounds, such as in Ada or when using GCC's `-ftrapv` flag. However, this can generate too many false positives for overflows that do not matter. §7 describes our proposal for a C language extension that helps developers deal with integer overflows in complex expressions, without forcing all expressions to avoid integer overflows.

# 5 Design

This section describes the design of KINT, and introduces a number of techniques that help KINT reduce the number of error reports for large systems.

## 5.1 Overview

Figure 10 summarizes the design of KINT. The first step in KINT's analysis is to compile the C source code to the LLVM intermediate representation (IR), using a standard C compiler (e.g., Clang). KINT then performs three different analyses on this IR, as follows.

The first analysis, which we will call *function-level analysis*, instruments the IR with checks that capture the conditions under which an integer error may occur, for each individual function. KINT infers integer errors in two ways: first, KINT looks for certain expressions whose value in C is different from its mathematically expected value, and second, KINT looks for values that can violate certain invariants—for example, array indexes that can be negative, control flow conditions that are tautologically true or false, or programmer-supplied invariants.

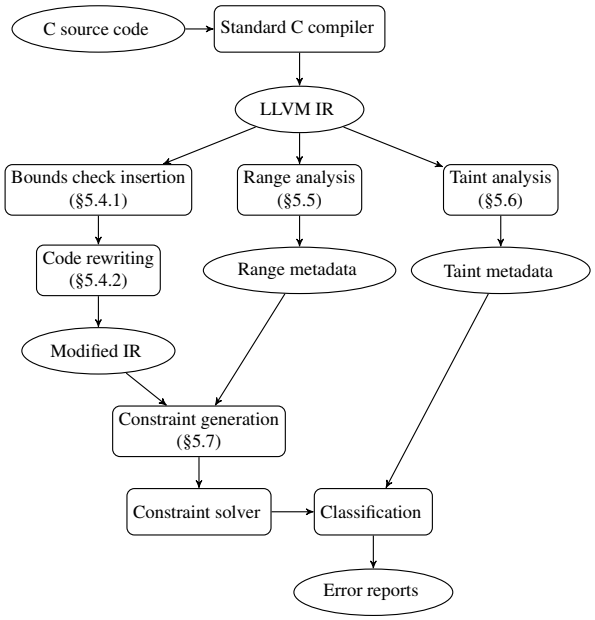



**Figure 10**: KINT's workflow. Ellipses represent data, and rectangles represent phases of KINT's workflow.

The second analysis, called *range analysis*, attempts to infer range constraints on values shared between functions (e.g., arguments, return values, and shared data structures). This analysis helps KINT infer global invariants and thus reduce false error reports.

The third analysis, which we will call *taint analysis*, performs taint tracking to determine which values can be influenced by an untrusted source, and which values may be used in a sensitive context, such as memory allocation; some of these sources and sinks are built in, and others are provided by the programmer. This analysis helps the programmer focus on the errors that are most likely to be exploitable.

Based on the output of function-level and range analyses, KINT generates constraints under which an integer error may occur, and feeds them to a solver to determine whether that integer error can be triggered, and if so, what inputs trigger it. Finally, KINT outputs all cases that trigger integer errors, as reported by the solver, along with annotations from the taint analysis to indicate the potential seriousness of the error.

## 5.2 Applying KINT to Linux

To help KINT detect integer errors, the programmer can define invariants whose violation indicates an integer error. For the Linux kernel, we annotate 23 functions like `memcpy` with the invariant that the size parameter must be non-negative. Annotations are in the form

| Integer operation | In-bounds requirement | Out-of-bounds consequence |
|---|---|---|
| $x +_s y$, $x -_s y$, $x \times_s y$ | $x_\infty$ op $y_\infty \in [-2^{n-1}, 2^{n-1}-1]$ | undefined behavior [21, §6.5/5] |
| $x +_u y$, $x -_u y$, $x \times_u y$ | $x_\infty$ op $y_\infty \in [0, 2^n-1]$ | modulo $2^n$ [21, §6.2.5/9] |
| $x /_s y$ | $y \neq 0 \wedge (x \neq -2^{n-1} \vee y \neq -1)$ | undefined behavior [21, §6.5.5] |
| $x /_u y$ | $y \neq 0$ | undefined behavior [21, §6.5.5] |
| $x << y$, $x >> y$ | $y \in [0, n-1]$ | undefined behavior [21, §6.5.7] |

**Figure 11**: In-bounds requirements of integer operations. Both $x$ and $y$ are $n$-bit integers; $x_\infty, y_\infty$ denote their $\infty$-bit mathematical integers.

(*function-name*,*parameter-index*) in a separate input to KINT. For example, (`memcpy`, 3) means that the third parameter of `memcpy` represents a data size, and KINT will check whether it is always non-negative.

Although KINT's automated analyses reduce the number of error reports significantly, applying KINT to the Linux kernel still produces a large number of false positives. In order to further reduce the number of error reports, the programmer can add range annotations on variables, function arguments, and function return values, which capture invariants that the programmer may know about. Range annotations in the Linux kernel help capture invariants that the programmer knows hold true: for example, that the `tail` pointer in an `sk_buff` is never greater than the `end` pointer. As another example, many `sysctl` parameters in the Linux kernel have lower and upper bounds encoded in the initialization code of their `sysctl` table entries. §6.3 evaluates such annotations, but we did not apply them for the case study in §3.

Finally, to help decide which of the reports are likely to be exploitable, and thus help focus on important errors, the programmer can annotate certain untrusted sources and sensitive sinks. For the Linux kernel, we annotated 20 untrusted sources. For example, (`copy_from_user`, 1) means the first parameter of `copy_from_user`, a pointer, is untrusted; KINT will mark all integers read from the pointer as untrusted. We also annotated 40 sensitive sinks, such as (`kmalloc`, 1); KINT will highlight errors the result of which is used as the first parameter of `kmalloc` (i.e., the allocation size).

## 5.3 Integer semantics

KINT assumes two's complement [20, §4.2.1], a de facto standard integer representation on modern architectures. An $n$-bit signed integer is in the bounds $-2^{n-1}$ to $2^{n-1}-1$, with the most significant bit indicating the sign, while an $n$-bit unsigned integer is in the bounds 0 to $2^n-1$.

KINT assumes that programmers expect the result of an $n$-bit arithmetic operation to be equal to that of the corresponding mathematical ($\infty$-bit) operation. In other words, the result should fall in the $n$-bit integer bounds. Any out-of-bounds operation violates the expectation and suggests an error. Figure 11 lists the requirements of producing an in-bounds result for each integer operation.

*Addition, subtraction, and multiplication.* The mathematical result of an $n$-bit signed additive or multiplicative operation should fall in $[-2^{n-1}, 2^{n-1}-1]$, and that of an unsigned operation should fall in $[0, 2^n-1]$. For example, $2^{31} \times_u 16$ is not in bounds, because the expected mathematical product $2^{35}$ is out of the bounds of 32-bit unsigned integers.

*Division.* The divisor should be non-zero. Particularly, the signed division $-2^{n-1}/_s -1$ is not in bounds, because the expected mathematical quotient $2^{n-1}$ is out of the bounds of $n$-bit signed integers (at most $2^{n-1}-1$).

*Shift.* For $n$-bit integers, the shifting amount should be non-negative and at most $n-1$. Unlike multiplication, KINT assumes that programmers are aware of the fact that a shift operation is lossy since it shifts some bits out. Therefore, KINT considers that $x << 1$ is always in bounds, but $x \times_u 2$ is not.

*Conversion.* KINT does not flag conversions as integer errors (even if a conversion truncates a value into a narrower type), but does precisely model the effect of the conversion, so that an integer error may be flagged if the resulting value violates some invariant (e.g., a negative array index).

## 5.4 Function-level analysis

The focus of function-level analysis is to detect candidate integer errors at the level of individual functions. The analysis applies to each function in isolation in order to scale to large code sizes.

### 5.4.1 Bounds check insertion

KINT treats any integer operation that violates the in-bounds requirements shown in Figure 11 as a potential integer error. To avoid false errors, such as when programmers explicitly check for overflow using an overflowing expression, KINT reports an error only if an out-of-bounds value is *observable* [14] outside of the function. A value is observable if it is passed as an argument to another function, used in a memory load or store (e.g., as an address or the value being stored), returned by the function, or can lead to undefined behavior (e.g., dividing by zero).

At the IR level, KINT flags potential integer errors by inserting a call to a special function called `kint_bug_on` which takes a single boolean argument that can be true

```
#define IFNAMSIZ 16
static int ax25_setsockopt(...,
    char __user *optval, int optlen)
{
    char devname[IFNAMSIZ];
    /* consider optlen = 0xffffffff */
    /* optlen is treated as unsigned: 2^32 - 1 */
    if (optlen < sizeof(int))
        return -EINVAL;
    /* optlen is treated as signed: -1 */
    if (optlen > IFNAMSIZ)
        optlen = IFNAMSIZ;
    copy_from_user(devname, optval, optlen);
    ...
}
```

**Figure 12**: An integer error in the AX.25 network protocol implementation of the Linux kernel (CVE-2009-2909). A negative `optlen` will bypass both sanity checks due to sign misinterpretation and reach the `copy_from_user` call, which interprets `optlen` as a large positive integer. Depending on the architecture-specific implementation, the consequence may be a silent failure, a kernel crash, or a stack overflow.

if an integer error can occur (i.e., the negation of the in-bounds requirements show in Figure 11). KINT will later invoke the solver to determine if this argument can ever be true, in which case an error report will be generated. For example, for division $x/_u y$, the in-bounds requirement of which is $y \neq 0$, KINT inserts `kint_bug_on(y==0)`.

KINT also generates calls to `kint_bug_on` for invariants hard-coded in KINT or specified by the programmer:

- *Array index.* For an array index $x$, KINT generates a call to `kint_bug_on`$(x <_s 0)$.
- *Data size.* A common programmer-supplied invariant is that data size arguments to functions like `memcpy` be non-negative. For calls to such functions with data size argument $x$, KINT generates a call to `kint_bug_on`$(x <_s 0)$. Figure 12 shows an example of such an error.

Tautological control flow conditions, such as in Figure 7, cannot be expressed using calls to the special `kint_bug_on` function. KINT separately generates constraints to check for these kinds of integer errors.

### 5.4.2 Code rewriting

In order to reduce false errors and to improve performance, KINT performs a series of code transformations on the generated LLVM IR.

*Simplifying common idioms.* Explicit overflow checks can lead to complex constraints that are difficult for constraint solvers to reason about. For example, given two $n$-bit unsigned integers $x$ and $y$, a popular overflow checking idiom for $x \times_u y$ is as follows:

$$(x \times_u y)/_u y \neq x.$$

KINT replaces such idioms in the LLVM IR with equivalent expressions, as shown in Figure 13, by using LLVM intrinsic functions that check for overflow. This helps KINT produce simpler constraints to improve solver performance.

| Original expression | Simplified expression |
|---|---|
| $x+y <_u x$ | uadd-overflow$(x,y)$ |
| $x-y <_s 0$ | $x <_u y$ |
| $(x \times y)/_u y \neq x$ | umul-overflow$(x,y)$ |
| $x >_u \text{uintmax}_n - y$ | uadd-overflow$(x,y)$ |
| $x >_u \text{uintmax}_n /_u y$ | umul-overflow$(x,y)$ |
| $x >_u N/_u y$ | $x_{2n} \times_u y_{2n} > N$ |

**Figure 13**: Bounds checking idioms that KINT recognizes and simplifies. Here $x, y$ are $n$-bit unsigned integers, and $x_{2n}, y_{2n}$ denote their $2n$-bit zero-extended values, respectively. Both uadd-overflow and umul-overflow are LLVM intrinsic functions for overflow detection.

*Simplifying pointer arithmetic.* KINT represents each pointer or memory address as a symbolic expression [33], and tries to simplify it if possible. KINT considers a pointer expression that it fails to simplify as an unconstrained integer, which can be any value within its range. Consider the following code snippet:

```
struct pid_namespace {
    int kref;
    struct pidmap pidmap[PIDMAP_ENTRIES];
    ...
};
struct pid_namespace *pid_ns = ...;
unsigned int last = ...;
struct pidmap *map =
    &pid_ns->pidmap[(last + 1)/BITS_PER_PAGE];
int off = map - pid_ns->pidmap;
```

Assume that the offset into the structure field `pidmap[]` is 4 bytes, and the size of its element is 8 bytes. The symbolic expression for `map` and `pid_ns->pidmap` would be `pid_ns` $+4+i\times 8$ and `pid_ns` $+4$ respectively, where the array index $i = (\texttt{last}+1)/_u \texttt{BITS\_PER\_PAGE}$.

Thus, the value of `off`, the subtraction of the two pointers, can be reduced to $(\texttt{pid\_ns}+4+i\times 8)-(\texttt{pid\_ns}+4) = i\times 8$, which is independent from the value of pointer `pid_ns`. Without this rewriting, KINT would have considered `off` to be the result of a subtraction between two unconstrained integers, and would have flagged an error.

*Merging memory loads.* KINT employs a simple memory model: a value returned from a load instruction is unconstrained (unless the value has a range annotation). KINT further merges load instructions to reduce false errors. Consider the example below.

```
/* arg is a function parameter */
if (arg->count < 1 || arg->count > 128)
    return -EINVAL;
int *klist = kmalloc(arg->count * sizeof(int), ...);
if (!klist)
    return -ENOMEM;
ret = copy_from_user(klist, user_ptr,
                     arg->count * sizeof(int));
```

The code correctly limits `arg->count` to prevent a multiplication overflow in `arg->count * sizeof(int)`. To avoid reporting false errors, KINT must know that the value loaded from `arg->count` that appears in the `copy_from_user` call is the same as in the earlier `if` check.

For this purpose, KINT aggressively merges these loads of `arg->count`. It adopts an *unsafe* assumption that a pointer passed to a function argument or a global variable points to a memory location that is distinct from any other pointers [23]. By assuming that `kmalloc` cannot hold a pointer to `arg`, KINT concludes that the call to `kmalloc` does not modify `arg->count`, and merges the two loads.

*Eliminating checks using compiler optimizations.* As the last step in code rewriting, KINT invokes LLVM's optimizer. For each call to `kint_bug_on` which KINT inserted for bounds checking, once the optimizer deduces that the argument always evaluates to false, KINT removes the call. Eliminating these calls using LLVM's optimizer helps avoid subsequent invocations to the constraint solver.

## 5.5 Range analysis

One limitation of per-function analysis is that it cannot capture invariants that hold across functions. Generating constraints based on an entire large system such as the Linux kernel could lead to more accurate error reports, but constraint solvers cannot scale to such large constraints. To achieve more accurate error reports while still scaling to large systems such as the Linux kernel, KINT employs a specialized strategy for capturing certain kinds of cross-function invariants. In particular, KINT's range analysis infers the possible ranges of values that span multiple functions (i.e., function parameters, return values, global variables, and structure fields). For example, if the value of a parameter $x$ ranges from 1 to 10, KINT generates the range $x \in [1, 10]$.

KINT keeps a range for each cross-function entity in a global range table. Initially, KINT sets the ranges of untrusted entities (i.e., the programmer-annotated sources described in §5.2) to full sets and the rest to empty. Then it updates ranges iteratively, until the ranges converge, or sets the ranges to full after a limited number of rounds.

The iteration works as follows. KINT scans through every function of the entire code base. When encountering accesses to a cross-function entity, such as loads from a structure field or a global variable, KINT retrieves the entity's value range from the global range table. Within a function, KINT propagates value ranges using range arithmetic [18]. When a value reaches an external sink through argument passing, function returns, or stores to structure fields or global variables, the corresponding range table entry is updated by merging its previous range with the range of the incoming value.

To propagate ranges across functions, KINT requires a system-wide call graph. To do so, KINT builds the call graph iteratively. For each indirect call site (i.e., function pointers), KINT collects possible target functions from initialization code and stores to the function pointer.

KINT's range analysis assumes strict-aliasing rules; that is, one memory location cannot be accessed as two different types (e.g., two different `structs`). Violations of this assumption can cause the range analysis to generate incorrect ranges.

After the range table converges or (more likely) a fixed number of iterations, the range analysis halts and outputs its range table, which will be used by constraint generation to generate more precise constraints for the solver.

## 5.6 Taint analysis

To help programmers focus on the highest-risk reports, KINT's taint analysis classifies error reports by indicating whether each error involves data from an untrusted input (source), or is used in a sensitive context (sink). KINT propagates untrusted inputs across functions using an iterative algorithm similar to the range analysis which we discussed in the previous subsection.

KINT hardcodes one sensitive context: tautological comparisons. Other sensitive sinks are specified by the programmer, as described in §5.2.

## 5.7 Constraint generation

To detect integer errors, KINT generates *error constraints* based on the IR as modified and annotated by the previous three analyses. For integer errors represented by calls to `kint_bug_on`, KINT reports an error if the argument to `kint_bug_on` may be true. To detect integer errors that lead to tautological comparisons, KINT derives an error constraint from each comparison operation used for control flow: if the expression is always true or always false, KINT reports an error.

For every integer error, KINT must also verify that the error can be triggered in the program's execution; otherwise, KINT would produce false error reports. To do this, KINT generates a *path constraint* for each integer operation, which encodes the constraints on the variables that arise from preceding operations in the function's control flow, similar to Saturn [37]. These constraints arise from two sources: assignments to variables by preceding operations, and conditional branches along the execution path. Satisfying the path constraint with a set of variable assignments means that the integer operation is reachable from the beginning of the function with the given variable values. The path constraint filters out integer errors that cannot happen due to previous statements in a function, such as assignments or explicit overflow checks.

Consider loop-free programs first, using the code in Figure 12 as an example. The control flow of the code is shown in Figure 14. There are two sanity checks on `optlen` before it reaches the call to `copy_from_user`. For clarification purposes, `optlen` is renumbered every time it

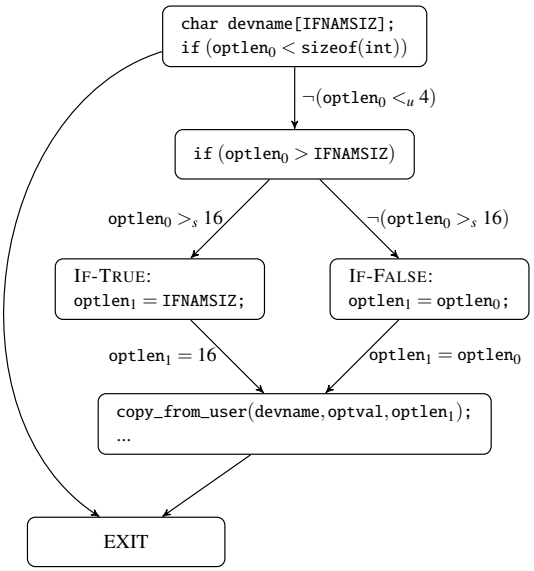



**Figure 14**: The control flow of the code snippet in Figure 12.

is assigned a new value [28, §8.11]. Our goal is to evaluate the path constraint for the call to `copy_from_user`.

The basic algorithm works as follows. Since there is no loop, the path constraint of the call to `copy_from_user` is simply the logical OR of the constraints from each of its predecessors, namely IF-TRUE and IF-FALSE. For each of those two blocks, the constraint is a logical AND of three parts: the branching condition (for the transition from that block to `copy_from_user`), the assignment(s) in that block, and the path constraint of that block. Both IF-TRUE and IF-FALSE unconditionally jump to `copy_from_user`, so their branching conditions are simply true, which can be ignored. Now we have the following path constraint:

$$((\texttt{optlen}_1 = 16) \wedge \text{PathConstraint}(\text{IF-TRUE})) \\ \vee((\texttt{optlen}_1 = \texttt{optlen}_0) \wedge \text{PathConstraint}(\text{IF-FALSE})).$$

By recursively applying the same algorithm to IF-TRUE and IF-FALSE, we obtain the fully expanded result:

$$((\texttt{optlen}_1 = 16) \wedge (\texttt{optlen}_0 >_s 16) \wedge \neg(\texttt{optlen}_0 <_u 4)) \\ \vee((\texttt{optlen}_1 = \texttt{optlen}_0) \wedge \neg(\texttt{optlen}_0 >_s 16) \\ \wedge \neg(\texttt{optlen}_0 <_u 4)).$$

After computing the path constraint, KINT feeds the logical AND of the path constraint and the error constraint (i.e., $\texttt{optlen}_1 <_s 0$) into the solver to determine whether the integer operation can have an error. In this case, the solver will reply with an assignment that triggers the error: for example, $\texttt{optlen}_0 = -1$.

For programs that contain loops, the path constraint generation algorithm unrolls each loop once and ignores branching edges that jump back in the control flow [37].

**function** PATHCONSTRAINT(*blk*)
  **if** *blk* is entry **then**
    **return true**
  $g \leftarrow$ **false**
  **for all** *pred* $\in$ *blk*'s predecessors **do**
    $e \leftarrow (pred, blk)$
    **if** *e* is not a back edge **then**
      $br \leftarrow e$'s branching condition
      $as \leftarrow \bigwedge_i (x_i = y_i)$ for all assignments along $e$
      $g \leftarrow g \vee (\text{PATHCONSTRAINT}(pred) \wedge br \wedge as)$
  **return** *g*

**Figure 15**: Algorithm for path constraint generation.

This approach limits the growth of complexity of the path constraint, and thus sacrifices soundness for performance. The complete algorithm is shown in Figure 15.

To alleviate missing constraints due to loop unrolling, KINT moves constraints inside a loop to the outer scope if possible. Consider the following loop:

```
for (i = 0; i < n; ++i)
    a[i] = ...;
```

KINT generates an error constraint $i <_s 0$ since $i$ is used as an array index. Simply unrolling the loop once (i.e., $i = 0$) may miss a possible integer error (e.g., if the code does not correctly limit $n$). KINT will generate a new constraint $n <_s 0$ outside the loop, by substituting the loop variable $i$ with its exit value $n$ in the constraint $i <_s 0$.

Finally, the Boolector constraint solver provides an API for constructing efficient overflow detection constraints [5, §3.5]. KINT invokes this API to generate constraints for additive and multiplicative operations, which reduces the solver's running time.

### 5.8 Limitations

KINT will miss the following integer errors. KINT only understands code written in C; it cannot detect integer errors written in assembly language. KINT will miss conversion errors that are not caught by existing invariants (see §5.4.1). KINT merges loads in an unsafe way and thus may miss errors due to aliasing. KINT analyzes loops by unrolling them once, so it will miss integer errors caused by looping, for example, an addition overflow in an accumulation. Finally, if the solver times out, KINT may miss errors corresponding to the queried constraints.

## 6 Evaluation of KINT

The evaluation answers the following questions:

- Is KINT effective in discovering new integer errors in systems? (§6.1)
- How complete are KINT's reports? (§6.2)
- What causes KINT to generate false error reports, and what annotations can a programmer provide to avoid these reports? (§6.3)

| | Caught in original? | Cleared in patch? |
|---|---|---|
| CVE-2011-4097 | ✓ | page semantics |
| CVE-2010-3873 | ✓ | CVE-2010-4164 |
| CVE-2010-3865 | accumulation | ✓ |
| CVE-2009-4307 | ✓ | bad fix (§3.3.4) |
| CVE-2008-3526 | ✓ | bad fix (§3.3.3) |
| All 32 others (⋆) | ✓ | ✓ |

(⋆) CVE-2011-4077, CVE-2011-3191, CVE-2011-2497, CVE-2011-2022, CVE-2011-1770, CVE-2011-1759, CVE-2011-1746, CVE-2011-1745, CVE-2011-1593, CVE-2011-1494, CVE-2011-1477, CVE-2011-1013, CVE-2011-0521, CVE-2010-4649, CVE-2010-4529, CVE-2010-4175, CVE-2010-4165, CVE-2010-4164, CVE-2010-4162, CVE-2010-4157, CVE-2010-3442, CVE-2010-3437, CVE-2010-3310, CVE-2010-3067, CVE-2010-2959, CVE-2010-2538, CVE-2010-2478, CVE-2009-3638, CVE-2009-3280, CVE-2009-2909, CVE-2009-1385, CVE-2009-1265.

**Figure 16**: The result of applying KINT to integer errors in Linux kernel from the CVE database. For each case, we show whether KINT catches the expected bugs in the original code, and whether KINT determines that the bug is fixed in the patched code.

- How long does it take KINT to analyze a large system such as the Linux kernel? (§6.4)
- How important are KINT's techniques to reducing the number of error reports? (§6.5)

All the experiments were conducted on a 64-bit Ubuntu Linux machine with an Intel Core i7-980 3.3 GHz CPU and 24 GB of memory. The processor has 6 cores, and each core has 2 hardware threads.

## 6.1 New bugs

We periodically applied KINT to the latest Linux kernel from November 2011 (v3.1) to April 2012 (v3.4-rc4), and submitted patches according to KINT's reports. As discussed in §3, Linux kernel developers confirmed and fixed 105 integer errors. We also applied KINT to two popular user-space applications, lighttpd and OpenSSH; the developers fixed respectively 1 and 5 integer errors reported by KINT. The results show that KINT is effective in finding new integer errors, and the developers are willing to fix them.

## 6.2 Completeness

To evaluate KINT's completeness, we collected 37 known integer errors in the Linux kernel from the CVE database [1] over the last three years (excluding those found by KINT). As shown in Figure 16, KINT is able to catch 36 out of the 37 integer errors.

KINT misses one case, CVE-2010-3865, an addition overflow that happens in an accumulation loop. KINT cannot catch the bug since it unrolls the loop only once.

## 6.3 False errors

To understand what causes KINT to generate false error reports, we performed three experiments, as follows.

*CVE experiment.* We first tested KINT on the patched code of the CVE cases in §6.2, expecting that ideally KINT would not report any error. The results are also shown in Figure 16. KINT reports no bugs in 33 of the 37 cases, and reports errors in 4 cases. One case, the patched code of CVE-2010-3873, contains additional integer errors that are covered by CVE-2010-4164, which KINT correctly identified; two cases contain incorrect fixes as we have shown in §3.3.3 and §3.3.4. One case is a false error in CVE-2011-4097, as detailed below.

```
long points; /* int points; */
points = get_mm_rss(p->mm) + p->mm->nr_ptes;
points += get_mm_counter(p->mm, MM_SWAPENTS);
points *= 1000;
points /= totalpages;
```

The code computes a score proportional to process $p$'s memory consumption. It sums up the numbers of different memory pages that $p$ takes, divides the result by the total number of pages to get a ratio, and scales it by 1000. When the whole system is running out of memory, the kernel kills the process with the highest score.

The patch changes the type of `points` from `int` to `long` because `points` could be large on 64-bit systems; multiplying it by 1000 could overflow and produce an incorrect score, causing an innocent process to be killed.

There is an implicit rule that the sum of these numbers of pages (e.g., from `get_mm_rss`) is at most `totalpages`, so the additions never overflow. KINT's automated analyses are unaware of the rule and reports false errors for these additions, although a programmer can add an explicit annotation to specify this invariant.

*Whole-kernel report analysis.* For the whole Linux kernel, KINT reported 125,172 warnings in total. After filtering for sensitive sinks, 999 are related to memory allocation sizes, 741 of which are derived from untrusted inputs.

We conducted two bug review "marathons" to inspect reports related to allocation sizes in detail. The first inspection was in November 2011: one author applied an early version of KINT to Linux kernel 3.1, spent 12 hours inspecting 97 bug reports and discovered the first batch of 6 exploitable bugs. The 97 reports were selected by manually matching function names that contained "ioctl," since range and taint analyses were not yet implemented.

The second inspection was in April 2012: another author applied KINT to Linux kernel 3.4-rc1, spent 5 hours inspecting 741 bug reports, and found 11 exploitable bugs. All these bugs have been confirmed by Linux kernel developers, and the corresponding patches we submitted have been accepted into the Linux kernel. This shows that KINT's taint-based classification strategy is effective in helping users focus on high-risk warnings.

*Single module analysis.* To understand in detail the sources of false errors that KINT reports, and how many annotations are required to eliminate all false errors (assuming developers were to regularly run KINT against their source code), we examined every error report for a single Linux kernel module, the Unix domain sockets implementation. We chose it because its code is mature and we expected all the reports to be false errors (although we ended up finding one real error).

Initially, KINT generated 43 reports for this module. We found that all but one of the reports were false errors. To eliminate the false reports, we added 23 annotations; about half of them apply to common Linux headers, and thus are reusable by other modules. We describe a few representative annotations next.

The ranges of five variables are determined by a computation. Consider the following example:

```
static u32 ordernum = 1 __range(0, 0xFFFFF);
...
ordernum = (ordernum+1)&0xFFFFF;
```

Since the result is masked with `0xFFFFF`, the value of `ordernum` is up to the mask value. We specified this range using the annotation `__range(`*min*`,`*max*`)` as shown. We used this same annotation to specify the ranges of two structure fields that have ranges defined by existing macros, to specify the lower bound for `struct sock`'s `sk_sndbuf`, and to specify the upper bound of `struct sk_buff`'s `len`. In one case of a reference counter (`struct dentry`'s `d_count`), we are not certain whether it is possible for an adversary to overflow its value. Using `__range` we specified a "workaround" range to suppress related warnings.

For ranges that cannot be represented by constant integers on structure fields or variables, we added assumptions using a special function `kint_assume`, similar to KLEE [7]. An example use is as follows:

```
int skb_tailroom(const struct sk_buff *skb)
{
    kint_assume(skb->end >= skb->tail);
    return skb_is_nonlinear(skb)
            ? 0 : skb->end - skb->tail;
}
```

Some of these annotations could be inferred by a better global analysis, such as an extension of our range analysis. However, many annotations involve complex reasoning about the total number of objects that may exist at once, or about relationships between many objects in the system. These invariants are likely to require programmer annotations even with a better tool.

### 6.4 Performance

To measure the running time of KINT, we ran KINT against the source code of Linux kernel 3.4-rc1, with all modules enabled. We set the timeout for each query to the constraint solver to 1 second. KINT analyzed 8,916 files within roughly 160 minutes: 33 minutes for compilation using Clang, 87 minutes for range and taint analyses, and 37 minutes for generating constraints and solving 420,742 queries, of which 3,944 (0.94%) queries timed out. The running time for other analyses was negligible. The results show that KINT can analyze a large system in a reasonable amount of time.

| Technique | Time (s) | Queries | Reports |
|---|---|---|---|
| Strawman (§6.5) | 834 | 770,445 | 231,003 |
| + Observability (§5.4.1) | 801 | 738,723 | 201,026 |
| + Code rewriting (§5.4.2) | 584 | 408,880 | 168,883 |
| + Range analysis (§5.5) | 1,124 | 420,742 | 125,172 |
| + Taint analysis (§5.6) | 2,238 | 420,742 | 85,017 |

**Figure 17**: Effectiveness of techniques in KINT, enabling each of them one by one in order. The time is for constraint generation and solving only. The compilation time is 33 minutes for all techniques. Range and taint analyses themselves take additional 87 minutes, if enabled.

### 6.5 Technique effectiveness

To evaluate the effectiveness of KINT's techniques, we measured the running time, the total number of queries, and the number of error reports for different configurations of KINT when analyzing the Linux kernel. We start with a strawman design which generates a constraint for each integer expression as shown in Figure 11, feeds this constraint (combined with the path constraint) to the solver, and reports any satisfiable constraints as errors. We then evaluate KINT's techniques by adding them one at a time to this strawman: observability-based bounds checking (§5.4.1), code rewriting (§5.4.2), range analysis (§5.5), and taint analysis (§5.6), using the annotations described in §5.2 and discarding reports with no source or sink classifications.

Figure 17 shows the results, which suggest that all of KINT's techniques are important for analyzing a large system such as the Linux kernel.

## 7 NaN integer semantics

§3 shows that writing correct overflow checks is tricky and error-prone, yet KINT generates many error reports, which makes it difficult for programmers to examine every one of them to ensure that no integer overflows remain. To help programmers nonetheless write correct code, we propose a new integer family with NaN (not-a-number) semantics: once an integer goes out of bounds, its value enters and stays in a special NaN state.

We demonstrate the use of NaN integers using an example from the Linux `perf` tools, which contains the two verbose overflow checks, as shown in Figure 18. Before this correct version, the developers proposed three incorrect checks [31], as we discussed in §3.3.2.

```
size_t symsz     = /* input */;
size_t nr_events = /* input */;
size_t histsz, totalsz;
if (symsz > (SIZE_MAX - sizeof(struct hist))
            / sizeof(u64))
    return -1;
histsz = sizeof(struct hist) + symsz * sizeof(u64);
if (histsz > (SIZE_MAX - sizeof(void *))
             / nr_events)
    return -1;
totalsz = sizeof(void *) + nr_events * histsz;
void *p = malloc(totalsz);
if (p == NULL)
    return -1;
```

**Figure 18**: Preventing integer overflows using manual checks, which are verbose and error-prone [31].

```
nan size_t symsz     = /* input */;
nan size_t nr_events = /* input */;
nan size_t histsz, totalsz;
histsz  = sizeof(struct hist) + symsz * sizeof(u64);
totalsz = sizeof(void *) + nr_events * histsz;
void *p = malloc(totalsz);
if (p == NULL)
    return -1;
```

**Figure 19**: Preventing integer overflows using NaN integers (see Figure 18 for a comparison).

With NaN integers, the developers can simplify this code by declaring the appropriate variables using type `nan size_t` and removing the overflow checks, as in Figure 19. If any computation overflows, `totalsz` will be in the NaN state. To catch allocation sizes that are in the NaN state, we modify `malloc` as follows:

```
void *malloc(nan size_t size)
{
    if (isnan(size))
        return NULL;
    return libc_malloc((size_t) size);
}
```

The modified `malloc` takes a `nan size_t` as an argument and uses the built-in function `isnan`$(x)$ to test if the argument is in the NaN state. If so, `malloc` returns `NULL`.

To help programmers insert checks for the NaN state, the type conversion rules for NaN integers are as follows:

- An integer of type $T$ will be automatically promoted to `nan` $T$ when used with an integer of type `nan` $T$.
- The resulting type of an arithmetic or comparison operation with operands of type `nan` $T$ is also `nan` $T$.
- An integer of type `nan` $T$ can be converted to $T$ only with an explicit cast.

We implemented NaN integers by modifying the Clang compiler. The compiler inserts overflow checks for every arithmetic, conversion, and comparison operation of type `nan` $T$, and sets the result to NaN if any source operand is in the NaN state, or if the result went out of bounds; otherwise the operation follows standard C rules. Currently we support only unsigned NaN integers. We chose the maximum value $2^n - 1$ to represent the NaN state for $n$-bit unsigned integers; `isnan`$(x)$ simply compares $x$ with the maximum value. This choice requires programmers to not store $2^n - 1$ in a NaN integer.

| | w/o `malloc` | w/ `malloc` |
|---|---|---|
| No check | 3.00±0.01 | 79.03±0.01 |
| Manual check | 24.01±0.01 | 104.04±0.03 |
| NaN integer check | 4.05±0.17 | 82.03±0.05 |

**Figure 20**: Performance overhead of checking for overflow in $x \times_u y$ using a manual check (`x != 0 && y > SIZE_MAX / x`) and using NaN integers, with and without a `malloc` call using the result, in cycles per operation over $10^6$ back-to-back operations, averaged over 1,000 runs.

The runtime overhead of NaN integers is low, since the compiler generates efficient overflow detection instructions for these checks. On x86, for example, the compiler inserts one `jno` instruction after the multiplication, which jumps in case of no overflow.

We compare the cost of a single multiplication $x \times_u y$, as well as a multiplication followed by a `malloc` call, in three scenarios: with no overflow check, with a manual overflow check using (`x != 0 && y > SIZE_MAX / x`), and with an overflow check using NaN integers. Figure 20 shows the results. With a single multiplication, overflow checking using NaN integers adds 1–3 cycles on average, and manual overflow checking adds 21–25 cycles. Given the negligible overhead, we believe that it is practical to replace manual overflow checks with NaN integers.

## 8 Conclusion

This paper describes the design and implementation of KINT, a tool that uses scalable static analysis to identify integer errors. It aided in fixing more than 100 integer errors in the Linux kernel, the lighttpd web server, and OpenSSH. KINT introduces several automated and programmer-driven techniques that help reduce the number of false error reports. The error reports highlight that a common integer error is unanticipated integer wraparound caused by values from untrusted inputs, and this paper also proposes NaN integers to mitigate this problem in the future. All KINT source code is publicly available at `http://pdos.csail.mit.edu/kint/`.

## Acknowledgments

We thank Jon Howell, Robert Morris, Yannick Moy, Armando Solar-Lezama, the anonymous reviewers, and our shepherd, Tim Harris, for their feedback. Fan Long helped implement an earlier version of range analysis. The Linux kernel, lighttpd, and OpenSSH developers reviewed our bugs reports and patches. This research was supported by the DARPA CRASH program (#N66001-10-2-4089). Zhihao Jia was supported by the National Basic Research Program of China Grant 2011CBA00300 and NSFC 61033001.

## References


[1] Common vulnerabilities and exposures (CVE). `http://cve.mitre.org/`.

[2] ext4: fixpoint divide exception at `ext4_fill_super`. Bug 14287, Linux kernel, 2009. `https://bugzilla.kernel.org/show_bug.cgi?id=14287`.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, May 2002.

[4] D. Brumley, T. Chiueh, and R. Johnson. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb–Mar 2007.

[5] R. Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University, Linz, Austria, Nov 2009.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Dec 2008.

[8] E. N. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. In *Proceedings of the 3rd GI/IEEE SIG SIDAR Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, pages 1–16, Berlin, Germany, Jul 2006.

[9] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, Jul 2011.

[10] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie. IntFinder: Automatically detecting integer bugs in x86 binary program. In *Proceedings of the 11th International Conference on Information and Communications Security*, pages 336–345, Beijing, China, Dec 2009.

[11] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime environment driven program safety. In *Proceedings of the 9th European Symposium on Research in Computer Security*, pages 385–406, Sophia Antipolis, France, Sep 2004.

[12] S. Christey and R. A. Martin. *Vulnerability Type Distributions in CVE*, May 2007. `http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf`.

[13] D. Crocker. *Verifying absence of integer overflow*, Jun 2010. `http://critical.eschertech.com/2010/06/07/verifying-absence-of-integer-overflow/`.

[14] R. B. Dannenberg, W. Dormann, D. Keaton, T. Plum, R. C. Seacord, D. Svoboda, A. Volkovitsky, and T. Wilson. As-if infinitely ranged integer model. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, pages 91–100, San Jose, CA, Nov 2010.

[15] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*, Zurich, Switzerland, Jun 2012.

[16] D. Evans, J. V. Guttag, J. J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, New Orleans, LA, Dec 1994.

[17] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. C. Seacord. Ranged integers for the C programming language. Technical Note CMU/SEI-2007-TN-027, Carnegie Mellon University, 2007.

[18] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[19] O. Horovitz. Big loop integer protection. *Phrack*, 9(60), 2002.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Intel, 2011.

[21] *ISO/IEC 9899:2011, Programming languages — C*. ISO/IEC, 2011.

[22] D. LeBlanc. Integer handling with the C++ SafeInt class. `http://msdn.microsoft.com/en-us/library/ms972705`, 2004.

[23] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, Helsinki, Finland, Sep 2003.

[24] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, and Experiments*, Philadelphia, PA, Jan 2012.

[25] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium*, pages 67–81, Montreal, Canada, Aug 2009.

[26] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Orsay, France, Jan 2009.

[27] Y. Moy, N. Bjørner, and D. Sielaff. Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis. Technical Report MSR-TR-2009-57, Microsoft Research, 2009.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] E. Revfy. Inside the size overflow plugin. `http://forums.grsecurity.net/viewtopic.php?f=7&t=3043`, Aug 2012.

[30] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. Technical Report MSR-TR-2006-44, Microsoft Research, 2006.

[31] C. Schafer. [PATCH v3] perf: prevent overflow in size calculation. `https://lkml.org/lkml/2012/7/19/507`, Jul 2012.

[32] R. C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 2008.

[33] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 118–132, Genova, Italy, Apr 2001.

[34] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb 2007.

[35] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd ACM SIGOPS Asia-Pacific Workshop on Systems*, Seoul, South Korea, Jul 2012.

[36] R. Wojtczuk. UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries. In *Proceedings of the 22nd Chaos Communication Congress*, Berlin, Germany, Dec 2005.

[37] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, Long Beach, CA, Jan 2005.

[38] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Symposium on Research in Computer Security*, pages 71–86, Athens, Greece, Sep 2010.

# Dissent in Numbers: Making Strong Anonymity Scale

David Isaac Wolinsky, Henry Corrigan-Gibbs, and Bryan Ford
Yale University

Aaron Johnson
U.S. Naval Research Laboratory

## Abstract

Current anonymous communication systems make a trade-off between weak anonymity among many nodes, via onion routing, and strong anonymity among few nodes, via DC-nets. We develop novel techniques in Dissent, a practical group anonymity system, to increase by over two orders of magnitude the scalability of strong, traffic analysis resistant approaches. Dissent derives its scalability from a client/server architecture, in which many unreliable clients depend on a smaller and more robust, but administratively decentralized, set of servers. Clients trust only that *at least one* server in the set is honest, but *need not know or choose which server to trust.* Unlike the quadratic costs of prior peer-to-peer DC-nets schemes, Dissent's client/server design makes communication and processing costs linear in the number of clients, and hence in anonymity set size. Further, Dissent's servers can unilaterally ensure progress, even if clients respond slowly or disconnect at arbitrary times, ensuring robustness against client churn, tail latencies, and DoS attacks. On DeterLab, Dissent scales to 5,000 online participants with latencies as low as 600 milliseconds for 600-client groups. An anonymous Web browsing application also shows that Dissent's performance suffices for interactive communication within smaller local-area groups.

## 1 Introduction

Anonymous communication is a fundamental component of democratic culture and critical to freedom of speech [5, 40,56,57,59], as an AAAS conference in 1997 concluded:

> "Anonymous Communication Should Be Regarded as a Strong Human Right; In the United States It Is Also a Constitutional Right" [56]

The Arab Spring underscored the importance of this right, as organizers used pseudonymous Facebook and Twitter accounts to coordinate protests [46], despite violating those sites' Terms of Service and risking account closure [48]. Authoritarian states routinely monitor and censor Internet communication [22]: though citizens may risk "slap-on-the-wrist" punishments like blocking or throttling if detected to be using anonymity or circumvention tools [24,25,50,62], users discussing the wrong topic *without* protecting their identity risk jail or worse.

Even in countries with strong free speech traditions, anonymity can protect minority groups from discrimination [53]. Increasingly pervasive, profit-motivated tracking practices [51] have made communication linkability a widespread privacy concern [30]. Finally, anonymity plays other well-established roles in modern societies, such as in voting [2, 19, 44] and auctions [52].

Anonymous relay tools such as Tor [26] offer the strongest practical identity protection currently available, but exhibit several classes of weaknesses. First, relay systems are vulnerable to traffic analysis [6, 13, 37, 43, 45]. A state-controlled ISP, for example, who can monitor both a user's "first-hop" link to Tor and the "last-hop" link from Tor to the user's communication partner, can correlate packets to de-anonymize flows [43]. Second, active disruption attacks can not only "deny service" but de-anonymize flows as well [6, 10]. Third, independent of the underlying anonymity protocols in use, widely-deployed tools often fail to isolate anonymous from non-anonymous communication state adequately, causing application-layer identity leaks via third-party browser plug-ins for example [1, 9, 17, 27].

As a step toward stronger anonymity and tracking protection we offer Dissent, a practical anonymous group group communication system resistant to traffic analysis. Dissent builds on and derives its strength from dining cryptographers or DC-nets [14, 36] and verifiable shuffles [11, 32, 44]. Prior systems to adopt these techniques, such as Herbivore [35, 49] and an earlier version of Dissent [20], demonstrated usability for anonymity sets only up to 40–50 participants, due to challenges in scaling and handling network dynamics. This paper improves the scalability of these strong anonymity techniques by at least two orders of magnitude, substantially narrowing the gap compared with relaying approaches [18, 21, 26, 38, 42].

Dissent derives scalability from an *anytrust* architecture [60]. A Dissent group consists of a potentially large set of *client* nodes representing users, and a smaller set of *servers*, facilitators of anonymous communication. Each client trusts that at least *any* one server will behave hon-

estly and not collude with the others against it, but the client *need not know or choose which server to trust*. While anytrust is not a new idea, Dissent rethinks DC-nets communication [14] around this model by sharing secret "coins" only between client/server pairs rather than between all node pairs, yielding a novel, practical and scalable system design. This design reduces clients' computation and communication burdens, and crucially in practical networks, decouples a group's overall communication performance from long "tail latencies" caused by slow, abruptly disconnected, or disruptive clients.

A Dissent prototype demonstrates usability on DeterLab with anonymity sets of over 5,000 members—over two orders of magnitude larger than anonymity sets demonstrated in comparable prior systems [20,35,49]. We expect Dissent to scale further with better optimization.

Although this paper's primary contribution is to show that strong anonymity can scale, Dissent also addresses certain disruption and information leakage vulnerabilities. In Tor and prior DC-nets schemes, an adversary who controls many nodes can anonymously disrupt partially-compromised circuits to increase the chance of *complete* compromise as circuits or groups re-form [10]. Dissent closes this vector with an *accusation* mechanism adapted to its anytrust network model, enabling a partially-compromised group to identify and expel disruptors without re-forming from scratch.

In local-area settings with low delay and ample bandwidth, Dissent can be used for anonymous interactive browsing with performance comparable to Tor. In this context Dissent can offer a strong local-area anonymity set complementing Tor's larger-scale but weaker anonymity. Dissent addresses an important class of anonymous browsing vulnerabilities, due to application-level information leaks [1, 9, 27], by confining the complete browser used for anonymous communication—including plug-ins, cookies, and other state—in a virtual machine (VM) that has no access to non-anonymous user state, and which has network access *only* via Dissent's anonymizing protocols.

Dissent has many limitations and does not yet address other weaknesses, such as long-term intersection attacks [39]. As a step toward stronger practical anonymity, however, this paper makes the following contributions:

1. An existence proof that traffic analysis resistant anonymity is feasible among thousands of participants.
2. A client/server design for DC-nets communication that tolerates slow or abruptly disconnecting clients.
3. A accusation mechanism offering disruption resistance in large-scale, low-latency DC-nets designs.
4. A VM-based browsing architecture enforcing a separation between anonymous and non-anonymous state.
5. Experiments demonstrating Dissent's usability in wide-area messaging applications, local-area interactive anonymity groups, and as a complement to Tor.

Section 2 of this paper describes Dissent's goals and how they relate to previous work. Section 3 presents Dissent's architecture. In Section 4, we overview our prototype, deployment models, and experiences. Section 5 presents the results of our experiments. We conclude with a summary of the paper's accomplishments.

# 2 Background and Related Work

This section outlines the state of the art in both practical anonymity systems and theoretical protocols, with a focus on the key security weaknesses that Dissent addresses.

## 2.1 Practical Anonymity on the Internet

Users can set "Do Not Track" flags [30] asking web sites not to track them. This advisory mechanisms asks the fox to guard the henhouse, however, relying on honest behavior from the web site and all network intermediaries. Even granted the force of law, such requests may be ignored by web sites in "grey markets" or foreign jurisdictions, just as today's anti-spam laws are ignored and circumvented.

For active protection against tracking or identification, centralized relay services such as Anonymizer [4] offer convenience but limited security, since one compromised server—or one subpoena—can break a user's anonymity. Users can create accounts under false names on popular services such as Facebook and Google+, but risk account loss due to Terms of Service violations—often for dubious reasons [48]—and may still be traceable by IP address.

For stronger protection without a single point of failure, decentralized relay networks [18, 21, 26, 38, 42] have proven practical and scalable. Relaying generally trades convenience against security, however, with some caveats [54]. Mixminion [21] forwards E-mail through a series of relays, delaying and batching messages at each hop to offer some traffic analysis protection. Tor [26], in contrast, consciously sacrifices traffic analysis protection to achieve low latencies for interactive Web browsing.

## 2.2 Anonymity Sets: Size versus Strength

The convenience that "weaker" systems such as Tor offer users may paradoxically give them a security *advantage* over "stronger" but less convenient systems such as Mixminion, because convenience attracts more users and thus yields much larger effective *anonymity sets* for their users to hide in. Tor only offers these large anonymity sets, however, *provided* the attacker is not capable of traffic

analysis—likely a reasonable assumption when Tor was designed. In today's more diverse global Internet, however, the adversary from whom users need identity protection may often be a national ISP controlled by an authoritarian state. Such an adversary realistically *can* monitor and "fingerprint" the traffic patterns of users and web sites *en masse*, completely de-anonymizing Tor flows that start and end within the same state. More recent traffic analysis attacks [6, 13, 37, 45] further accentuate this class of vulnerabilities. Thus, Tor may informally be viewed as offering a *potentially large but weak* anonymity set.

Two other approaches to anonymity theoretically offer security even against traffic analysis: verifiable shuffles [11, 32, 44], and "dining cryptographers" or DC-nets [14, 36, 58]. Communication and computation costs have in practice limited these methods to small anonymity sets, however. Herbivore [35, 49] supports mass *participation* by securely dividing large networks into smaller DC-nets groups, but guarantees each node *anonymity* only within its own group, showing scalability only to 40-node groups. The first version of Dissent [20] focused on accountability rather than scalability, combining verifiable shuffles with DC-nets to prevent anonymous disruption, but scaled only to 44-node groups. These techniques thus have so far offered *strong but small* anonymity sets.

Today's anonymity techniques thus present even well-informed users with a security conundrum: to use a tool like Tor that under favorable conditions hides them among tens of thousands of others, but under unfavorable conditions may not hide them at all; or to use a tool that can offer only a small anonymity set but with higher confidence. Dissent's goal is to alleviate this conundrum.

# 3 Dissent Architecture

This section first summarizes DC-nets, then details how Dissent achieves scalability and resilience to slow or unreliable clients. It finally outlines how Dissent traces disruptors and schedules rounds, and current limitations.

## 3.1 DC-nets Overview and Challenges

In classic DC-nets [14], one anonymous sender in a group wishes to share a message with fellow group members. To exchange a 1-bit message, every member shares a secret random coin with each of the other $N-1$ members. Every pair together first flips their shared coin, agreeing on the outcome. Then each member individually XORs together the values of all the coins he shares, while the anonymous sender additionally XORs in his 1-bit message, to produce the member's *ciphertext*. Finally, all members broadcast their ciphertexts to each other. Since each coin is XORed into exactly two members' ciphertexts, all shared coins cancel, revealing the anonymous sender's message without revealing who sent it. For longer messages, the group uses multiple coin flips—in practice, cryptographic pseudo-random number generators (PRNGs) seeded by pairwise shared secrets.

Practical implementations of this conceptually simple design face four key challenges: scheduling, disruption, scalability, and network churn. First, since DC-nets yield an Ethernet-like broadcast channel, in which only one member can transmit anonymously in each bit-time without colliding and yielding garbled output, an *arbitration* or *scheduling* mechanism is needed. Second, any misbehaving member can anonymously disrupt or "jam" the channel simply by transmitting random bits all the time. Dissent builds on and extends several prior approaches to address these challenges [20, 36, 58].

The third challenge directly limits scalability. Every member normally shares coins (or keys) with every other member, so each node must compute and combine $O(N)$ coins for every bit of shared channel bandwidth. Computing ciphertexts via modular arithmetic instead of XORed bits [36] can address this issue asymptotically, but at a large constant-factor cost. Communication cost can also limit scalability if every node broadcasts its ciphertext to every other. In Herbivore [35, 49] a single node collects and combines ciphertexts for efficiency, but this leader-centric design offers no reliable way to identify anonymous disruptors without re-forming the group, leaving groups vulnerable to DoS attacks against anonymity [10].

The fourth challenge, network churn, indirectly limits its scalability in practice. As each member shares a coin with every other, a round's output is indecipherable until *all* members submit their ciphertexts. Thus, one slow member delays the entire group's progress. If any member disconnects during a round, all other members must recompute and rebroadcast their ciphertexts anew. Beyond "normal-case" churn, an adversary who controls $f$ group members can take them offline one one at a time to force a communication round to timeout and restart $f$ times in succession. Threshold cryptography can address this issue in non-interactive scenarios [36], but may be too heavyweight for interactive communication.

## 3.2 Design and Deployment Assumptions

Dissent assumes a cloud-like *multi-provider* deployment model illustrated in Figure 1, similar to the model assumed in COR [38]. A Dissent *group* consists of a possibly large number of *client* nodes representing individual users desiring anonymity within the group, supported by a small number of reliable and well-provisioned cloud of *servers*. We assume each server is run by a respected, technically competent, and administratively independent

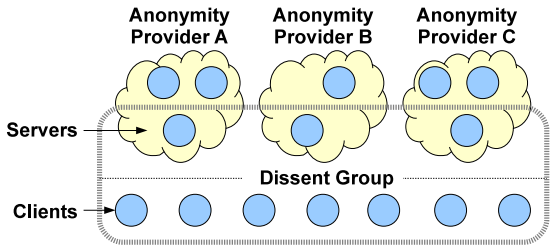



Figure 1: Dissent's multi-provider *anytrust* cloud model

anonymity service provider. We envision several commercial or non-profit organizations each deploying a cluster of Dissent servers to support groups, as either a for-profit or donation-funded community service.

For anonymity and other security properties Dissent relies on an *anytrust* assumption [60]. Clients need not rely on *all* or even *particular* providers or their servers being honest. Instead, each client trusts only that *there exists* at least one provider—*any* provider—who is honest, technically competent, and uncompromised. Clients *need not know or guess which provider's server is the most trustworthy.* Later sections detail how Dissent's design relies on this assumption to achieve scalability.

This paper focuses on the operation of a single group—which forms an anonymity set from a user's perspective—and leaves out of scope most details of how groups are formed or subsequently administered, how providers deploy their services or scale to support many groups, etc. As a simple group formation mechanism we have prototyped, an individual creates a file containing a list of public keys—one for each server (provider) and one for each client (group member)—then distributes this *group definition* file to the clients and servers. A cryptographic hash of this group definition file thereafter serves as a *self-certifying identifier* for the group [31], avoiding membership consensus and PKI issues at the cost of making the group's composition static. The group formation techniques explored in Herbivore [35, 49] could offer complementary methods of forming Dissent groups dynamically.

## 3.3 Dissent Protocol Outline

To initiate communication, a group's servers periodically run a scheduling process described later in Section 3.10. This process yields a list of *pseudonym keypairs*, one for each participating client. All nodes know and agree on the list of public keys and their order, and each client knows the private key for its slot in the DC-net defined by the ordered list of pseudonym keypairs, but neither clients nor servers know which clients hold which *other* slots. This list schedules subsequent DC-nets rounds as shown in Figure 2, and enables the protocol to offer accountability as described later in Section 3.9.

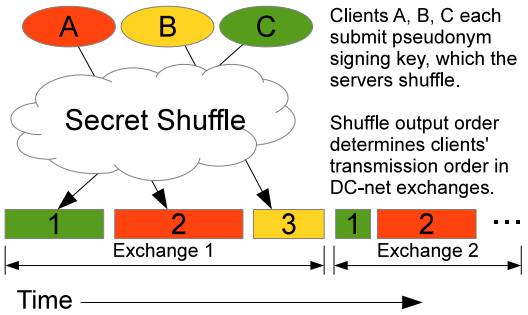



Figure 2: DC-nets scheduling via a verifiable shuffle

After setup, group members commence a continuous series of rounds. Each round allows the owner of each slot to transmit one or more bits anonymously, as defined by the schedule and information from prior rounds. During a round, each client first generates $M$ pseudo-random strings, each based on a secret key he shares with each of the $M$ servers, and XORs these strings together. To send a message, the client additionally XORs his cleartext message into the bit positions corresponding to his anonymous transmission slot. The client then transmits his ciphertexts to one or more servers, then waits.

The servers collect as many client ciphertexts as possible within a time window. At the end of this window, the servers exchange with each other the list of clients whose ciphertexts they have received. Each server then computes one pseudo-random string for each client that submitted a ciphertext, using the secret shared with that client. The server XORs these strings, together with with the client ciphertexts the server received, to form the server's ciphertext (Figure 3). The servers then distribute their ciphertexts among themselves. Upon collecting all server ciphertexts, each server XORs them to reveal the clients' cleartexts, and distributes the cleartexts to the clients connected to them, completing one DC-net round. Successive DC-net rounds ensue.

The rest of this section describes Dissent's client and server protocols, summarized in Algorithms 1 and 2, respectively, and in Figure 4. All network messages are signed to ensure integrity and accountability, but we omit these signatures to simplify presentation.

## 3.4 Secret Sharing in the Anytrust Model

The key to Dissent's scalability and resilience to churn is its client/server secret-sharing graph. Unlike the "all-to-all" secret-sharing graph in most DC-nets designs, Dissent shares secrets *only* between all client/server pairs.

As formalized by Chaum [14], the anonymity set an honest node obtains via DC-nets consists of the node's connected component in the secret-sharing graph, after

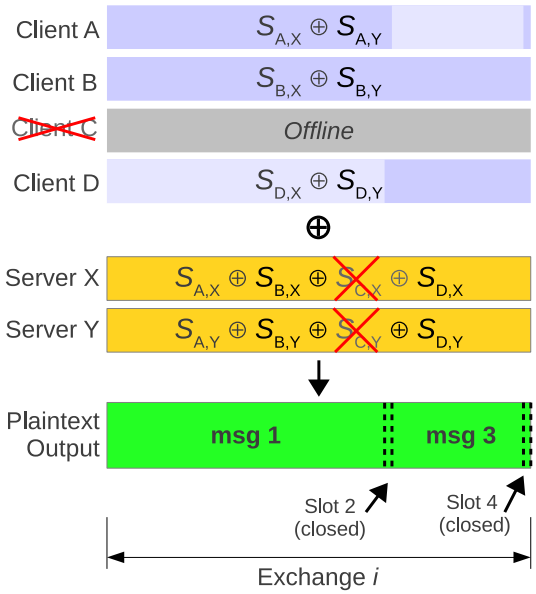



Figure 3: Dissent round structure. Each client-server pair shares a secret pseudo-random string $s_{i,j}$. Client D anonymously transmits a message in slot 1 and client A anonymously transmits a message in slot 3. Servers do not XOR in the strings for offline client C.

removing dishonest nodes and their incident edges from the graph. A sparser secret-sharing graph thus reduces a node's anonymity, compared with the ideal anonymity set consisting of all honest nodes, if and only if the dishonest nodes partition the honest nodes into multiple connected components. Because each Dissent client shares a secret with each server, the honest nodes remain connected—yielding an ideal anonymity set—if and only if there is at least one honest server. This is precisely what Dissent's anytrust model assumes. The downside is that if *all* servers maliciously collude, clients obtain *no* anonymity.

A direct benefit of Dissent's client/server secret-sharing is that clients enjoy a lighter computational load during exchanges. Each client shares secrets with only the $M \ll N$ servers, thus clients need only compute $M$ pseudo-random bits for each effective bit of DC-net channel bandwidth. Each of the servers must compute $N$ pseudo-random bits per cleartext bit, as in traditional DC-nets, but these computations are parallelizable, and Dissent assumes that the servers are provisioned with enough computing capacity to handle this load. Just as important in practice, however, are the model's indirect benefits to network communication and resiliency, detailed next.

## 3.5 Optimizing Network Communication

Dissent leverages its client/server architecture to reduce network communication overhead. In conventional DC-nets, all nodes broadcast messages to all other nodes. Dissent reduces the number of communication channels by a factor of $N$ by organizing clients and servers into a two-level hierarchy. Clients communicate with only a single server, and servers communicate with all other servers.

**Algorithm 1** Dissent Client DC-net Protocol

1. **Scheduling:** Each client $i$ creates a fresh secret *pseudonym key*, $k_{\pi(i)}$, then encrypts and submits it to a key-shuffle protocol, which permutes and decrypts all clients' keys, giving client $i$ a secret *permutation slot* $\pi(i)$ unknown to all other nodes. A well-known scheduling function $S(r, \pi(i), H)$ determines the set of bit-positions client $i$ owns in each subsequent DC-nets round $r$, after a history $H$ of prior round outputs.
2. **Submission:** Each client $i$ forms a cleartext message $m_i$ containing arbitrary data in the bit-positions $i$ owns according to $S(r, \pi(i), H)$, and zero elsewhere. From secrets $K_{ij}$ that client $i$ shares with each server $j$, $i$ computes pseudo-random strings $c_{ij} = \text{PRNG}(K_{ij})$. Client $i$ then XORs these strings with message $m_i$ to produce ciphertext $c_i = m \oplus c_{i1} \oplus \cdots \oplus c_{iM}$, which $i$ signs and transmits to one or more servers.
3. **Output:** Each client $i$ waits for a message from any server containing round $r$'s cleartext output signed by all servers: $(r, \vec{m}, \vec{sig})$. Client $i$ verifies all servers' signatures, extracts the messages in all slots, then proceeds to round $r + 1$ by repeating from step 2.

This optimization does *not* make a client's anonymity dependent on the particular server it chooses to connect to, because anonymity depends on the secret-sharing graph described above and not on physical communication topology. Since each client shares a secret with *all* servers, even if a client's directly upstream server is malicious, that server cannot decode the client's anonymous transmissions except with the cooperation of all the servers, including the honest one we assume exists. A server can DoS-attack an attached client by persistently dropping its submitted ciphertexts, but the client will recognize such an attack from the absence of its cleartexts in the group's output—which *all* servers must sign—signaling the client to switch to a different server.

To reduce communication costs further, servers locally combine their pseudo-random strings with their clients' ciphertexts. Servers thus avoid forwarding individual client ciphertexts to other servers, reducing total communication complexity from $O(N^2)$ to $O(N + M^2)$ when $M \ll N$. Related optimizations reduce the number of signature verifications a client performs from $O(N)$

**Algorithm 2** Dissent Server DC-net Protocol

1. **Submission:** In each round $r$, each server $j$ collects ciphertexts $c_i$ from some clients $i$, until all of $j$'s directly-connected clients have responded or the round closure deadline has passed.
2. **Inventory:** Server $j$ forms a list $\vec{l_j}$ of client identities from whom $j$ has received ciphertexts by the deadline, then broadcasts this list to the other servers.
3. **Commitment:** Given all servers' vectors $l_j$, the servers deterministically trim redundant entries for clients who submitted ciphertexts to multiple servers, yielding new lists $l'_j$, then form a composite client list $l = \bigcup_j l'_j$. If the round $r$ participation count $p_r = |l|$ is below a policy-defined fraction $\alpha$ of the previous round's participation count $p_{r-1}$, the servers return to step 1 and wait for more clients to submit ciphertexts.

   Otherwise each server $j$ computes pseudo-random strings $s_{ij} = \text{PRNG}(K_{ij})$ from the shared secrets $K_{ij}$ of clients $i \in l$, and XORs these strings with the client ciphertexts $j$ received directly, forming server ciphertext $s_j = (\bigoplus_{i \in l} s_{ij}) \oplus (\bigoplus_{i \in l'_j} c_{ij})$. Server $j$ computes a commit $C_j = \text{HASH}(s_j)$ and sends $C_j$ to all servers.
4. **Combining:** Upon receiving all other servers' commit, server $j$ shares $s_j$ with the other servers.
5. **Certification:** The servers verify $C_j = \text{HASH}(s_j)$ for all $j$, and form cleartext output $\vec{m} = \bigoplus_j s_j$. Each server $j$ signs $\vec{m}$, and sends its $sig_j$ to all servers.
6. **Output:** Servers collect all signatures $sig_j$ into $\vec{sig}$, then distribute $(r, \vec{m}, \vec{sig})$ to directly attached clients.

to $O(M)$ and the number of messages clients must parse from $O(N)$ to $O(1)$ (see Algorithm 1, steps 2, and 3).

## 3.6 Tolerating Network Churn

More important than reducing the computational load on clients, Dissent's client/server secret-sharing graph enables the servers to collaborate to reduce the group's vulnerability to client disconnection and churn, as well as deliberate DoS attacks by malicious clients. In conventional online DC-nets, if any member goes offline, all other members must recompute and resend new ciphertexts, omitting the PRNG stream they shared with the failed member. The chance that a given round will have to be "re-run" in this way increases dramatically as group size and client churn increase.

Since Dissent clients share secrets only with the servers and not with other clients, a client's ciphertext is independent of the online status of other clients. The servers can therefore complete a messaging round even if some clients

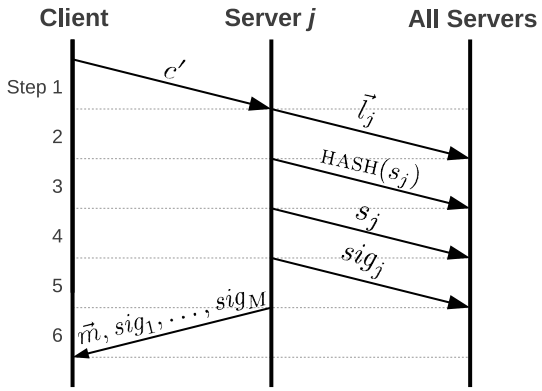



Figure 4: Dissent DC-net protocol.

disconnect at arbitrary points during the round, or otherwise fail to deliver their ciphertexts before a deadline. The servers first collect those client ciphertexts that arrive in time, then agree among each other on the complete set of client ciphertexts available (the union of all servers' client ciphertext sets), and finally XOR these client ciphertexts with the pseudo-random strings each server shares with those clients to form the servers' ciphertexts. Thus, client delays or disconnections never require servers to interact with clients iteratively within the same round, as they would in standard DC-nets to obtain revised ciphertexts.

## 3.7 Participation and Anonymity Metrics

To ensure "strength in numbers," users may wish to send anonymous messages only when at least some number of other group members are online and participating. Since the servers know the set of clients who are online and successfully deliver ciphertexts each round, the servers publish a *participation count* for each round. A user who judges this count to be too low can continue to participate passively in the group but send only an empty ("null") message in each round until participation increases.

Servers can publish participation counts only for *past* rounds, but clients can come and go at any time. A client thus might decide to send a message on the basis of one round's high participation count, and submit a sensitive message in the next round, only to discover after the round completes that far fewer clients remained online or delivered their ciphertexts in time. A powerful adversary might even start a DoS attack against many honest clients just as a sensitive anonymous posting is anticipated, in hopes of isolating and exposing the poster this way.

To address such risks, if the last round's participation count was $P$, the servers will not complete the next round until at least $\alpha P$ clients submit ciphertexts, where $0 \leq \alpha \leq 1$ is a policy constant defined at group creation time. If fewer than $\alpha P$ clients submit ciphertexts by the round's

deadline, the servers keep waiting until at least $\alpha P$ clients show up, or until a much longer *hard timeout* occurs. On a hard timeout, the servers discard all clients' ciphertexts, report the round as failed, and publish a new participation count on whose basis the clients make fresh decisions for the next round. The fraction $\alpha$ thus limits the rate at which participation may decrease unexpectedly round-to-round.

While Dissent can guarantee users that a sensitive message will be posted only when participation is at some threshold level, participation count unfortunately offers only an estimate of anonymity set size. If some participants are dishonestly colluding with the adversary, a client is anonymous only among the set of *honest* participants. A group's risk of infiltration of course depends on how it is formed and managed. In our current approach where a group is defined by a static list of public keys (Section 3.2), the dishonest members are those whose public keys the adversary manages to persuade the group creator to include in the list at definition time, plus any formerly-honest members who the adversary might compromise after group formation. Since users cannot ultimately know how many of their peers may be "spies," estimating anonymity necessarily remains subjective.

## 3.8 Eliminating Empty Slot Overhead

In typical blogging or chat applications we expect many clients to be silent much of the time, sending null messages in most rounds and real messages only occasionally. To optimize this common case, Dissent's scheduling scheme gives each client two slots: a one-bit *request* slot and a variable-length *message* slot. Initially the message slot is *closed*, with length 0. When a client sets its request bit in round $r$, its message slot *opens* to a fixed size in round $r+1$. The message slot includes a length field, with which the client can adjust the slot's length in subsequent rounds: to send a larger message efficiently in round $r+2$, for example, or to close its message slot back to length 0.

A dishonest member could DoS attack another client by guessing when the victim will transmit and sending a 1 in the victim's request slot, cancelling the victim's open request. To address such attacks, a client first sets its request bit unconditionally, but if its slot fails to open, the client randomizes its request bit in subsequent rounds, ensuring success after $t+1$ rounds with probability $1-(\frac{1}{2})^t$.

## 3.9 The Accusation Process

Dishonest members can disrupt DC-nets in general by XORing non-zero bits into other members' slots, corrupting the victim's cleartext. Earlier approaches to this challenge used complex *trap* protocols [58], expensive pairing-based cryptography [36], or required a costly shuffle before every DC-nets round [20]. Herbivore [35] mitigates the risk of disruption by limiting clique size and the rate at which disruptors can join them, at the cost of small anonymity sets and potentially increased vulnerability when disruptors are common [10].

Dissent introduces a new accusation scheme that adds little overhead in the absence of disruptors, but enables the servers to identify and expel a persistent disruptor quickly with high probability. The overall scheme operates in three stages. First, the victim of a disruption must find a *witness bit* in some round's DC-net output, which we define as a bit that was 0 in the victim's cleartext, but which the disruptor flipped to a 1. Second, the victim anonymously broadcasts an *accusation*, a message signed with the victim's pseudonym key identifying the witness bit. Finally, the servers publish all PRNG outputs that contributed to the client and server ciphertexts at the witness bit position, using them to trace the client or server that XORed an unmatched 1 bit into this position. The signed accusation attests that the traced node must be a disruptor.

The first challenge is ensuring that a disruption victim can find a witness bit. If a disruptor could predict the victim's cleartext output—or discover it from other honest nodes' ciphertexts before computing its own ciphertext—then the disruptor could avoid leaving witness bits by flipping only 1 bits to 0 in the victim's slot. To make all cleartext bits unpredictable, clients apply a cryptographic padding scheme analogous to OAEP [7]. The client picks a random seed $r$, generates a one-time pad $s = \text{PRNG}\{r\}$, XORs it with the original message $m$, and transmits $r||m \oplus s$ in the client's message slot. Since clients submit their ciphertexts before the servers compute theirs, and the commitment phase in Algorithm 2 prevents dishonest servers from learning honest servers' ciphertexts before computing their own, any disruptive bit-flip has a $1/2$ chance of producing a witness bit.

The second challenge is enabling the victim to transmit its accusation: if it did so via DC-nets, the disruptor could simply corrupt that transmission as well. To avoid this catch-22, Dissent falls back on the less efficient but disruption-resistant verifiable shuffle it uses for scheduling. Each client's DC-net message slot includes a $k$-bit *shuffle request* field, which the client normally sets to 0. When a disruption victim identifies a witness bit, it sets its shuffle request field in subsequent rounds to a $k$-bit random value. Any nonzero value signals the servers to start an accusation shuffle, in which the victim transmits its signed accusation. The disruptor may try to squash the shuffle request, but succeeds with at most $1-(\frac{1}{2})^k$ chance, and the victim simply retries until it succeeds.

The final challenge is tracing the actual disruptor. An accusation consists of the round number in which the dis-

ruption occurred, a slot index $\pi(i)$, and the index $k$ of a bit the disruptor flipped from 0 to 1 in this slot, all signed by the slot owner's pseudonym key, $k'_{\pi(i)}$. On receiving an accusation, the servers verify its signature, and check that the indicated bit was indeed output as 1. The servers then recompute and exchange all the individual PRNG bits that the clients and servers should have XORed together to compute their ciphertexts: $c_{ij}[k]$ in Algorithm 1 and $s_{ij}[k]$ in Algorithm 2. Each server independently attempts to find a mismatch, where: (a) a server did not transmit the full set of client ciphertext bits; (b) the accumulation of transmitted bits do not match what the server sent out earlier: $s_j[k] \neq (\bigoplus_{i \in l} s_{ij}[k]) \oplus (\bigoplus_{i \in l'_j} c_i[k])$; or (c) the client's ciphertext bit does not match the accumulation across servers: $\bigoplus_j s_{ij}[k] \neq c_i[k]$, for some client $i$. The first two cases trivially expose a server as dishonest. In the final case, each server requests from client $i$ a rebuttal on why the set of server bits, $s_{ij}[k]$, are incorrect, namely which server equivocated. An honest client can respond with the malicious server's identity, their shared secret, and proof of this shared secret.

## 3.10 Scheduling via Verifiable Shuffles

Dissent uses verifiable shuffles [11, 32, 44] both to schedule and distribute pseudonym keys for subsequent DC-nets rounds, and for transmitting accusations to servers. Clients submit messages (or keys to be anonymized) to the shuffle protocol, and the shuffle outputs a random permutation of these messages (or keys), such that no subset of clients or servers knows the permutation. Dissent depends minimally on the shuffle's implementation details, so many shuffle algorithms should be usable.

Dissent uses Neff's verifiable shuffle [44] to create verifiable secret permutations, and Chaum-Pedersen proofs [15] for verifiable decryptions. To shuffle, each client submits an ElGamal-encrypted group element. In a general message shuffle, clients embed their messages within a group element, encrypt it with a combination of all server keys, then transmit it to first server, who shuffles the input and removes a layer of encryption. Each server shuffles and decrypts in turn, until the last server reveals the cleartexts and distributes them to all clients.

The design supports both general message shuffles and more efficient *key shuffles*. Since Neff's algorithm shuffles ElGamal ciphertexts, general messages must be embedded within group elements. The entries of a key shuffle are already group elements, however, thus requiring no message embedding. Key shuffles also permit the use of more computationally efficient groups that are suitable for keys but not for message embedding.

## 3.11 Limitations

This section discusses a few of Dissent's shortcomings and possible ways to address them in future work.

**Large networks with many groups** Our evaluations (Section 5) demonstrate that a single Dissent network can accommodate over 5,000 clients. To be broadly usable at Internet scale, Dissent must scale to much larger network sizes, of hundreds of thousands of nodes or more. One way to serve very large networks would be to adopt a technique introduced by Herbivore [35]: break the overall network into smaller parallel Dissent groups—with tens of servers and thousands of clients each. A secure join protocol, as in Herbivore, could protect a single session from being overrun with Sybil identities.

**Intersection attacks** Dissent's traffic analysis resistance does not protect against *membership intersection attacks*, where an adversary correlates linkable anonymous transmissions to changes in clients' online status. If an anonymous blogger posts a series of messages, each signed by the same pseudonym but posted in different rounds, and the adversary sees that only Alice was online in *all* of those rounds (though many other group members were present in *some* rounds), the adversary might pinpoint Alice as the blogger. There is no perfect defense against intersection attacks when online status changes over time [39]. Dissent users could gain some protection against the intersection attack by avoiding linkable anonymous transmissions (e.g., the use of pseudonyms). Alternatively, users could adopt a "buddy system," transmitting linkable cleartexts only when *all* of a fixed set of "buddies" are also online. With certain caveats, this discipline ensures that a user's anonymity set includes at least his honest buddies, at the availability cost of making the user unable to transmit (safely) when *any* buddy is offline.

**Handling server failure** Dissent addresses network churn only among clients: if a server goes offline, the protocol halts completely until all servers are available again or the group is administratively re-formed to exclude the failed server (which currently amounts to creating a new group). We expect that Byzantine fault-tolerance techniques [12] could be adapted to mask benign or malicious server failures, at a cost of imposing a stronger security assumption on the servers. In a BFT group designed to tolerate $f$ concurrent failures, for example, client anonymity would likely depend on at least $f+1$ servers being honest, rather than just one. A malicious group leader could form a live "view" deliberately excluding up to $f$ honest and online servers, replacing them with $f$ dishonest servers who appear live and well-behaved but privately collude in attempt to de-anonymize clients.

**Group management and server selection** As discussed in Section 3.2, Dissent groups currently contains a static list of clients and servers; allowing more dynamic group administration while maintaining security remains an important challenge. In a public Dissent deployment with hundreds of servers and thousands of parallel groups, users would benefit especially from automatic server selection. Since the user must trust at least one of the servers, a server selection algorithm might have to consider which servers user trusts, how close the user is to which servers, a server's reliability, and other security and performance factors. Dissent's server selection problem is likely analogous to the path selection problem in Tor [6, 28], and might build on prior work on this topic.

**Mobile devices** In the United States, consumers now use mobile phones more for Internet browsing and non-voice data transfer than for making phone calls [61]. As everyday computing shifts to mobile devices, an ongoing challenge is to offer users the same privacy protections on phones as they would have on desktop computers [8,33,34]. We have yet to deploy or test Dissent on mobile devices, but expect Dissent's computation and communication optimizations to be useful in this context.

**Formal security analysis** While Dissent is based on techniques with formal security proofs [14, 15, 44], a full formal analysis of Dissent remains for future work.

# 4 Implementation

This section describes the current Dissent prototype and how we have applied it to two anonymous communication use cases: wide-area group messaging and local-area anonymous Web browsing.

## 4.1 Prototype Overview

We have implemented Dissent in C++ with the Qt framework and the CryptoPP cryptography library. The prototype implements the complete Dissent anonymity protocol along with the accountability sub-protocols described in Section 3.9. The system assumes the existence of a certificate authority (or other entity) that manages the long-term public keys of all servers and clients. The prototype also assumes that participants have used an outside channel to agree upon a common set of servers. Source code may be found at the Dissent project home page.[1]

User applications interact with a node running our Dissent prototype using HTTP API calls or a SOCKS proxy interface. The HTTP API allows clients to post raw messages (byte-strings) directly into the protocol session. The prototype's SOCKS v5 proxy allows users to tunnel TCP and UDP traffic flows transparently through the Dissent protocol session. One or more nodes in the Dissent network serve as SOCKS entry nodes, which listen for incoming SOCKS proxy requests from user applications (e.g., Skype or Firefox). The entry node accepts SOCKSified traffic flow from the user application, assigns the flow a random identifier (to allow a receiving node to distinguish between many flows), adds destination IP and port headers to the flow, and sends it into the active Dissent protocol round. A single SOCKS exit node (who is a non-anonymous protocol participant) reads the tunneled traffic from the Dissent protocol round, forwards it over the public network to the destination server, and sends the response back through the Dissent session.

[1] `http://dedis.cs.yale.edu/2010/anon/`

## 4.2 Anonymous Microblogging Application

Dissent's decentralized architecture and trust model make it potentially attractive as a substitute for commercial microblogs in high-risk anonymous communication scenarios. Since Dissent relies on no single trusted party, we expect it to be much more challenging even for a powerful adversary—such as an authoritarian government or its state-controlled ISP—to identify an anonymous blogger without compromising *all* participating Dissent servers.

In our evaluation section, we present performance results for a prototype microblogging system running on PlanetLab with up to 2,000 nodes and DeterLab with up to 5,000 nodes. A simple chat-like Web interface allows users to post short messages into an Dissent protocol session using our HTTP API. Our results suggest Dissent could form a practical platform for Internet-scale microblogging in situations requiring stronger security properties than the current commercial platforms offer.

## 4.3 Local-Area Web Browsing

When deployed in a local-area network, Dissent can provide interactive communication with local-area anonymity: requests are anonymous among a local set of users. To demonstrate this use of Dissent, we have developed *WiNoN*, a system that uses virtual machines to isolate a user's identifiable OS environment from their anonymous browsing environment, an important issue given that browser signatures have a reasonable chance of uniquely identifying a user [27].

In WiNoN, depicted in Figure 5, the Dissent client software runs on the host OS with network traffic from the WiNoN VM tunneled through the Dissent SOCKS proxy. Since applications in the WiNoN VM have no access to the network interface or to the user's non-anonymous storage, they are unable to learn the user's long-term identity (unless the user inadvertently enters some identifiable information into the WiNoN VM).

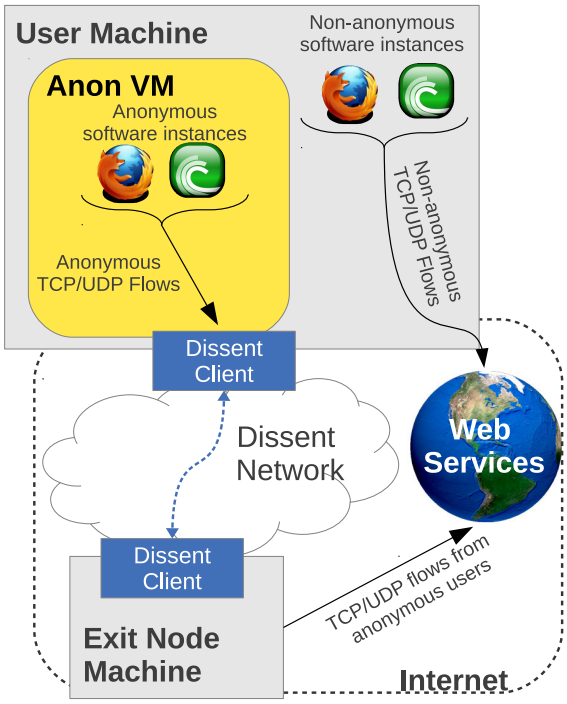



Figure 5: WiNoN system diagram. Traffic from the anonymous VM flows through an Dissent tunnel to an exit node. The exit node reads traffic from the tunnel and forwards it onto the Internet on behalf of the WiNoN client.

On simulated WiFi networks with tens of nodes and typical bandwidth and delay parameters, we find the WiNoN anonymization network fast enough for browsing the Internet and streaming videos. Prior work uses virtual machines to isolate network environments [41] and tunnel traffic through Tor [55]. No previous system to our knowledge, however, enables a user to run Flash movies, Skype, and other untrusted applications safely and anonymously.

# 5 Evaluation

This section first examines Dissent's ability to handle unreliable client nodes. Next we evaluate Dissent's performance and scalability in instant-messaging and data sharing scenarios with varying numbers of clients and servers. We then explore the costs of different protocol stages: the initial key shuffle, a DC-net exchange, and finally the accusation process. Finally, we evaluate the performance of Dissent applied to the WiNoN Internet browsing application described in section 4.3.

In our evaluations, we used the Emulab [29], DeterLab [23], and PlanetLab [16] testbeds, and nodes from Amazon's EC2 service. Emulab and DeterLab offer controlled, repeatable conditions on isolated networks, while

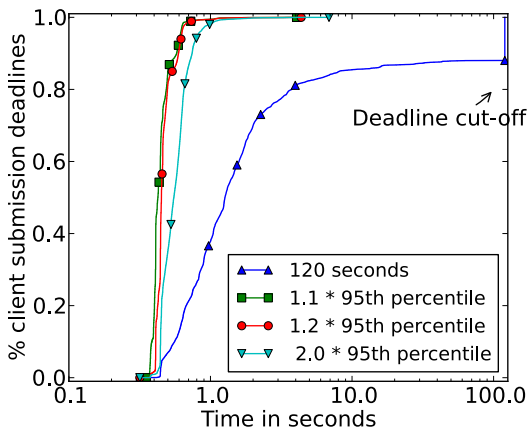



Figure 6: A CDF plot demonstrating the time for a message exchange to complete when using four message window policies.

we used PlanetLab nodes on the public Internet to offer a more realistic test of Dissent's ability to handle client delays and churn. For our evaluations on the public Internet, we used eight server machines—one located at our university and seven at EC2 sites—located at unique locations on four different continents, and we used the entire set of available PlanetLab nodes as clients. We indicate the exact number of PlanetLab clients where applicable.

## 5.1 Slow and Unreliable Clients

On public networks, distributed systems must cope with slow and unreliable machines [47]. Dissent's servers prevent slow nodes from impeding the protocol's overall progress by imposing a ciphertext submission window. Once the client submission time window has closed, servers continue executing the protocol even if every client has not submitted a ciphertext. Larger windows potentially allow more clients to participate in each message exchange, but increase messaging latency. Smaller windows size reduce latency of exchanges but might prevent slower clients from participating.

To help us select an effective window closure policy for our evaluations on PlanetLab, we collected a data trace from a Dissent deployment with over 500 clients running on PlanetLab nodes and eight servers running on EC2, using a static window size of 120 seconds. The exact number of clients varied over the course of the 24-hour evaluation period. We used the data from this PlanetLab experiment to test a variety of window closure policies.

To ensure that most clients are able to participate in each message exchange, we do not close the submission window until at least 95% have submitted messages. Once

95% of clients submit messages, we multiply the time elapsed by a constant factor to determine window time.

The fraction of clients who missed the submission window decreased as this multiplicative constant increased: 1.1×: 2.3%, 1.2×: 1.5%, and 2×: 0.5%. The data in Figure 6 demonstrate that the client submission time is not very sensitive to multiplicative constant used. For the rest of the evaluation, we chose the 1.1× policy, since there was not significant variation among the three.

Regardless of the specific window closure policy chosen, Figure 6 demonstrates the importance of insulating the group's progress from that of its slowest clients in an unpredictable environment like PlanetLab. In the baseline case where the servers wait until *all* clients submit or a 120-second hard deadline is reached, 50% of DC-net rounds are delayed by "stragglers" by an order of magnitude or more compared with early-cutoff policies, and 15% of rounds are delayed until the 120-second hard deadline versus almost none with early cutoff policies.

## 5.2 Wide-Area Applications

To evaluate Dissent's usability in wide-area microblogging or data sharing scenarios, as described in Section 4.2, we evaluated the protocol on both DeterLab [23] and PlanetLab [16]. On DeterLab, which offered controlled test conditions and greater hardware resources, we evaluated both a microblog and data sharing like behavior; we evaluated only the microblog scenario on PlanetLab.

To simulate a plausible traffic load in the microblog scenario, a random 1% of all clients submit 128-byte messages during any particular round. In the data sharing scenario, one client transmits a 128KB message per round.

Due to time and resource limitations, the DeterLab evaluation used two system topologies: 32 servers with 10 client machines per server, and 24 servers with 12 client machines per server, for a total of 320 and 288 client machines respectively. To simulate a larger number of Dissent client participants than we had physical testbed machines for, we ran up to 16 Dissent client processes on each client machine, for up to 5120 client processes.

In the testbed topology, servers shared a common 100 Mbps network with 10 ms latency, while clients shared a 100 Mbps uplink with 50 ms latency to their common server. We intended the servers to share a 1000 Mbps network, but the testbed did not support this configuration.

For the PlanetLab tests, we deployed 17 servers: 16 on EC2 using US East servers and one control server located at Yale University. The latency between Yale and EC2 was approximately 14 ms round trip. This clustered setup is intended to represent a deployment scenario in which multiple organizations offer independently-managed Dissent servers physically co-located in the same or geographically nearby data centers, facilitating high-bandwidth and low-latency communication among the servers while keeping their management decentralized for security.

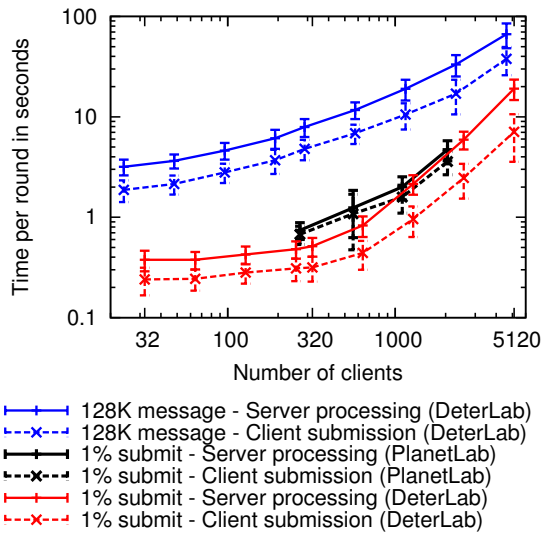



Figure 7: Time per round in microblog (1% submit) and data sharing scenarios, for varying number of clients.

Figure 7 shows the system's scalability with client load by varying the number of clients relying on a static set of 32 servers. Figure 8 in turn varies the number of servers while maintaining a static set of 640 clients. At smaller group sizes, additional servers do not benefit performance. As demands on the servers scale, however, their utility becomes more apparent, especially in the 128K message scenario. Performance appears to be dominated by client delays, namely the time between clients receiving the previous round's cleartext and the servers receiving the current round's ciphertext message. However, in comparing the PlanetLab evaluation to the DeterLab evaluation and server size of 1, we can ascertain that latency between servers tends to dominate delays in that environment, though computational load is not negligible.

The prototype shows greatest usability for group sizes up to 1,000; thereafter delays become longer than 1 second in the microblogging scenario. At best, delays were on the order of 500 to 600 ms for 32 to 256 clients. In the static client network, varying server count, showed time increases on server-related aspects of the protocol but reduced time on client-related aspects. We therefore expect that with greater demand—either in terms of nodes or bandwidth—client-related costs are likely to dominate.

In comparing the microblogging and 128K message scenarios, the graphs suggest that bandwidth tends to dominate for larger messages and latency for smaller messages. Most importantly, the evaluations suggest that

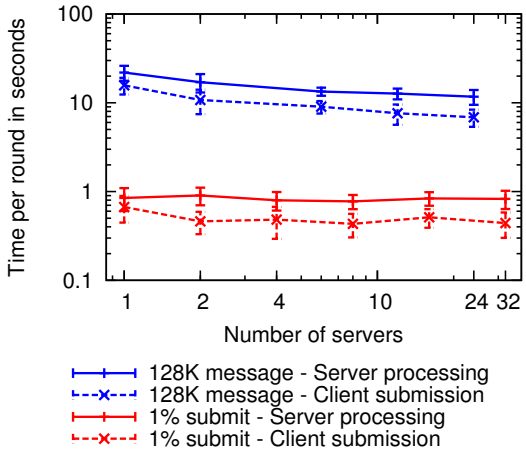



Figure 8: Time per round in microblog (1% submit) and data sharing scenarios, for varying number of servers supporting 640 clients.

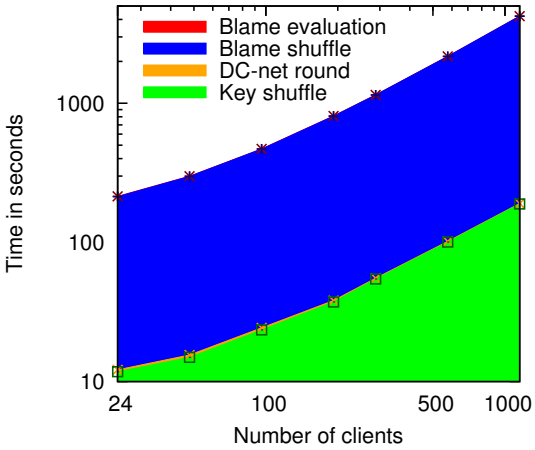



Figure 9: Time elapsed during a whole protocol run for varying client sizes, 24 servers, and 128 byte messages.

Dissent can support delay-sensitive applications like microblogs and instant messaging.

## 5.3 Full System Evaluation

While the previous experiments focused on measuring DC-nets rounds, the primary focus of this paper, we now explore time durations in a single full execution of the entire Dissent protocol: key shuffle, a single DC-net exchange, accusation shuffle, and accusation tracing. Our results shown in Figure 9 used the same DeterLab configuration consisting of 24 servers with 12 clients each configuration as described in the previous section. In contrast to verifiable mix-nets, the Dissent protocol's DC-nets round is extremely efficient, accounting for a negligible portion of total time in large groups.

The time difference between accusation and key shuffles illustrate the performance benefits of the key shuffle discussed in Section 3.10. In small groups the accusation shuffle is reasonably fast, but in larger groups its cost increases quickly, to over an hour for 1,000-client groups.

## 5.4 Web Browsing: Dissent and Tor

To explore the practicality of Dissent for local-area anonymity as described in section 4.3, we deployed a smaller-scale Dissent network of 5 servers and 24 clients on the Emulab [29] network testbed. The testbed's experimental network topology approximated the characteristics of a small WiFi network: each node was connected to a central switch via a 24 Mbps link with 10 ms of latency. One of the servers acted as a gateway connecting the private test network to the public Internet.

In this environment, we ran an automated HTTP browser on one of the client nodes to download the index pages from each site on the Alexa "Top 100" Web sites [3] from the real public Web server. For each index page, the client requested the HTML page and then recursively and concurrently requested dependent assets (images, CSS, JS, etc.). Although we used an automated HTTP browser for these trials, Dissent supports standard Web browsers as well.

We used four different network configurations to test Dissent's performance under four deployment scenarios. In the first scenario, no anonymity, the gateway connects directly to the public Internet. The second scenario, Tor alone, shows the performance of state-of-the-art wide-area anonymous Internet access. We emphasize that we compare with Tor only to provide a general reference point for gauging Dissent's usability: this is by no means an "apples-to-apples" comparison since the functionality, scale, security properties, and network conditions of the two systems under test are incomparable in myriad ways.

The third test scenario, a local-area deployment of Dissent, is intended to test whether Dissent is fast enough for interactive browsing on local-area networks. The fourth scenario, a serial composition of Dissent and Tor, considers the performance of a configuration offering "best of both worlds" security, where we compose a local-area Dissent network with the public Tor network. This configuration offers users Tor's wide-area anonymity against limited-strength adversaries, combined with Dissent's local-area security against adversaries who might use traffic analysis to de-anonymize Tor circuits.

The results in Figure 10 indicate that anonymous web browsing under the local-area deployment of Dissent we tested performs comparably to Tor, suggesting that users are likely to find some Dissent configurations similarly

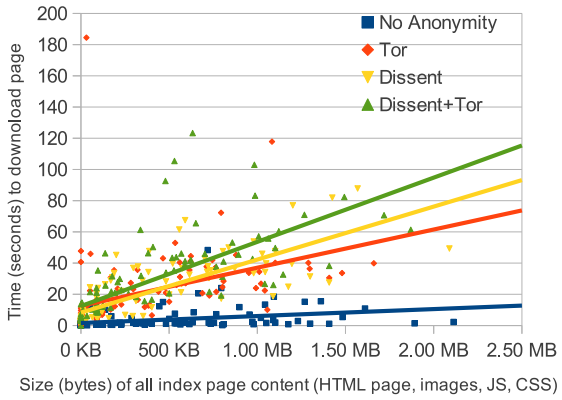



Figure 10: Download times for the Alexa "Top 100" home pages in which nodes access the Internet over an Dissent network running on an Emulab-simulated wireless LAN, over the public Tor network, and over a composition of wLAN Dissent and Tor.

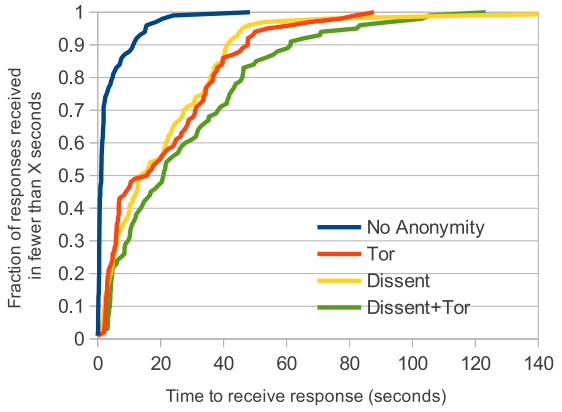



Figure 11: CDF of download times presented in Figure 10.

usable under appropriate network conditions. On average, downloading 1MB of Web content took 10 seconds with no anonymization, it took 40 seconds through Tor, 45 seconds with Dissent, and 55 seconds with Dissent and Tor together. Comparing Dissent+Tor with Tor alone, this data suggests that a user willing to tolerate a 35% slowdown could retain Tor's wide-area benefits while gaining traffic analysis resistant anonymity in the user's local area.

Figure 11 shows a CDF of page download times, showing that a client using Tor downloads the first 50% of Web pages in 15 seconds, while a client using Dissent+Tor downloads 50% of Web pages in just under 20 seconds. We expect that many users, especially those with strong security requirements, might find a few extra seconds per Web page a reasonable price for local-area security.

# 6 Conclusion

This paper has made the case that by delegating collective trust to a decentralized group of servers, strong anonymity techniques offering traffic analysis resistance may be adapted and scaled to offer anonymity in groups of thousands of nodes, two orders of magnitude larger than previous systems offering strong anonymity. Through its novel client/server DC-nets model, Dissent is able to accommodate anonymity set sizes of up to 5,000 members, while maintaining end-to-end latency low enough to enable wide-area interactive messaging. In local-area settings, Dissent is fast enough to handle interactive Web browsing while still offering users strong local anonymity guarantees. Although Dissent represents a step towards strong anonymous communication at large Internet scales, many challenges remain for future work, such as further scalability and robustness improvements and protection against long-term intersection attacks.

## Acknowledgments

We would like to thank Michael F. Nowlan, Vitaly Shmatikov, Joan Feigenbaum, Ramakrishna Gummadi, Emin Gün Sirer, Jon Howell, Roger Dingledine, and the anonymous OSDI reviewers for their extraordinarily helpful comments, as well as the Emulab and DeterLab folks for their time and effort. This material is based upon work supported by the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contract No. N66001- 11-C-4018.

# References

[1] T. G. Abbott, K. J. Lai, M. R. Lieberman, and E. C. Price. Browser-based attacks on Tor. In *PETS*, 2007.

[2] B. Adida. *Advances in cryptographic voting systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2006.

[3] Alexa top 500 global sites, April 2012. `http://www.alexa.com/topsites`.

[4] Anonymizer, September 2012. `http://anonymizer.com/`.

[5] J. M. Balkin. Digital speech and democratic culture: A theory of freedom of expression for the information society. *Faculty Scholarship Series*, 2004. Paper 240.

[6] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-resource routing attacks against Tor. In *WPES*, Oct. 2007.

[7] M. Bellare and P. Rogaway. Optimal asymmetric encryption – how to encrypt with RSA. In *Eurocrypt*, May 1994.

[8] A. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2003.

[9] S. L. Blond, P. Manils, A. Chaabane, M. A. Kaafar, A. Legout, C. Castellucia, and W. Dabbous. De-anonymizing BitTorrent users on Tor, Apr. 2010. `http://arxiv.org/abs/1004.1267`.

[10] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of service or denial of security? How attacks on reliability can compromise anonymity. In *ACM CCS*, Oct. 2007.

[11] J. Brickell and V. Shmatikov. Efficient anonymity-preserving data collection. In *ACM KDD*, Aug. 2006.

[12] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, Feb. 1999.
[13] S. Chakravarty, A. Stavrou, and A. D. Keromytis. Identifying proxy nodes in a Tor anonymization circuit. In *SITIS*, Nov. 2008.
[14] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, Jan. 1988.
[15] D. Chaum and T. P. Pedersen. Wallet databases with observers. CRYPTO, 1992.
[16] B. Chun et al. PlanetLab: An overlay testbed for broad-coverage services. In *ACM CCR*, July 2003.
[17] J. Clark, P. C. van Oorschot, and C. Adams. Usability of anonymous web browsing: an examination of tor interfaces and deployability. In *SOUPS*, 2007.
[18] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, July 2000.
[19] M. Clarkson, S. Chong, and A. Myers. Civitas: Toward a secure voting system. In *IEEE SP*, may 2008.
[20] H. Corrigan-Gibbs and B. Ford. Dissent: accountable anonymous group messaging. In *ACM CCS*, Oct. 2010.
[21] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a Type III anonymous remailer protocol. In *IEEE SP*, May 2003.
[22] R. Deibert, J. Palfrey, R. Rohozinski, and J. Zittrain. *Access Denied: The Practice and Policy of Global Internet Filtering*. MIT Press, Jan. 2008.
[23] Deterlab network security testbed, September 2012. `http://isi.deterlab.net/`.
[24] R. Dingledine and J. Appelbaum. How governments have tried to block Tor, 2012. Tor project presentation, `https://svn.torproject.org/svn/projects/presentations/slides-28c3.pdf`.
[25] R. Dingledine and N. Mathewson. Design of a blocking-resistant anonymity system, Nov. 2006. Tor Project technical report, `https://svn.torproject.org/svn/projects/design-paper/blocking.html`.
[26] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *USENIX Security Symposium*, 2004.
[27] P. Eckersley. How unique is your web browser? In *PETS*, July 2010.
[28] M. Edman and P. Syverson. As-awareness in tor path selection. In *ACM CCS*, 2009.
[29] Emulab network emulation testbed, September 2012. `http://emulab.net/`.
[30] Federal Trade Commission. Protecting consumer privacy in an era of rapid change, Dec. 2010. Preliminary FTC Staff Report.
[31] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM TOCS*, Feb. 2002.
[32] J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In *CRYPTO*, Aug. 2001.
[33] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *IEEE ICDCS*, june 2005.
[34] G. Ghinita, P. Kalnis, and S. Skiadopoulos. PRIVE: anonymous location-based queries in distributed mobile systems. In *16th WWW*, May 2007.
[35] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A Scalable and Efficient Protocol for Anonymous Communication. Technical Report 2003-1890, Cornell University, February 2003.
[36] P. Golle and A. Juels. Dining cryptographers revisited. *Eurocrypt*, May 2004.
[37] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? In *ACM CCS*, Oct. 2007.
[38] N. Jones, M. Arye, J. Cesareo, and M. J. Freedman. Hiding amongst the clouds: A proposal for cloud-based onion routing. In *FOCI*, Aug. 2011.
[39] D. Kedogan, D. Agrawal, and S. Penz. Limits of anonymity in open environments. In *5th International Workshop on Information Hiding*, Oct. 2002.
[40] S. F. Kreimer. Technologies of protest: Insurgent social movements and the First Amendment in the era of the Internet. *University of Pennsylvania Law Review*, October 2001.
[41] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. *Tech Trend Notes*, 2000.
[42] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach. AP3: Cooperative, decentralized anonymous communication. In *ACM SIGOPS EW*, Sept. 2004.
[43] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Security and Privacy*, May 2005.
[44] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *ACM CCS*, Nov. 2001.
[45] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *WPES*, Oct. 2011.
[46] J. Preston. Facebook Officials Keep Quiet in Its Role in Revolts, Feb. 2011. `http://www.nytimes.com/2011/02/15/business/media/15facebook.html`.
[47] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *WORLDS*, 2005.
[48] S. Sengupta. Rushdie runs afoul of web's real-name police. *New York Times*, Nov. 2011.
[49] E. G. Sirer et al. Eluding carnivores: File sharing with strong anonymity. In *ACM SIGOPS EW*, Sept. 2004.
[50] R. Smits et al. BridgeSPA: Improving Tor bridges with single packet authorization. In *WPES*, Oct. 2011.
[51] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. Flash cookies and privacy, Aug. 2009.
[52] F. Stajano and R. Anderson. The cocaine auction protocol: On the power of anonymous broadcast. In *Information Hiding Workshop*, Sept. 1999.
[53] E. Stein. Queers anonymous: Lesbians, gay men, free speech, and cyberspace. *Harvard Civil Rights-Civil Liberties Law Review*, 2003.
[54] P. Syverson. Sleeping dogs lie on a bed of onions but wake when mixed. In *HotPETs*, July 2011.
[55] Tails: The amnesic incognito live system, September 2012. `https://tails.boum.org/`.
[56] A. Teich, M. S. Frankel, R. Kling, and Y. Lee. Anonymous communication policies for the Internet: Results and recommendations of the AAAS conference. *Information Society*, May 1999.
[57] E. Volokh. Freedom of speech and information privacy: The troubling implications of a right to stop people from speaking about you. *Stanford Law Review*, May 2000.
[58] M. Waidner and B. Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Eurocrypt*, Apr. 1989.
[59] J. D. Wallace. Nameless in cyberspace: Anonymity on the internet, Dec. 1999. Cato Briefing Paper No. 54.
[60] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Scalable anonymous group communication in the anytrust model. In *EuroSec*, Apr. 2012.
[61] J. Wortham. Cellphones Now Used More for Data Than for Calls, May 2010. `http://www.nytimes.com/2010/05/14/technology/personaltech/14talk.html`.
[62] J. Wright, T. de Souza, and I. Brown. Fine-grained censorship mapping information sources, legality and ethics. In *FOCI*, Aug. 2011.

# Efficient patch-based auditing for web application vulnerabilities

Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich
*MIT CSAIL*

## Abstract

POIROT is a system that, given a patch for a newly discovered security vulnerability in a web application, helps administrators detect past intrusions that exploited the vulnerability. POIROT records all requests to the server during normal operation, and given a patch, re-executes requests using both patched and unpatched software, and reports to the administrator any request that executes differently in the two cases. A key challenge with this approach is the cost of re-executing all requests, and POIROT introduces several techniques to reduce the time required to audit past requests, including filtering requests based on their control flow and memoization of intermediate results across different requests.

A prototype of POIROT for PHP accurately detects attacks on older versions of MediaWiki and HotCRP, given subsequently released patches. POIROT's techniques allow it to audit past requests 12–51× faster than the time it took to originally execute the same requests, for patches to code executed by every request, under a realistic MediaWiki workload.

## 1 Introduction

New security vulnerabilities are routinely discovered in many web applications, and web application developers frequently release patches for such bugs in their software. Once an administrator learns about a new vulnerability and applies a patch to prevent new attacks, the administrator may want to check whether anyone exploited the bug before the patch was applied, in order to take any necessary remedial measures. This check is important to ensure that no data was leaked or that the attacker did not leave any back doors for later intrusions, yet today this check remains a mostly manual process.

As one example, consider the HotCRP conference management software [21], which recently had a information disclosure bug that allowed paper authors to view a reviewer's private comments meant only for program committee members [22]. After applying the patch for this vulnerability, an administrator of a HotCRP site would likely want to check if any comments were leaked as a result of this bug. In order to do so, the administrator would have to manually examine the patch to understand what kinds of requests can trigger the vulnerability, and then attempt to determine suspect requests by manually poring over logs (such as HotCRP's application-level log, or Apache's access log) or by writing a script to search the logs for requests that match a specific pattern. This process is error-prone, as the administrator may miss a subtle way of triggering the vulnerability, and the logs may have insufficient information to determine whether this bug was exploited, making every request potentially suspicious. For example, there is no single pattern that an administrator could search for to find exploits of the HotCRP bug mentioned above.

Manual auditing by the administrator may be an option for HotCRP sites with a small number of users, but it is prohibitively expensive for large-scale web applications. Consider the recent vulnerability in Github—a popular collaborative software development site—where any user was able to overwrite any other user's SSH public key [27], and thus modify any software repository hosted on Github. After Github administrators learned about and patched the vulnerability, their goal was to determine whether anyone had exploited this vulnerability and possibly altered user data. Although the patch was just a one-line change in source code, it was difficult to determine who may have exploited this vulnerability in the past. As a result, Github administrators disabled all SSH public keys as a precaution, and required users to re-confirm their keys [13]—an intrusive measure, yet one that was necessary because of the lack of alternatives.

This paper presents POIROT, a system that can audit a web application's past requests and identify requests that potentially exploited a vulnerability, given a patch that fixes the vulnerability. POIROT focuses on vulnerabilities in server-side application code, which includes seven of the top ten web application vulnerabilities [29].

POIROT adopts the record-and-replay approach from previous systems [9, 33], and records each request to the web application during the application's normal execution. When a patch is released, the administrator invokes POIROT, which re-runs past requests on two versions of the application source code—one with and one without the patch—and compares the results. If the results are the same (including any side-effects such as modifying files or issuing SQL queries), POIROT concludes that the request did not exploit the vulnerability. Conversely, if the results differ, POIROT reports the request to the administrator as a possible attack, along with a diff of the results with and without the patch.

POIROT's key contribution lies in performance. The closest related work is Warp [9], which undoes the effects of past attacks given a patch by re-executing every request that touched a patched file. In the worst case, a developer may patch code that is executed by every request, in which case auditing several months worth of requests with Warp would take yet another several months on production servers. POIROT shows that it is possible to audit 1–2 orders of magnitude faster than simply re-running every request, even for the challenging patches that modify code executed by *every* request. POIROT's design speeds up auditing by leveraging three techniques, as follows.

First, POIROT performs *control flow filtering* to avoid re-executing requests that did not invoke patched code. To filter out these requests, POIROT records a control flow trace of basic blocks executed by each request during normal execution, and indexes them for efficient lookup. For a given patch, POIROT computes the set of basic blocks modified by the patch, and determines the set of requests that executed those basic blocks. This allows POIROT to skip many requests for patches that modify rarely used code.

Second, POIROT optimizes the two re-executions of each request—one with the patch and one without—by performing *function-level auditing*. Each request is initially re-executed using one process. When a patched function is invoked, POIROT forks the process into two, executes the patched code in one process and the unpatched code in another, and compares the results. If the results don't match, POIROT marks the request as suspect and stops re-executing that request, and if the results are identical, POIROT kills off one of the forked processes and continues re-executing in the other process. Function-level auditing improves performance since forking is often cheaper than re-executing long runs of common application code.

As an extension of function-level auditing, POIROT terminates re-execution of a request if it can determine, based on previously recorded control flow traces, that this request will not invoke any patched functions for the rest of its re-execution. We call this *early termination*.

Third, POIROT eliminates redundant computations—identical instructions processing identical data—that are the same across *different* requests, using a technique we call *memoized re-execution*. POIROT keeps track of intermediate results while re-executing one request, and reuses these results when re-executing subsequent requests, instead of recomputing them. The remaining code re-executed for subsequent requests can be thought of as a dynamic slice for the patched code [3], and is often 1–2 orders of magnitude faster than re-executing the entire request.

An evaluation of a POIROT prototype for PHP-based applications with MediaWiki and HotCRP shows that POIROT can accurately and efficiently detect past intrusions given a patch, and that the three techniques mentioned above are important to achieve good performance. Out of 34 real MediaWiki security patches, POIROT takes 1,013 seconds to audit a patch in the worst case (when the patch affects all requests) for a workload that takes 12,116 seconds to complete during normal execution. For a realistic workload based on Wikipedia traces, POIROT imposes 15% CPU overhead during normal execution and requires 5 KB of storage per request, which amounts to 3.3 GB per day for one server. Finally, POIROT has no false negatives, and incurs no false positives for most patches.

The rest of this paper is organized as follows. §2 starts off with an overview of POIROT's design and its workflow. §3, §4, and §5 describe POIROT's three key techniques for minimizing re-execution. §6 discusses our prototype implementation, and §7 evaluates it. §8 touches on some of the limitations of POIROT. §9 compares POIROT with related work, and §10 concludes.

## 2 Overview

To understand how POIROT helps an administrator automate the auditing process, suppose that some request exploited the HotCRP vulnerability mentioned in the previous section, and saw confidential comments. When that request is re-executed by POIROT, the HTTP response with the patch applied will be different from that without the patch (since the response will not contain the comments), and the request will be flagged as suspect, leaving the administrator to decide on the appropriate remedy. On the other hand, requests that did not exploit the vulnerability will likely generate the same responses, and will not be flagged as suspect. Similarly, in the Github scenario mentioned earlier, an attack request that exploited the vulnerability would issue an SQL query to modify the victim's public key. When the attack is re-executed on patched code, the query will not be issued, and POIROT will report the discrepancy to the administrator.

More precisely, given a patch fixing a vulnerability in a web application, POIROT's goal is to identify a minimal set of requests that may have exploited the vulnerability. Conceptually, POIROT re-runs each past request to the web application twice—once each with the vulnerable and the patched versions of the application's source code—and compares the results of these runs. If the results are the same, the request is assumed to not exploit the vulnerability; otherwise, POIROT adds the request to a list of requests that may have exploited the vulnerability, to be further audited by the administrator.

A request's result in POIROT logically includes the web server's HTTP response, as well as any side effects of request execution, such as changes to the file system or queries issued to an SQL database. This ensures that

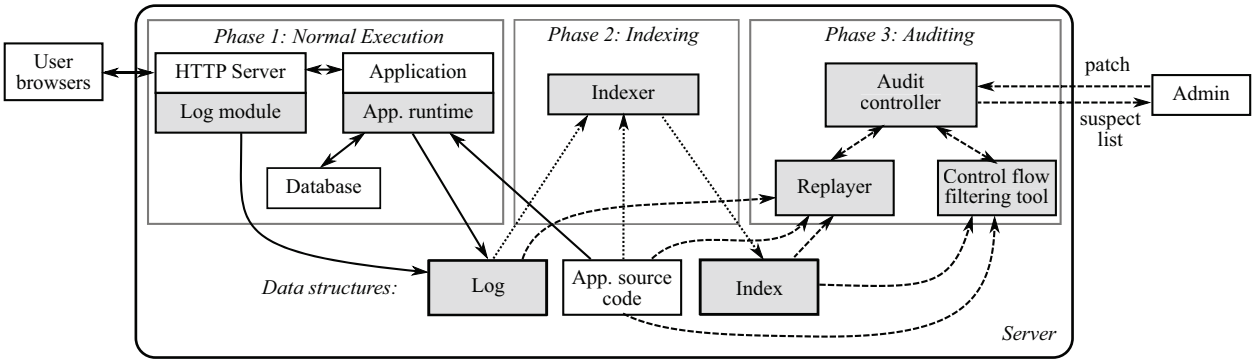



**Figure 1**: Overview of POIROT's design. Components introduced by POIROT are shaded. Solid lines, dotted lines, and dashed lines indicate interactions during normal execution, indexing, and auditing stages, respectively.

POIROT will catch both attacks that altered server state (e.g., modifying rows in a database), as well as attacks that affect the server's HTTP response.

POIROT consists of three phases of operation, as illustrated in Figure 1: normal execution, indexing, and auditing. The rest of this section describes these phases, and explains the assumptions and limitations of POIROT.

## 2.1 Logging during normal execution

In order to rerun a past request in a web application, POIROT needs to record the original inputs to the application code that were used to handle the request. Additionally, in order to perform control flow filtering, POIROT must record the original control flow path of each request.

During the normal execution of a web application, POIROT records four pieces of information about each request to a log. First, it records the request's URL, HTTP headers, any POST data, and the CGI parameters (e.g., the client's IP address). Second, it records the results of non-deterministic function calls made during the request's execution, such as calls to functions that return the current date or time, and functions that return a random number. Third, it records the results of calls to functions that return external data, such as calls to database functions. Finally, it records a control flow trace of the application code, at the level of basic blocks [5].

POIROT implements logging by extending the application's language runtime (e.g., PHP in our prototype implementation) and by implementing a logging module in the HTTP server.

It is up to the administrator to decide on how long to store POIROT's logs. For an application such as HotCRP, it may make sense to store all logs from the time when the conference starts, and audit every request if a vulnerability is discovered. For a larger-scale web site, such as Wikipedia, it may make sense to discard logs of old requests at some point (e.g., after several months), although POIROT would be unable to audit discarded requests for possible attacks.

## 2.2 Indexing

The second step in POIROT's auditing process is to build an index from the logs recorded during normal execution. The index contains two data structures, as follows.

The first data structure, called the *basic block index*, maps each basic block to the set of requests that executed that basic block, and is generated from the control flow traces recorded during normal execution. This data structure is used by POIROT's *control flow filtering* to efficiently locate candidate requests for re-execution given a set of basic blocks that have been affected by a patch.

The second data structure, called the *function call table*, is a count of the number of times each request invoked each function in the application, and is also generated based on the control flow traces recorded during normal execution. This data structure is used to implement the *early termination* optimization.

POIROT's indexing step can be performed on any machine, and simply requires access to the application source code as well as the control flow traces. Performing the indexing step before auditing (described next) both speeds up the auditing step and avoids having to re-generate these data structures for multiple audit operations.

## 2.3 Auditing

When a patch for a newly discovered security vulnerability is released, an administrator can invoke POIROT's auditing phase and supply the patch to POIROT. POIROT's auditing code requires access to the original log of requests, as well as to the index. POIROT first performs *control flow filtering* to filter out requests that did not invoke the patched code, and then uses *function-level auditing* and *memoized re-execution* to efficiently re-execute requests that did invoke the patched code. To ensure requests execute in the same way during auditing as they did during the original execution, POIROT uses the log to replay the original inputs (such as the URL and POST data), as well as the results of any non-deterministic functions and external I/O (e.g., SQL queries) that the application invoked. Note that POIROT does not require a past

snapshot of the database for re-executing requests: if the application issues a different SQL query during request re-execution—for which POIROT's log does not contain a recorded result—POIROT flags the request as a potential attack and stops re-executing that request. POIROT performs re-execution by modifying the language runtime (e.g., the PHP interpreter in our prototype), as we will describe later.

Once re-execution finishes, POIROT provides the administrator with a list of suspect requests that executed differently with the patched code than they did with the unpatched code, for further examination.

### 2.4 Limitations and assumptions

POIROT is designed to detect attacks that exploit bugs in a web application's code. Consequently, POIROT assumes that adversaries do not subvert the language interpreter, the web server, or the OS kernel. An adversary that violates this assumption would be able to alter POIROT's logs to hide the attack.

POIROT assumes that the vulnerability being audited is correctly fixed by the security patch used for auditing. Under this assumption, POIROT incurs no false negatives. However, POIROT can incur false positives because it treats any change in application output as an indication of a possible attack. For example, if POIROT failed to record some non-determinism during a request's original execution, re-executing the request could change the request's output and cause POIROT to flag it, even if the request did not exploit the patched vulnerability.

POIROT works best with patches that do not change program behavior aside from fixing a security vulnerability. Patches that both fix security bugs and introduce new features, or that significantly modify the application in order to fix a vulnerability, could generate false positives. For example, if a patch issues a new database query, POIROT flags every request executing the patched code as a possible attack. Extending POIROT to snapshot the database state during original execution, and restore it during request re-execution (as in Warp [9]) would likely prevent these false positives.

POIROT's design focuses on high performance auditing. Once POIROT flags a request as suspicious, an administrator familiar with the application must manually inspect that request to determine the appropriate course of action. POIROT can be combined with a system like Warp [9] to undo the effects of suspicious requests. The problem of helping administrators understand the impact of a suspicious request is left to future work.

POIROT's prototype is built for PHP; PHP's single-threaded nature and its higher-level primitives (e.g., string operations) simplified the prototype's implementation. We believe it is straightforward to extend POIROT to other scripting languages such as Python and Ruby; however, extending POIROT to low-level bytecode such as x86 poses some challenges, which we discuss in §8.

POIROT's prototype assumes the application is never upgraded. We discuss this limitation further in §8.

## 3 Control flow filtering

POIROT's control flow filtering involves three steps. First, during normal execution, POIROT logs a control flow trace of each request to a log file. Second, during indexing, POIROT computes the set of basic blocks executed by each request. Third, when presented with a patch to audit, POIROT computes the set of basic blocks affected by that patch, and filters out requests that did not execute any of the affected basic blocks, since they could not have possibly exploited the vulnerability in the affected basic blocks. As an optimization, POIROT builds an index that maps basic blocks to the set of requests that executed that basic block, which helps speed up the process of locating all requests affected by a patch.

POIROT performs control flow filtering at the granularity of basic blocks because filtering at a coarser granularity (e.g., at function granularity) can result in fewer requests being filtered out, reducing the effectiveness of filtering. Furthermore, control flow traces at the granularity of basic blocks are also needed for memoized re-execution (§5).

The rest of this section describes POIROT's control flow filtering in more detail.

### 3.1 Recording control flow

In order to implement control flow filtering, POIROT needs to know which application code was executed by each request during original execution. POIROT records the request's *control flow trace*, which is a log of every bytecode instruction that caused a control flow transfer. For example, our prototype implements control flow filtering at the level of instructions in the PHP interpreter (called "oplines"), and our prototype modifies the PHP runtime to record branch instructions, function calls, and returns from function calls. For each instruction that caused a control flow transfer, POIROT records the instruction's opcode, the address of that instruction, and the address of the jump target.

Recording control flow traces across multiple requests requires a persistent way of referring to bytecode instructions. PHP translates application source code to bytecode instructions at runtime, and does not provide a standard way of naming the instructions. In order to refer to specific instructions in the application, POIROT names each instruction using a $\langle func, count \rangle$ tuple, where *func* identifies the function containing the instruction, and *count* is the position of the instruction from the start of the translated function (in terms of the number of bytecode instructions). Functions, in turn, are named as $\langle filename, classname, funcname \rangle$.

### 3.2 Determining the executed basic blocks

During the indexing phase, POIROT uses the log recorded above to reconstruct the set of basic blocks executed by each request. To reduce overhead during normal execution, POIROT does not log branches that were not taken. As a result, two adjacent control flow transfers in the log may span $n$ basic blocks, where the branches at the end of the first $n-1$ basic blocks were not taken.

To compute the set of basic blocks executed by a given request, POIROT first computes the sequence of basic blocks within each function, by translating the application's source code into bytecode instructions and analyzing the control flow graph in that function. Then, for each pair of adjacent control flow transfers $A$ and $B$ in the request's log, POIROT adds the sequence of basic blocks between the jump target of $A$'s instruction and the address of $B$'s instruction to the set of basic blocks executed by that request. To consistently name basic blocks across requests, POIROT refers to basic blocks by the first instruction of that basic block.

### 3.3 Determining the patched basic blocks

Once the administrator provides a patch to POIROT in the auditing phase, POIROT must determine the set of requests to re-execute. To filter out requests that were not affected by a given patch, POIROT must determine which basic blocks are affected by a change to the application's source code, and which basic blocks are unchanged. In general, deciding program equivalence is a hard problem. POIROT simplifies the problem in two ways. First, POIROT determines which functions were modified by a patch. Second, POIROT generates control flow graphs for the modified functions,[1] with and without the patch, and compares the basic blocks in the control flow graph starting from the function entry point. If the basic blocks differ, POIROT flags the basic block from the unpatched code as "affected." If the basic blocks are the same, POIROT marks the basic block from the unpatched code as "unchanged," and recursively compares any successor basic blocks, avoiding loops in the control flow graph.

### 3.4 Indexing

To avoid re-computing the set of basic blocks executed by each request across multiple audit operations, and to reduce the user latency for auditing, POIROT caches this information in an index for efficient lookup. POIROT's index contains a mapping from basic blocks (named by the first bytecode instruction in the basic block) to the set of requests that executed that basic block. By using the index, POIROT can perform control flow filtering by computing just the set of basic blocks affected by a patch, and looking up these basic blocks in the index.

[1] PHP has no computed jumps within a function, making it possible to statically construct control flow graphs for a function.

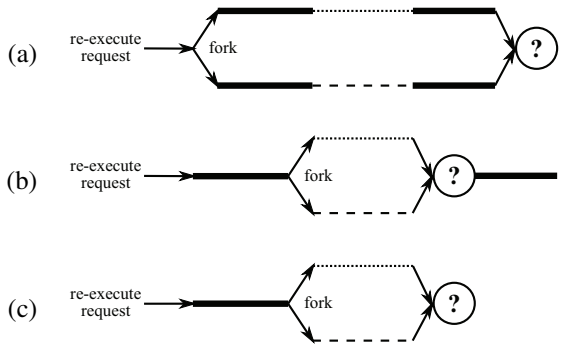



**Figure 2**: Three refinements of request re-execution: (a) naïve, (b) function-level auditing, and (c) early termination. Thick lines indicate execution of unmodified application code, dotted lines indicate execution of the original code for patched functions, and dashed lines indicate execution of new code for patched functions. A question mark indicates comparison of executions for auditing.

The index is generated asynchronously, after the control flow trace for a request has been logged, to avoid increasing request processing latency. The index is shared by all subsequent audit operations. In principle, the index (and the recorded control flow traces for past requests) may need to be updated to reflect new execution paths taken by patched code, after each patch is applied in turn, if the administrator wants to audit the cumulative effect of executing all of the applied patches. Our current prototype does not update the control flow traces for past requests after auditing.

## 4 Function-level auditing

After POIROT's auditing phase uses control flow filtering to compute the set of requests affected by the patch, it re-executes each of those requests twice—once with and once without the patch applied—in order to compare their outputs. A naïve approach of this technique is shown in Figure 2(a). However, the only code that differs between the two executions comes from the patched functions; the rest of the code invoked by the two executions is the same. For example, suppose an application developer patched a bug in an access control function that is invoked by a particular request. All the code executed by that request before the access control function will be the same both with and without the patch applied. Moreover, if the patched function returns the same result and has the same side-effects as the unpatched function, then all the code executed after the function is also going to be the same both with and without the patch.

To avoid executing the common code twice, POIROT implements function-level auditing, as illustrated in Figure 2(b). Function-level auditing starts executing each request in a single process. Whenever the application code invokes a function that was modified in the patch, POIROT forks the process, and invokes the patched function in one process and the unpatched function in the

other process. Once the functions return in both processes, POIROT terminates the child fork, and compares the results and side-effects of executing the function in the two forks, as we describe in §4.1. If the results and side-effects are identical, POIROT continues executing common application code. Otherwise, POIROT flags the request as suspect, since the request's execution may have been affected by the patch.

Comparing the results of each patched function invocation, as in POIROT's function-level auditing, can lead to more false positives than comparing the output of the entire application. This is because the application may produce the same output even if a patched function produces a different return value or has different side-effects with or without the patch. For example, some request may have invoked a patched function, and obtained a different return value from the patched function, but this return value did not affect the eventual HTTP response. These extra false positives can be eliminated by doing full re-execution on the suspect list, and comparing application-level responses, after the faster forked re-execution filters out the benign requests. Our PHP prototype does not implement this additional step, as none of our experiments observed such false positives.

### 4.1 Comparing results and side-effects

A key challenge for function-level auditing is to compare the results and side-effects of invoking an individual function, rather than comparing the final HTTP response of the entire application. To do this, POIROT tracks three kinds of results of a function invocation: HTTP output, calls to external I/O functions (such as invoking an SQL query), and writes to *shared objects*, which are objects not local to the function.

To handle HTTP output, POIROT buffers any output during function execution. When the function returns, POIROT compares the outputs of the two executions.

To handle external I/O functions, POIROT logs the arguments and return values for all external I/O function calls during normal execution. When an external I/O function is invoked during re-execution (in either of the two forks), POIROT checks that the arguments are the same as during the original execution. If so, POIROT supplies the previously recorded return value in response. Otherwise, POIROT declares the request suspect and terminates re-execution.

To handle writes to shared objects, POIROT tracks the set of shared objects that are potentially accessed by the patched function. Initially, the shared object set includes the function's reference arguments and object arguments. The function's eventual return value is also added to the shared object set, unless POIROT determines that the caller ignores the function's return value (by examining the caller's bytecode instructions). To catch accesses to global variables, POIROT intercepts PHP opcodes for accessing a global variable by name, and adds any such object being accessed to the shared object set.

When the function returns, POIROT serializes all objects in the shared object set, and checks that their serialized representations are the same between the two runs. If not, it flags the request as suspect and terminates re-execution. POIROT recursively serializes objects that point to other objects, and records loops between objects, to ensure that it can compare arbitrary data structures.

### 4.2 Early termination

If a patch just modifies a function that executes early in the application, re-executing the rest of the application code after the patched function has already returned is not necessary. To avoid re-executing such code, POIROT implements an *early termination* optimization, as shown in Figure 2(c). Early termination stops re-execution after the last invocation of a patched function returns.

To determine when a request invokes its last patched function, POIROT uses the request's recorded control flow trace to count the number of times each request invoked each function. As an optimization, the indexing phase builds a *function call table* storing these counts.

## 5 Memoized re-execution

Many requests to a web application execute similar code. For example, if two requests access the same Wiki page in Wikipedia, or the same paper in HotCRP, the computations performed by the two requests are likely to be similar. To avoid recomputing the same intermediate results across a group of similar requests, POIROT constructs, at audit time, a *template* that memoizes any intermediate results that are identical across requests in that group. Of course, no two requests are entirely identical: they may differ in some small ways from one another, such as having a different client IP address or a different timestamp in the HTTP headers. To capture the small differences between requests, POIROT's templates have *template variables* which act as template inputs for these differences. POIROT can use a template to quickly re-execute a request by plugging in that request's template variables (i.e., unique parameters) and running the template.

Memoizing identical computations across requests requires addressing two challenges. First, locating identical computations—sequences of identical instructions that process identical data—across requests is a hard problem. Even if two requests invoke the same function with the same arguments, that function may read global variables or shared objects; if these variables or objects differ between the two invocations, the function will perform a different computation, and it would be incorrect to memoize its results. Similarly, a function can have side effects other than its return value. For instance, a function can

modify a global variable or modify an object whose reference was passed as an argument. Memoizing the results of a function requires also memoizing side effects.

Second, POIROT's templates must interleave memoized results of identical computations with re-execution of code that depends on template variables. For example, consider the patch for a simple PHP program shown in Figure 3, and suppose the web server received three requests, shown in Figure 4. The value of `$s` computed on lines 7, 8, and 9 is the same across all three requests, but line 10 generates a different value of `$s` for every request, and thus must be re-executed for each of the three requests. This is complicated by the fact that memoized and non-memoized computations may have control flow dependencies on each other. For instance, what should POIROT do if it also received a request for `/script.php?q=foo`, which does not pass the `if` check on line 5?

POIROT's approach to addressing these two challenges leverages control flow tracing during normal execution. In particular, POIROT builds up templates from groups of requests that had identical control flow traces, even if their inputs differed, such as the three requests shown in Figure 4. By considering requests with identical control flow, POIROT avoids having to locate identical computations in two arbitrary executions. Instead, POIROT's task is reduced to finding instructions that processed the same data in all requests with identical control flow traces, in which case their results can be memoized in the template. Moreover, by grouping requests that share control flow, POIROT simplifies the problem of separating memoized computations from computations that depend on template variables, since there can be no control flow dependencies.

More precisely, POIROT's memoized re-execution first groups requests that have the same control flow trace into a *control flow group*. POIROT then builds up a template for that group of requests, which consists of two parts: first, a sequence of bytecode instructions that produces the same result as the original application, when executing any request from the control flow group, and second, a set of memoized intermediate results that are identical for all requests in the control flow group, used by the instructions in the template. Due to memoization, the number of instructions in a template is often 1–2 orders of magnitude shorter than the entire application (§7).

The rest of this section explains how POIROT generates a template for a group of requests with identical control flow, and how that template is used to efficiently re-execute each request in the group.

## 5.1 Template generation

To generate a template, POIROT needs to locate instructions that processed the same data in all requests, and memoize their results. A naïve approach is to execute every request, and compare the inputs and outputs of ev-

```
1   function name($nm) {
2 -   return $nm;
2 +   return htmlspecialchars($nm);
3   }
4
5   if ($_GET['q'] == 'test') {
6     $nm = ucfirst($_GET['name']);
7     $s  = "Script ";
8     $s .= $_SERVER['SCRIPT_URL'];
9     $s .= " says hello ";
10    $s .= name($nm);
11    echo $s;
12  }
```

**Figure 3**: Patch for an example application, fixing a cross-site scripting vulnerability that can be exploited by invoking this PHP script as `/script.php?q=test&name=<script>..</script>`. The `ucfirst()` function makes the first character of its argument uppercase.

```
1 /script.php?q=test&name=alice
2 /script.php?q=test&name=bob
3 /script.php?q=test&name=<script>..</script>
```

**Figure 4**: URLs of three requests that fall into the same control flow group, based on the code from Figure 3.

| Line | Op | Bytecode instruction | |
|---|---|---|---|
| 5 | 1 | FETCH_R | $0 ← '_GET' |
| 5 | 2 | FETCH_DIM_R | $1 ← $0, 'q' |
| 5 | 3 | IS_EQUAL | ~2 ← $1, 'test' |
| 5 | 4 | JMPZ | ~2 →20 |
| 6 | 5 | FETCH_R | $3 ← '_GET' |
| 6 | 6 ⋆ | FETCH_DIM_R | $4 ← $3, 'name' |
| 6 | 7 ⋆ | SEND_VAR | $4 |
| 6 | 8 ⋆ | DO_FCALL | $5 ← 'ucfirst' |
| 6 | 9 ⋆ | ASSIGN | !0 ← $5 |
| 7 | 10 | ASSIGN | !1 ← 'Script ' |
| 8 | 11 | FETCH_R | $8 ← '_SERVER' |
| 8 | 12 | FETCH_DIM_R | $9 ← $8, 'SCRIPT_URL' |
| 8 | 13 | ASSIGN_CONCAT | !1 ← !1, $9 |
| 9 | 14 | ASSIGN_CONCAT | !1 ← !1, ' says hello ' |
| 10 | 15 ⋆ | SEND_VAR | !0 |
| 10 | 16 ⋆ | DO_FCALL | $12 ← 'name' |
| 10 | 17 | ASSIGN_CONCAT | !1 ← !1, $12 |
| 11 | 18 | ECHO | !1 |
| 12 | 19 | JMP | →20 |
| 13 | 20 | RETURN | 1 |

**Figure 5**: PHP bytecode instructions for lines 5–12 in Figure 3. The line column refers to source lines from Figure 3 and the op column refers to bytecode op numbers, used in control transfer instructions. A ⋆ indicates instructions that are part of a template for the three requests shown in Figure 4 when auditing the patch in Figure 3.

ery instruction to find ones that are common across all requests. However, this defeats the point of memoized re-execution, since it requires re-executing every request.

To efficiently locate common instruction patterns, POIROT performs a taint-based dependency analysis [25], building on the observation that the computations performed by an application for a given request are typically related to the inputs provided by that request. Specifically, POIROT considers the inputs for all of the requests that share a particular control flow trace: each GET and POST parameter, CGI parameters (such as requested URL and the client's IP address), and stored sessions. In PHP, these inputs appear as special variables, called "superglobals", such as `$_GET` and `$_SERVER`. POIROT then determines which of these inputs are common across all requests in the group (and thus computations depending purely on those inputs can be memoized), and which inputs differ in at least one request (and thus cannot be memoized). Inputs in the latter set are called *template variables*. For instance, for the three requests shown in Figure 4, the GET parameter `name` is a template variable, but the GET parameter `q` is not.

To generate the template, POIROT chooses an arbitrary request from the group, and executes it while performing dependency analysis at the level of bytecode instructions; we describe the details of POIROT's dependency tracking mechanism in §5.2. POIROT initially marks all template variable values as "tainted", to help build up the sequence of instructions that depend on the template variables and thus may compute different results for different requests in the group. Any instructions that read tainted inputs are added to the template's instruction sequence, and their outputs are marked tainted as well. If an instruction is added to the template but some of its input operands are not tainted, the current values of those operands are serialized, and the operand in the instruction is replaced with a reference to the serialized object, such as the `$3` operand of instruction 6 in Figure 5. This implements memoization of identical computations. Instructions that have no tainted inputs, as well as any control flow instructions (jumps, calls, and returns), are not added to the template.

For example, consider the PHP bytecode instructions shown in Figure 5. Instructions 1–5 do not read any tainted inputs, and do not get added to the template. Instructions 6–9 depend on the tainted `$_GET['name']` template variable, and are added to the template. Instructions 10–14 again do not read any tainted inputs, and do not get added to the template. Finally, instructions 15 and 16 are tainted, and get added to the template, for a total of 6 template instructions.

When POIROT's template generation encounters an invocation of one of the functions being audited, it marks the start and end of the function invocation in the template, to help audit these function invocations later on, as we will describe in §5.3. If the recorded control flow trace indicates that there will not be any more invocations of patched functions, template generation stops. Going back to Figure 5, template generation stops after instruction 16, because there are no subsequent calls to the patched `name()` function.

## 5.2 Dependency tracking

In order to determine the precise set of instructions that depend on template variables, POIROT performs dependency analysis while generating each template at audit time. In particular, POIROT keeps track of a fine-grained "taint" flag for each distinct memory location in the application. The taint flag indicates whether the current value of that memory location depends on any of the template variables (which are the only memory locations initially marked as tainted). The value of any untainted memory location can be safely memoized, since its value cannot change if the template is used to execute a different request with a different value for one of the template variables. In the PHP runtime, this corresponds to tracking a "taint" flag for every zval, including stack locations, temporary variables, individual elements in an array or object, etc.

POIROT computes the taint status of each bytecode instruction executed during template generation. If any of the instruction's operands is flagged as tainted, the instruction is said to be tainted, and is added to the template. The instruction's taint status is used to set the taint flag of all output operands. For example, instruction 6 in Figure 5 reads a template variable `$_GET['name']`; as a result, it is added to the template and its output $4 is marked tainted. On the other hand, instruction 12 reads `$_SERVER['SCRIPT_URL']`, which is not tainted; as a result, its output $9 is marked as non-tainted.

A template contains only the tainted instructions, which are a subset of the total instructions executed during a request. The output of executing the template instructions for a request is a subset of the output of fully re-executing a request. It is sufficient for POIROT to use the output of template instructions for auditing because the output of non-tainted instructions would be the same in both the patched and unpatched executions.

POIROT's taint tracking code knows the input and output operands for all PHP bytecode instructions. However, PHP also includes several C functions (e.g., string manipulation functions), which appear as a single instruction at the bytecode level (e.g., instruction 8 in Figure 5). To avoid having to know the behavior of each of those functions, POIROT assumes that such functions do not access global variables that are not explicitly passed to them as arguments. Given that assumption, POIROT conservatively estimates that each C function depends on all of its input arguments, and writes to its return value, reference arguments, and object arguments. We encountered one

function that violates our assumption about not affecting global state: the `header()` function used to set HTTP response headers. POIROT special-cases this function.

### 5.3 Template re-execution

Once a template for a control flow group is generated, POIROT uses the template to execute every request in that control flow group. To invoke the template for a particular request, POIROT assigns the template variables (e.g., `$_GET['name']` in Figure 5) with the values from that request, and invokes the template bytecode. In the example of Figure 5, this would involve re-executing instructions 6–9 and 15–16. When the template bytecode comes to an invocation of a patched function (e.g., instruction 16 in Figure 5), POIROT performs function-level auditing, as described in §4, to audit the execution of this function for this particular request. Once the function returns, POIROT compares the results of the function between the two versions (with and without the patch), and assuming no differences appear, POIROT continues executing the template's bytecode instructions.

In principle, it should be possible to use memoized re-execution to reduce the number of bytecode instructions executed inside the patched function as well. We chose a simpler approach, where the entire patched function is re-executed for auditing, mostly to reduce the complexity of our prototype. Most patched functions are short compared to the number of instructions executed in the entire application, allowing us to gain the bulk of the benefit by focusing on instructions outside of the patched functions.

### 5.4 Collapsing control flow groups

The efficiency of memoized re-execution depends on the number of requests that can be aggregated into a single control flow group. Even though the cost of template generation is higher than the cost of re-executing a single request, that cost is offset by the much shorter re-execution time of all other requests in that control flow group.

Building on the early termination optimization from §4.2, we observe that the only part of the control flow trace that matters for grouping is the trace up to the return from the last invocation of a patched function. Instructions executed after that point are not re-executed due to early termination. Thus, two requests whose control flow traces differ only after the last invocation of a patched function can be grouped together for memoized re-execution.

POIROT uses this observation to implement control flow group collapsing. Given a patch, POIROT first locates the last invocation of a patched function in each control flow group, and then coalesces control flow groups that share the same control flow prefix up to the last invocation of a patched function in each trace. This optimization generates larger control flow groups, and thus amortizes the cost of template generation over a larger number of similar requests.

| Component | Lines of code |
|---|---|
| PHP runtime logger / replayer | 9,400 lines of C |
| Indexer | 300 lines of Python |
| Audit controller | 1,200 lines of Python |
| Control flow filter tool | 4,800 lines of Python |

**Table 1**: Lines of code for components of the POIROT prototype.

## 6 Implementation

We implemented a prototype of POIROT for PHP. Table 1 shows the lines of code for the different components of our prototype. We modified the PHP language runtime to implement POIROT's logging and re-execution. The rest of the POIROT components are implemented in Python. The indexer and control flow filter tool use the PHP Vulcan Logic Dumper [28] to translate PHP source code into PHP bytecode in an easy-to-process format, and use that to identify executed and patched basic blocks during control flow filtering.

In order to perform efficient re-execution, POIROT assumes that all patched code resides in functions. However, PHP also supports "global code," which does not reside in any function and is executed when a script is loaded. This causes function-level auditing to execute all of the application code twice, since the "patched function", namely, the global code, returns only at the end of the script. This can be avoided by refactoring the patched global code into a new function that's invoked once from the global code. We performed this refactoring manually for one patch when evaluating POIROT.

POIROT's control flow filtering does not support PHP's reflection API. For example, if a patch adds a new function that was looked up during the original execution of a request (and did not get executed because it did not exist), control flow filtering would miss that request, and not re-execute it. Supporting reflection would require logging calls to the reflection API, and re-executing requests that reflected on modified functions or classes. We did not find this necessary for the applications we evaluated.

## 7 Evaluation

Our evaluation aims to support the following hypotheses:

- POIROT incurs low runtime overhead (§7.2).
- POIROT detects exploits of real vulnerabilities with few false positives (§7.3).
- Even for challenging patches that affect every request, POIROT can audit much faster than either naïve re-execution or the closest related system, Warp (§7.4).
- POIROT's techniques are important for performance (§7.5).

| Workload | # CFG | Latency increase | Thruput reduction | Per-request overheads | | |
|---|---|---|---|---|---|---|
| | | | | Log space | Index space | Indexing time |
| Single URL (1k) | 5 | 13.8% | 10.3% | 4.95 KB | 0.06 KB | 12.3 msec |
| Unique URLs (1k) | 238 | 14.9% | 20.4% | 21.32 KB | 1.79 KB | 28.9 msec |
| Wikipedia (10k) | 499 | 14.1% | 16.9% | 6.72 KB | 4.12 KB | 3.5 msec |
| Wikipedia (100k) | 834 | 14.1% | 15.3% | 5.12 KB | 0.23 KB | 0.8 msec |

**Table 2**: POIROT's logging and indexing overhead during normal execution for different workloads. The CFG column shows the number of control flow groups. Storage overheads measure the size of compressed logs and indexes. For comparison with the last column, the average request execution time during normal execution is 120 msec.

Using a realistic MediaWiki workload and a synthetic HotCRP workload, we show that POIROT's auditing performance is 24–133× that of naïve re-execution, and an additional factor of $\sim 5\times$ faster than Warp (due to Warp's overheads compared to naïve). POIROT catches exploits of real vulnerabilities, with only one patch out of 34 in MediaWiki (and none out of four in HotCRP) causing false positives.

## 7.1 Experimental setup

The test applications used for these experiments were MediaWiki [24], a popular Wiki application that also runs the Wikipedia site, and HotCRP, a popular web-based conference management system. All experiments ran on a 3.07 GHz Intel Core i7-950 machine with 12 GB of RAM. Since the POIROT prototype is currently single-threaded (although in principle the design has lots of parallelism), we used only one core in all experiments.

To obtain a realistic workload, we derived our MediaWiki workload from a real Wikipedia trace [31]. That trace is a 10% sample of the 25.6 billion requests to Wikipedia's ∼20 million unique Wiki pages during a four-month period in 2007. As we did not have time to run the entire four-month trace, we downsampled it to 100k requests. To maintain the same distribution of requests in our workload as in the Wikipedia trace, we chose 1k Wikipedia Wiki pages and synthesized a workload of 100k requests to them, with the same Zipf distribution as in the Wikipedia trace. This new workload has an average of 100 requests per Wiki page, which is more challenging for POIROT than the Wikipedia workload (1k requests per Wiki page), since memoized re-execution works better when more requests have identical control flow traces.

As the Wikipedia database is several terabytes in size, we used the database of the smaller Wikimedia Labs site [1] for our experiments, and mapped the URLs of Wikipedia Wiki pages in our workload to the URLs of Wikimedia Labs Wiki pages. Finally, for privacy reasons, the trace we used did not contain user-specific information such as client IP addresses; to simulate requests by multiple users in the workload, we assigned random values for the client IP address and the user-agent HTTP headers.

## 7.2 Normal execution overheads

To illustrate POIROT's overhead during normal execution, we used several workloads; the results are shown in Table 2. The single URL workload has 1k requests to the same URL, the unique URLs workload has one request to each of the 1k unique URLs in the Wikipedia workload, and the Wikipedia 10k and 100k workloads contain 10k and 100k requests respectively, synthesized as above.

The results demonstrate that POIROT's logging increases average request latency by about 14%, reduces the throughput of normal execution by 10–20%, and POIROT logs require 21 KB per request in the worst case, when all URLs are distinct. POIROT's storage overhead drops considerably for workloads with more common requests, because the log size primarily depends on the number of unique control flow groups. We expect that log sizes for the full Wikipedia trace [31] would be even smaller, since it has an order of magnitude more common requests than our 100k workload.

Table 2 additionally reports the time taken by POIROT's indexing, even though it can be executed at a later time on a separate machine. The indexer takes 1–29 msec per request, and the index file size is 0.06–4.12 KB per request. As with normal execution, indexing time and storage requirements drop for workloads with more common requests. This is because most of the indexing overhead lies in indexing control flow traces, and common requests often have identical control flow traces.

## 7.3 Detecting attacks

We evaluated how well POIROT detects exploits of patched vulnerabilities by using previously discovered vulnerabilities in our two applications, MediaWiki and HotCRP. Using MediaWiki helps compare POIROT to Warp, the closest related work, and we used the same five vulnerabilities evaluated by Warp's authors. The real Wikipedia trace [31] did not contain any attack requests for these vulnerabilities, so we constructed exploits for all five vulnerabilities, and added these requests to our 100k workload. Table 3 shows the results of auditing this workload with POIROT. POIROT can detect all the attacks detected by Warp, and has no false positives for four out of the five attacks. For the clickjacking vulnerability, the

| CVE | Description | Detected? | False +ves |
|---|---|---|---|
| 2009-4589 | Stored XSS | ✓ | 0 |
| 2009-0737 | Reflected XSS | ✓ | 0 |
| 2010-1150 | CSRF | ✓ | 0 |
| 2004-2186 | SQL injection | ✓ | 0 |
| 2011-0003 | Clickjacking | ✓ | 100% |

**Table 3**: Detection of exploits and false positives incurred by POIROT for the five MediaWiki vulnerabilities handled by Warp.

| CVE | POIROT | | Naïve | | Warp | |
|---|---|---|---|---|---|---|
| | # Req | Time (s) | # Req | Time (s) | # Req | Time (s) |
| 2011-4360 | 100k | 267 | 100k | 23,900 | 100k | ∼121,000 |
| 2011-0537 | 100k | 269 | 100k | 23,700 | 100k | ∼121,000 |
| 2011-0003 | 100k | 989 | 100k | 25,100 | 100k | ∼121,000 |
| 2007-1055 | 100k | 1,013 | 100k | 24,300 | 100k | ∼121,000 |
| 2007-0894 | 100k | 236 | 100k | 31,500 | 100k | ∼121,000 |
| 12 cases (⋆) | 0 | 0.03–0.11 | 100k | ∼25,000 | 100k | ∼121,000 |
| 17 cases (†) | 0 | 0.02–0.19 | 100k | ∼25,000 | 0 | $\varepsilon$ |

⋆ 2011-1766, 2010-1647, 2011-1765, 2011-1587, 2011-1580, 2011-1578, 2008-5688, 2008-5249, 2011-1579, 2011-0047, 2010-1189, 2008-4408.

† 2011-4361, 2010-2789, 2010-2788, 2010-2787, 2010-1648, 2010-1190, 2010-1150, 2009-4589, 2009-0737, 2008-5687, 2008-5252, 2008-5250, 2008-1318, 2008-0460, 2007-4828, 2007-0788, 2004-2186.

**Table 4**: POIROT's auditing performance with 34 patches for MediaWiki vulnerabilities, compared with the performance of the naïve re-execution scheme and Warp's estimated performance for the same patches (estimated to be 10× the original execution time, based on results from [9]). $\varepsilon$ for Warp indicates the cost of accessing its index, which was not reported in the Warp paper. Naïve results are measured only for the top 5 patches; its performance would be similar for the 29 other patches.

patch adds an extra `X-Frame-Options` HTTP response header. This modifies the output of every request, causing POIROT to flag each request as suspect. Extending POIROT to include the browser (as in Warp) would likely prevent these false positives. Additionally, POIROT incurs no false positives for 29 other patches shown in Table 4.

To show that POIROT can detect information disclosure vulnerabilities in HotCRP, we constructed exploits for four recent vulnerabilities, including the comment disclosure vulnerability mentioned in §1, and interspersed attack requests among a synthetic 200-user workload consisting of user creation, user login, paper submissions, etc. Table 5 shows the results. POIROT is able to detect all four attacks with no false positives.

## 7.4 Auditing performance

To show POIROT's auditing performance, we used POIROT to audit the Wikipedia 100k workload for 34 real MediaWiki security patches, released between 2004 and 2011. We ported each patch to one of three major versions of MediaWiki released during this time period. We ran the workload against the three MediaWiki versions, which took an average of 12,116 seconds (3.4 hours) to execute during normal operation. POIROT's indexing took on average 79 seconds for this workload. We measured the time taken by POIROT to audit all requests for these patches, the time taken by a naïve scheme that simply re-executes every request twice—with and without the patch—and compares the outputs, and the time taken by Warp, based on numbers reported by Chandra et al. [9].

| Patch | D? | F+ | Description |
|---|---|---|---|
| f30eb4e5 | ✓ | 0 | Capability token lets users see restricted comments. |
| 638966eb | ✓ | 0 | Chair can view an anonymous reviewer's identity. |
| 3ff7b049 | ✓ | 0 | Acceptance decisions visible to all PC members. |
| 4fb7ddee | ✓ | 0 | Chair-only comments are exposed through search. |

**Table 5**: POIROT detects information leak vulnerabilities in HotCRP, found between April 2011 and April 2012. We exploited each vulnerability and audited it with patches from HotCRP's git repository (commit hashes for each patch are shown in the "patch" column). "D?" indicates whether POIROT detects the attack, and "F+" counts false positives.

Table 4 shows the results. For the bottom 29 out of 34 patches (85% of the vulnerabilities), POIROT's control flow filtering took less than 0.2 seconds to determine that the patched code was not invoked by the workload requests, thereby completing the audit within that time. This is compared to the more than 6.5 hours needed to audit using the naïve re-execution scheme.

POIROT audits the remaining five challenging patches, which affect code executed by every request, 24–133× faster than naïve re-execution (top 5 rows in Table 4). This means that POIROT can audit 3.4 hours worth of requests in ∼17 minutes in the worst case.

Our estimate of Warp's performance, based on that paper, is shown in the rightmost columns of Table 4. Warp's file-level filtering allows it to statically discard some requests, although it is unable to filter out requests for 12 patches that POIROT's basic-block-level filtering can. Moreover, when Warp re-executes requests, it is an order of magnitude slower than normal execution, which is a total of 2–3 orders of magnitude slower than POIROT for the worst case patches; for our 3.4 hour workload, Warp could take 1.4 days to audit all of the requests for one patch.

## 7.5 Technique effectiveness

Control flow filtering allows POIROT to quickly filter out unaffected requests (in under 0.2 seconds), as illustrated by the bottom 29 patches in Table 4. As vulnerabilities typically occur in rarely exercised code, we expect control flow filtering to be highly effective in practice.

For the five challenging patches where re-execution is necessary, function-level re-execution and early termination speed up re-execution, as shown in Table 6. The "Func-level re-exec" column shows that it is 1.3–3.4× faster than naïve re-execution, and the "early term. ops" column shows that early termination executes a fraction

| CVE | Naïve re-exec (s) | Func-level re-exec (s) | # early term. ops | # collapsed CF groups | Collapse time (s) | Template gen. time (s) | # template ops | Memoized re-exec (s) |
|---|---|---|---|---|---|---|---|---|
| 2011-4360 | 23,900 | 8,480 | 6,437 / ∼200k | 4 / 844 | 31.0 | 2.10 | 289 | 234 |
| 2011-0537 | 23,700 | 18,900 | 4,801 / ∼200k | 1 / 834 | 30.3 | 1.17 | 96 | 238 |
| 2011-0003 | 25,100 | 19,600 | 117,045 / ∼200k | 589 / 834 | 30.5 | 395.00 | 5,427 | 563 |
| 2007-1055 | 24,300 | 7,150 | 5,571 / ∼200k | 2 / 844 | 30.1 | 0.83 | 177 | 982 |
| 2007-0894 | 31,500 | 10,500 | 24,973 / ∼200k | 18 / 844 | 30.4 | 9.90 | 1,085 | 196 |

**Table 6**: Performance of the POIROT replayer in re-executing all the 100k requests of the Wikipedia 100k workload, for the five patches shown here. The workload has a total of 834 or 844 control flow groups, depending on the MediaWiki version to which the patch was ported. POIROT incurs no false positives for four out of the five patches; it has 100% false positives for the patch 2011-0003, which fixes a clickjacking vulnerability. The "naïve re-exec" column shows the time to audit all requests with full re-execution and the "func-level re-exec" column shows the time to audit all requests with function-level re-execution and early termination. The "early term. ops" column shows the average number of PHP instructions executed up to the last patched function call with early termination (§4.2) across all the control flow groups. The "collapsed CF groups" and "collapse time"columns show the number of collapsed control flow groups and the time to perform collapsing of the control flow groups (§5.4), respectively. The "template gen. time", "template ops", and "memoized re-exec" columns show the time taken to generate templates for all the control flow groups in the workload, the average number of PHP instructions in the generated templates, and the time to re-execute the templates for all the requests, respectively.

of the ∼200k total instructions. For the CVE-2011-0003 vulnerability, the patched function is invoked towards the end of the request, making early termination less effective.

Memoized re-execution further reduces re-execution time, as shown in Table 6. In particular, template collapsing reduces the number of distinct templates from 834–844 to 1–589 ("collapsed CF groups" column), thereby reducing the amount of time spent in template generation ("template gen. time" column). Templates reduce the number of PHP opcodes that must be re-executed by 22–50×, compared to early termination, as illustrated by the "template ops" column. For the CVE-2007-1055 vulnerability, memoized re-execution time is high even though it uses a single template (for its one control flow group); this is because the patched function writes to many global variables, making serialization for comparison expensive.

## 8 Discussion

Our prototype currently assumes the application source code is static, but in practice, application source code is upgraded over time. In order to audit past requests that were executed on different versions of the software, the patch being audited must be back-ported to each of those software versions; this is already common practice for large software projects such as MediaWiki. From POIROT's point of view, the indexes generated for each version of the software must be kept separate, and POIROT's control flow filter must separately analyze the basic blocks for each version. Finally, re-execution of a request must use the source code originally used to run that request (plus the backported patch for that version).

Although our prototype targets PHP web applications, POIROT's techniques should be equally applicable to web applications in other scripting languages such as Python and Ruby. However, when used with low-level bytecode, such as x86 server programs, POIROT's techniques may be less effective due to the following reasons. First, for x86 server applications such as Apache, recording all basic blocks for control flow filtering can impose ∼60% overhead during normal execution [26]; it may be possible to reduce this overhead by profiling the application and recording branches only along the uncommon paths. Second, x86 applications can be multi-threaded, and the non-determinism of thread interleaving can reduce the effectiveness of generating templates for memoized re-execution. Since servers such as Apache typically execute each request in a single thread, independent of other requests, it may be possible to record the execution of each request's thread as a separate control flow trace and use that for memoization. Finally, memoized re-execution in x86 applications may be less effective at finding many requests that share the exact same control flow; for example, string operations in assembly often iterate over all characters in a string, whereas the same operations appear as a single opcode in PHP, Python, and Ruby. One way to apply memoized re-execution at a low level would be to treat string operations as primitives.

Our prototype is currently single-threaded and it was evaluated on a single-core machine. However, the design of the POIROT replayer has lots of parallelism, and it is straightforward to extend it to re-execute requests in parallel. This can be used to significantly reduce auditing time, perhaps taking advantage of cloud computing platforms such as Amazon EC2.

## 9 Related work

This paper's key contribution over prior work lies in the techniques for achieving high auditing performance, particularly in efficiently re-executing many requests to audit them for exploits of a security vulnerability. The rest of this section explains the relation between POIROT and prior work in more detail.

POIROT's approach to auditing a system for intrusions is based on comparing the execution of past requests us-

ing two versions of the code: one with a patch applied, and one without. This approach is similar to delta execution [30], Rad [33], Warp's retroactive patching [9], and TACHYON [23]. POIROT's contributions lie in techniques to improve the performance of this approach for web applications: control flow filtering, function-level auditing, early termination, and memoized re-execution. Function-level auditing in particular is similar to delta execution and Rad.

Past intrusion recovery systems explored several approaches to identify initial intrusions. Some relied on the user for identification [7, 11, 14, 16, 19, 20], which is both tedious for the user and is error-prone. Others asked developers to specify vulnerability-specific predicates [17] for *each* discovered vulnerability; this imposes significant extra effort for developers. Finally, Warp [9] and Rad [33] used the actual patch fixing a vulnerability to identify intrusions, relieving the users and developers of the burden of intrusion detection. Similar to Warp and Rad, POIROT also uses the patch to identify intrusions.

Warp's retroactive patching [9] used file-level dependency tracking to determine requests that were affected by a patch and required re-execution. However, in practice, file-level dependencies are too coarse-grained for many patches: for example, Warp re-executes all requests from our Wikipedia trace for about half of the patches (§7). POIROT uses finer-grained basic-block-level filtering, which filters out requests for many more patches. POIROT also requires less intrusive changes than Warp: it does not require any changes to the browser or the database, and does not require re-execution to take place on the production system.

POIROT's memoized re-execution is similar to dynamic slicing [3], which computes the set of instructions that indirectly affected a given variable. Program slicing, and dynamic slicing in particular, was proposed in the context of helping developers debug a single program. POIROT shows that similar techniques can be applied to locate and memoize identical computations across multiple invocations of a program.

POIROT's control flow filtering is similar to the problem of regression test selection [4, 6]: given a set of regression tests and a modification to the program, identifying the regression tests that need to be re-run to test the modified program. POIROT demonstrates that control flow filtering works well for patch-based auditing under a realistic workload, and further introduces additional techniques (function-level auditing and memoized re-execution) which significantly speed up the re-execution of requests beyond static control flow filtering.

Khalek et al. [18] show that eliminating common setup phases of unit tests in Java can speed up test execution, similar to POIROT's function-level auditing. However, Khalek et al. require the programmer to define undo methods for all operations in a unit test, which places a significant burden on the programmer that POIROT avoids.

Memoization has been used to speed up re-execution of an application over slightly different inputs [2, 15, 32]. Though POIROT's techniques can be extended to work for that scenario as well, memoized re-execution in the current design detects identical computations *across* different executions of a program, and separates memoized computations from input-dependent computations, by grouping requests according to their control flow traces.

POIROT's dependency analysis is similar to taint tracking systems [12, 25]. A key distinction is that taint tracking systems are prone to "taint explosion" if taint is propagated on all possible information flow paths, including through control flow. As a result, taint tracking systems often trade off precision for fewer false positives (i.e., needlessly tainted objects). POIROT addresses the problem of taint explosion through control flow by *fixing* the control flow path for a group of requests, thereby avoiding the need to consider control flow dependencies.

Dynamic dataflow analysis [10] and symbolic execution [8] have been used to generate constraints on a program's input that elicit a particular program execution. These techniques are complementary to control flow filtering and could be extended to apply to POIROT's auditing.

## 10 Summary

This paper presented POIROT, a system that can audit past requests in a web application for exploits of a newly patched security vulnerability. POIROT incorporates three techniques—control flow filtering, function-level auditing, and memoized re-execution—to significantly speed up auditing compared to previous systems that audit through re-execution. POIROT is effective at detecting exploits of real vulnerabilities in MediaWiki and HotCRP. POIROT's optimizations allow it to audit challenging patches, which affect every request, 12–51× faster than the original execution time of those requests.

## Acknowledgments

We thank David Terei for pointing us at prior work on self-adjusting computation [2]. We also thank Eddie Kohler, Neha Narula, Alex Pesterev, Jacob Strauss, Keith Winstein, Eugene Wu, the anonymous reviewers, and our shepherd, Mike Dahlin, for helping improve this paper. This research was partially supported by the DARPA CRASH program (#N66001-10-2-4089), by NSF award CNS-1053143, by Quanta, and by Google.

## References

[1] Wikimedia labs database dump. `http://dumps.wikimedia.org/en_labswikimedia/20111228/`, December 2011.

[2] U. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 35th ACM Symposium on*

*Principles of Programming Languages*, San Francisco, CA, January 2008.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, September 1993.

[5] F. E. Allen. Control flow analysis. In *Proceedings of the Symposium on Compiler Optimization*, 1970.

[6] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3):289–321, October 2011.

[7] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 1–14, San Antonio, TX, June 2003.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[9] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, October 2011.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, Boston, MA, December 2002.

[12] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[13] Github. SSH key audit. `https://github.com/settings/ssh/audit`, 2012.

[14] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara. The Taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 163–176, Brighton, UK, October 2005.

[15] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, July 2011.

[16] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *22nd Annual Computer Security Applications Conference (ACSAC)*, pages 257–268, December 2006.

[17] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Brighton, UK, October 2005.

[18] S. A. Khalek and S. Khurshid. Efficiently running test suites using abstract undo operations. *IEEE International Symposium on Software Reliability Engineering*, pages 110–119, 2011.

[19] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, Vancouver, Canada, October 2010.

[20] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, February 2005.

[21] E. Kohler. Hot crap! In *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, April 2008.

[22] E. Kohler. Correct humiliating information flow exposure of comments. `http://www.read.cs.ucla.edu/gitweb?p=hotcrp;a=commit;h=f30eb4e52e91ab230944eebe8f31bf61e9783d3a`, March 2012.

[23] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proceedings of the 21st Usenix Security Symposium*, Bellevue, WA, August 2012.

[24] MediaWiki. MediaWiki. `http://www.mediawiki.org`, 2012.

[25] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[26] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[27] T. Preston-Werner. Public key security vulnerability and mitigation. `https://github.com/blog/1068`, March 2012.

[28] D. Rethans. Vulcan logic dumper. `http://derickrethans.nl/vld.php`, 2009.

[29] The Open Web Application Security Project. OWASP top 10. `http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf`, 2010.

[30] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.

[31] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11): 1830–1845, 2009.

[32] A. Vahdat and T. Anderson. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[33] X. Wang, N. Zeldovich, and M. F. Kaashoek. Retroactive auditing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011. 5 pages.

# Experiences from a Decade of TinyOS Development

http://www.tinyos.net

Philip Levis
Stanford University
pal@cs.stanford.edu

## Abstract

When first written in 2000, TinyOS's users were a handful of academic computer science researchers. A decade later, TinyOS averages 25,000 downloads a year, is in many commercial products, and remains a platform used for a great deal of sensor network, low-power systems, and wireless research.

We focus on how technical and social decisions influenced this success, sometimes in surprising ways. As TinyOS matured, it evolved language extensions to help experts write efficient, robust systems. These extensions revealed insights and novel programming abstractions for embedded software. Using these abstractions, experts could build increasingly complex systems more easily than with other operating systems, making TinyOS the dominant choice.

This success, however, came at a long-term cost. System design decisions that seem good at first can have unforeseen and undesirable implications that play out over the span of years. Today, TinyOS is a stable, self-contained ecosystem that is discouraging to new users. Other systems, such as Arduino and Contiki, by remaining more accessible, have emerged as better solutions for simpler embedded sensing applications.

## 1. INTRODUCTION

Wireless sensor network research is just over a decade old. Starting as a handful of academic institutions studying networks of tiny, low-power wireless sensing devices, it now has numerous academic conferences and journals that serve a large, worldwide research community. Sensor networks have also grown from research projects to commercial systems. Commercial systems today include ad-hoc wireless smart meter networks, home area networks, and industrial monitoring systems. When Cisco talks about an "Internet of Things," it means the coming Internet with millions or billions of tiny networked devices that interact with and sense the physical environment: sensor networks.

TinyOS is an operating system designed for such embedded devices. It emerged from UC Berkeley in 2000 when sensor network research was beginning, starting as a set of Perl scripts that auto-generated #define statements [23]. Since then, it has evolved to use a C dialect called nesC, has gone through four major revisions, supports tens of sensor network platforms, and has approximately 25,000 downloads per year. TinyOS is the dominant software platform used for sensor network research, enabling hundreds of research results. It is used in numerous commercial products, such as Zolertia [3], Cisco's smart grid systems (formerly Arch Rock), and People Power Company [2].

This paper examines how TinyOS evolved over the past decade. TinyOS is interesting for two reasons. First, like projects such as Xen [11, 44] and OpenFlow [16], TinyOS started as an academic research project that transitioned to significant success and impact outside academia. It managed to make this transition while simultaneously remaining a linchpin of the research community. Second, TinyOS differs from these other examples in that it is a successful, principled, and novel operating system for a new class of computing devices.

This paper examines how technical and social decisions encouraged or restricted the growth of TinyOS and therefore its impact on practice, sometimes in unforeseen ways. For example, fine-grained software components allow users to easily customize the OS with small, local changes. As TinyOS was still forming and being used speculatively in a large number of domains, this easy customization was beneficial. But once core OS services solidified, fine grained components became ultimately harmful, as reading a core system requires leafing through many tiny components.

The paper is divided into four parts. Section 2 describes the two basic principles that have driven TinyOS. The first principle is resource use minimization. The costs of scale and low power operation say that TinyOS code should trade off runtime flexibility or generality for smaller code and data, in contrast to many modern "large" software systems. The bug prevention principle, motivated by the tremendous difficulty of debugging embedded systems, says that TinyOS should be

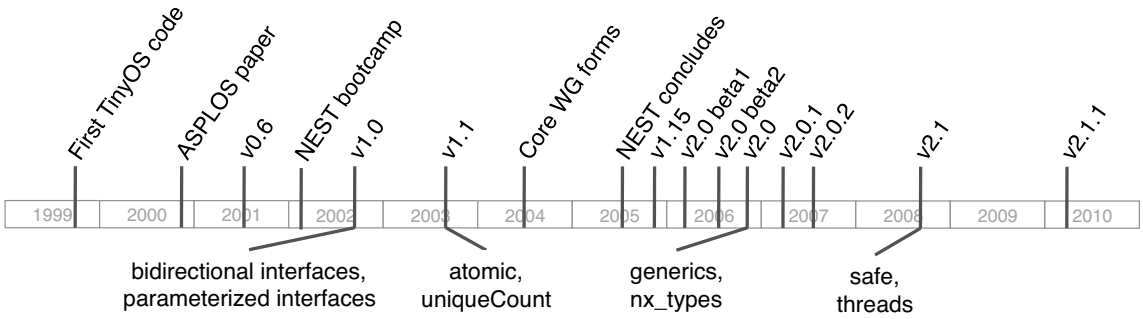



**Figure 1: Timeline of major events in TinyOS development from 1999-2010.**

structured to make it hard to write bugs, sometimes at the cost of making it generally harder to write code. To help support these principles, TinyOS developers chose to design and use nesC [20], a new C dialect. The language and OS co-evolved, such that it does not make sense to talk about one without the other: when we talk about the evolution of TinyOS, we mean the evolution of both the OS and its language.

Sections 3-6 walk through how four approaches TinyOS took had unforeseen long-term implications. The first two, memory allocation and isolation, relate to the unique properties of embedded software. The second two, components and systems language design, relate to systems software more generally. Section 3 discusses how new language features allowed TinyOS to optimally allocate RAM while simultaneously removing the need for some run-time memory access checks. Section 4 describes how a novel software pattern based on this memory allocation, static virtualization, improves software isolation by making the finite state machine of each virtualized instance completely independent. Section 5 examines how using nesC was critically important to TinyOS's early success, but also how its evolution limited TinyOS from even broader, long-term use. Section 6 looks at the benefits and drawbacks of fine-grained, reusable software components, concluding they are a poor fit for operating systems.

Section 7 examines the TinyOS project from a social perspective: how did the project grow such a large developer community? Open source projects live and die based on their contributors. TinyOS today has a large community of developers and users from all across the world. It examines how this community is structured and how that structure evolved. It presents several pitfalls the project encountered, relating to hiring staff, managing code contributions, and the interactions between academia and industry. It also discusses the role of documentation and target audiences and how the project was able to reduce the barrier to entry caused by its increasing technical complexity.

Section 8 takes a step back to examine lessons from TinyOS that can apply to embedded software, systems more generally, and systems projects. One conclusion is that fine-grained components are good for experimentation but add unnecessary and painful complexity to stable software that expects reuse (e.g., a kernel). A second conclusion is the natural tendency to support long-standing, dedicated users and evolve a system to better meet their needs undermines system adoption. Research wants to push a frontier, but doing so can alienate a broader audience and stifle long-term success. We discuss some ways in which future projects seeking large-scale adoption might avoid these and other pitfalls.

| Model | ROM | RAM | Sleep | Price |
|---|---|---|---|---|
| F2002 | 1kB | 128B | 1.3$\mu$A | $0.94 |
| F1232 | 8kB | 256B | 1.6$\mu$A | $2.73 |
| F155 | 16kB | 512B | 2.0$\mu$A | $6.54 |
| F168 | 48kB | 2048B | 2.0$\mu$A | $9.11 |
| F1611 | 48kB | 10240B | 2.0$\mu$A | $12.86 |

(a) TI MSP430 Microcontrollers

| Model | ROM | RAM | Sleep | Price |
|---|---|---|---|---|
| LM2S600 | 32kB | 8kB | 950$\mu$A | $2.73 |
| LM3S1608 | 128kB | 32kB | 950$\mu$A | $4.59 |
| LM3S1968 | 256kB | 64kB | 950$\mu$A | $6.27 |

(b) TI ARM CortexM3 Processors

**Table 1: A representative sampling of popular processors used in low-power wireless sensors. The price values are from DigiKey's catalog on March 3rd, 2010, when purchased in quantities of 1,000 - 10,000.**

## 2. MINIMIZATION AND PREVENTION

TinyOS's design has two major goals: minimizing resource use and preventing bugs. Both are driven by the unique intersection of requirements that sensor networks pose.

The minimization principle states that TinyOS software should use as few hardware resources as possible.

This means being computationally efficient (minimizing cycle counts and wake time), requiring little state (minimizing RAM) and having very tight code (minimizing code ROM). Traditional computing systems want to be efficient, but they typically trade off some efficiency for flexibility and efficiency in the form of kernel modules, plugins, or other mechanisms. In contrast, TinyOS focuses on producing an ultra-optimized binary that can run unattended for months to years.

Two properties of embedded sensors motivate the minimization principle. The first is energy. Within a device class, parts with more hardware resources draw more power both when awake and when asleep. Since nodes sleep almost all of the time, even small sleep power draws are significant. Table 1 shows a selection of recent microcontrollers. 16-bit MSP430 microcontrollers dominate platforms today, due to their 1.3-2$\mu$A sleep draw. An "ultra-low" power 32-bit architecture (ARM Cortex M3), in contrast, has a 950$\mu$A sleep current.

As these devices are already designed for ultra-low power operation, there is no low-hanging fruit which will show large improvements in the short term. Furthermore, microcontrollers do not follow Moore's Law due to market and performance considerations that differ from processors. While the first TinyOS prototypes had 8kB of code and 512 bytes of RAM, 48kB of code and 10kB of RAM has been typical for the past seven years.

Harsh energy concerns ("every bit transmitted brings a sensor node one moment closer to death" [36]) cause nodes to spend almost all of their time asleep. Correspondingly, real-time operating systems, such as FreeRTOS [41], eCos [40], and $\mu$C/OS-II [32], are a poor fit. Their primary purpose is to schedule use of a limited resource (e.g., a CPU) to meet deadlines, but scheduling is easy when the resource is almost always idle. The other benefit of hard real-time is stability in very precise control systems. This stability breaks down in the presence of an unreliable wireless network and so is typically not useful in practice.

Cost is the second motivation for the minimization principle. While research prototypes use top-end microcontrollers for flexibility (e.g., the bottom row of Table 1(a)), for large scale or commercial use they are overkill and raise prices unnecessarily. Using 16kB of code and 512B of RAM instead of the top-end MSP430 could cut unit costs by $6. For 100,000 units, this $600,000 is well worth the cost of a year of software engineer time to optimize and squeeze overly general code.

Over the first four years of TinyOS development, RAM was generally the most limiting resource. The mica [22] and mica2 [4] platforms have 128kB of ROM and 4kB of RAM, and applications typically hit RAM limits before ROM. Unlike a computer with virtual memory and swap, where a slightly-too-big program will run slowly, there is no margin for error on a microcontroller. A too-big program either has a compile error or crashes almost immediately when the stack overruns data memory.

The prevention principle means preventing bugs through software structure. All software wants to prevent bugs, but TinyOS took a very extreme position due to how astonishingly difficult in-the-field debugging of sensor networks is. Debugging is so difficult that it has prompted a wide range of research [13, 37, 42]. A sensor network is a highly distributed system, where nodes dynamically react to the environment and each other. The limited resources, as well as possible energy constraints, on each device preclude extensive logging or other traditional debugging techniques. Many sensor networks do not even support the equivalent of a TCP connection or other per-node access. How does one debug a node's response to an unknown input?

The sensor network research literature has many papers describing application experiences, from volcanoes [43] to bird burrows [38] to HVAC systems and oil tankers [28] to industrial steam pipe monitoring [46]. Application deployments using early versions of TinyOS almost always report a failure that occurred in bringing the system from lab to deployment, yet are unable to pinpoint the cause of the failure [42]. These experiences by users led TinyOS developers to follow the prevention principle more strongly as it matured. Recent deployment papers that use TinyOS 2.x, such as a hospital application in SenSys 2010 [14] are in comparison unabashed success stories.

To meet these goals, TinyOS and nesC evolved language primitives and programming abstractions to push what are traditionally dynamic, run-time operations into static, compile-time ones. Doing so allowed it to have near-optimal RAM overhead while simultaneously enabling large, complex, and dependable software systems. The next sections examine how TinyOS evolved in four ways: ROM and RAM allocation, code isolation, software components, and language features. Figure 1 shows a timeline of the project between 1999 and 2010 that highlights important organizational and technical events.

## 3. RAM AND ROM ALLOCATION

TinyOS programs generally require a 10:1 ratio of ROM:RAM ratio. There are exceptions, such as large packet queues or imaging sensors, but a 10:1 rule of thumb is good for predicting whether RAM or ROM will be the limiting resource. For example, TinyOS 1.x was designed predominantly for the mica platform [22], which had an Atmega atm128 microcontroller with 128kB of ROM and 4kB of RAM. Applications on the mica family typically run into RAM limits before ROM. In contrast, the Telos family [35] uses a Texas Instruments

MSP430 with 48kB of ROM and 10kB of RAM; applications on Telos typically run into ROM limits first.

While minimizing CPU cycles is useful, most resource use minimization efforts focused on RAM and ROM. The nesC paper discusses the major techniques used to minimize ROM (inlining and dead code elimination) [20]. RAM reduction, in contrast, was mostly through software structure. RAM received more attention because mica preceded Telos and so applications fought with RAM limits first.

Some design decisions that traded off increased code size for reduced RAM then posed problems for Telos applications. One example of this tradeoff is how a sensor driver configures a chip's analog-to-digital converter (ADC). Configuration options include which pin to sample, the reference voltage, the sample hold time, and the clock source. Before the driver samples the ADC, it must reconfigure it appropriately. Since reconfiguration is very fast (just twiddling a few control bits in registers), ADC software automatically handles the configuration on every sample. A simple way to set these parameters would be for a sensor driver to allocate a structure in RAM with the correct values, which it passes to the ADC software. But this approach means that each sensor driver allocates a structure even though the ADC needs only one of them at any time. This wastes RAM. Instead, TinyOS sensor drivers implement a function that returns their configuration structure directly on the stack (i.e., not a pointer). Rather than maintain the structure in memory, they regenerate it when needed, reducing RAM needs by 4 bytes per client but increasing ROM by 50-60 bytes. This approach worked well for mica, but "ADC bloat" became a common complaint for Telos applications. RAM-conserving and a ROM-conserving APIs look quite different; forcing developers to choose one or the other has the unwanted side effect of making code less portable.

Minimizing the RAM needed by service APIs, in particular, became exceptionally critical. Where in a traditional OS one wants to make system calls fast, in TinyOS we wanted them to require as little RAM state as possible. Take, as an example, the timer service. Many components and systems need timers. Applications need to periodically collect data, routing protocols need to periodically send beacons, and link layers need to manage backoff intervals as well as retransmissions. A complete application can require anywhere from 3 to 15 timers, and each 32-bit timer requires 10 bytes of state (when it started, its interval, and some control bits, such as whether it's a repeating timer). In the best case, the system will allocate 10 bytes for each timer and no more.

The first version of the timer system (pre-1.0) had clients allocate their timer state and pass a pointer into the timer system. On one hand, this meant that applications allocated precisely the right number of timer structures. On the other, it required additional state in each struct: a pointer so the timer implementation could string them into a linked list. The pointer increased the timer structure to 12 bytes, a 20% overhead. Furthermore, the dynamic data structure became a common source of runtime failures due to memory corruption. As each user of the timer service allocated its own structure, a local off-by-one error could corrupt the pointer, breaking the link list. Recall that there was no debugger. After collecting 30 nodes to reprogram them due to a simple memory bug, you don't ever want to again.

In response to difficult experiences debugging timer problems, the second version of the timer system (v1.0) allocated a fixed array of private timer structures. To distinguish different timers, nesC introduced a special function, unique. The nesC compiler evaluates unique at compile time. Each invocation of unique with a given string $s$ returns a unique integer in the range of 0 to $n-1$, where $n$ is the number of times unique is invoked with $s$. Because there is no binary loading or linking, the nesC compiler parses every call to unique and can compute $n$ correctly. Unique uses the string $s$ as a general way to manage needed sets of unique values. A component that needed a timer allocated a key with unique and passed this key in all calls to the timer system. The second timer implementation used the key to index into its timer structure array.

The second version of the timer system was much more stable, but often wasted even more RAM. Programs made the timer array safely large so calls to unique would not reach past the bounds of the array. This problem was not limited to timers. It existed for ADC sampling, packet queues, and many other components.

The third version of the timer (v1.1) fully minimized RAM through a new nesC function, uniqueCount. Like unique, uniqueCount takes a string and returns an integer. The return value is the number of calls to unique with that string. In the case of timers, for example, the timer service can declare an array of timer state:

```
timer_state_t timers[uniqueCount("Timer")];
```

The unique values can safely access timer state accordingly. Assuming that all timer clients use the correct string (something static virtualization, below, ensures) the timer service can even elide run-time checks that the index parameter is within the size of the array, reducing code size. The final result is that TinyOS today allocates precisely the minimal amount of RAM needed for timers and is 988 bytes of code on mica platforms. If each timer requires 10 bytes of state and there are $n$ timers, it allocates $10n$ bytes of RAM, exactly the minimum required.

## 4. ISOLATION

Initially, TinyOS did not support dynamic memory allocation of any kind. While the need for more flexible memory allocation became increasingly apparent, so did the dangers of a malloc-like approach. TinyOS 1.1 has many cases where multiple components share a single memory resource. For example, the core OS scheduler provides the abstraction of a "task," a form of deferred procedure call. The scheduler maintains a fixed-size array of tasks to execute. If a component posts a task to a full queue, the post fails. This raises a very difficult failure condition: how does the component repost the task? Since TinyOS has a single stack, the component cannot spin or wait, as the scheduler will not free an entry until the current function returns. Instead, the component must somehow be re-invoked, e.g., by starting a timer. But the timer system uses tasks, and it can drop timers when it cannot post its task.

Packet transmission suffered from a similar problem. In TinyOS 1.x, a transmission request fails if the send queue is full. As this queue is shared across many components, it is possible for one component to fill the queue and starve other senders. Some protocols expect periodic transmissions (e.g., routing beacons) and infer their absence as packet losses. Therefore, the calling semantics in TinyOS 1.x caused several deployments with one badly behaving component to have entire protocols collapse.

We concluded that global, shared memory pools, even when hidden and very limited, were too dangerous for robust software and violated the bug prevention principle. One bad component can create hard to handle failures across the entire system. They lead to hard-to-find or unidentifiable bugs, which are excruciatingly frustrating in embedded platforms with no easy debugging interface.

Over time, it became apparent that a lack of isolation in programming interfaces was a major impediment to writing highly reliable TinyOS 1.x software. By isolating processes, a traditional OS greatly simplifies application implementations. The task queue example shows how TinyOS 1.x components, in contrast, had poor isolation. There were numerous other examples of this limitation, such as the link layer send queue, sensors, and generally almost every OS service except timers. Very robust TinyOS 1.x software therefore had to consider that any operation might fail and handle the error, increasing RAM and ROM use.

TinyOS 2.x improves prevention through better component isolation: it makes each component's interactions to an underlying shared resource completely independent. Each client has a perfectly virtualized instance of the underlying service. For example, the return value of a send packet call is independent of whether other packets are in the transmit queue. Memory allocation for this virtualization, however, needs to occur at compile-time, otherwise it would introduce a source of run-time failure.

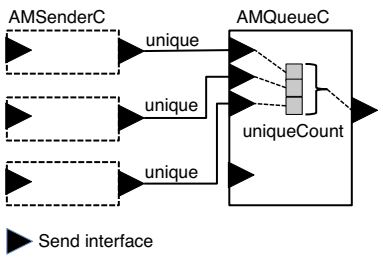



**Figure 2: Static virtualization with AMSenderC.**

TinyOS 2.x achieves this "static virtualization" behavior by combining generic components and the memory allocation techniques described above in Section 3. Generic components are instantiable nesC components, taking types and constant primitive types as parameters (before 2.x, all nesC components were singletons in a global namespace). Generic components improve code reuse just as Java generic, C++ templates and other similar language mechanisms do.

The basic idea behind static virtualization is that a piece of software can declare a logical (virtualized) instance of a service, such as the ability to send a link layer packet. The behavior of an API is completely independent of all other users of the API. A caller can deterministically know the result of any call, as all transitions in the interface's finite state machine come from that client. This differs from deterministic parallelism [9] in that it is concerned with the behavior of only a single API and avoids shared state.

TinyOS accomplishes static virtualization entirely at compile-time. It uses an abstraction called parameterized interfaces to distinguish between multiple clients, the unique and uniqueCount functions to determine exactly how many clients there are, and generic components to prevent bugs by hiding all of this machinery from the user. In TinyOS 2.x, all APIs to core OS services use static virtualization. For example, to send a link layer packet, a program instantiates an AMSenderC component. AMSenderC has the property that it rejects a valid transmission request if and only if that client already has a transmission request outstanding.

Underneath, AMSenderC connects AMSend to a packet queue, shown in Figure 2. The packet queue has a parameterized Send interface. Each instance of AMSenderC connects to it with a call to unique. The queue uses uniqueCount to allocate the correct number of queue entries. When a component tries to send a packet, the queue checks if the corresponding client's entry in the

queue is occupied. If not, it accepts the packet for transmission; if so, it tells the caller to retry.

Static virtualization is an example of a novel programming abstraction from TinyOS that emerged from the unique requirements that wireless sensors face. We believe it represents a large step forward for highly efficient and dependable embedded software. With static virtualization, software can use an OS service, safely isolated from all other users of the service. Because the behavior of the API is based solely on the calling component, one can statically verify that some components are correct (e.g., with interface contracts [7]). Furthermore, the underlying implementation allocates exactly the amount of RAM needed and has simple, concise code.

## 5. LANGUAGE/OS CO-DESIGN

Early on in TinyOS development we made the decision to design a language to better support its programming and concurrency model. The nesC language allowed TinyOS to achieve near-optimal resource efficiency (minimization) and a surprisingly low bug rate (prevention). Having a new language also allowed us to evolve and extend features as new problems arose. For example, the language features for static virtualization (parameterized interfaces, unique, uniqueCount, generic components) emerged over a 4 year period. Being able to control both the language and operating system gave the project tremendous flexibility to achieve system design goals.

On one hand, static virtualization is an excellent programming interface. On the other, the software complexity it takes to achieve in nesC turns out to be formidable. Reaching it took a circuitous path through 4 major releases of TinyOS and five years of development. As a result, static virtualization involves emergent, rather than planned, uses of language mechanisms and a handful of programming idioms which are foreign to a new user.

Language evolution is a two-edged sword. As TinyOS became more robust and users began to tackle more challenging software projects, both the OS and nesC language evolved to meet these needs. On one hand, this evolution made it possible to tackle larger and harder problems. On the other, each stage of this evolution added new features, moving TinyOS and nesC further from C and raising the barrier to entry. Furthermore, the most effective software patterns, such as static virtualization, used all of these features in complex and novel ways. By focusing on expert TinyOS users and making it possible to write larger software, TinyOS 2.x became less accessible to new users. Making it harder to write buggy code had the unfortunate result of making it just plain harder to write code.

In retrospect, the focus on expert users missed a great opportunity: hobbyists and the "Maker" do-it-yourself crowd. The past five years have seen a huge growth in simple, DIY electronic projects, spearheaded by Make Magazine [1]. This community has latched onto the Arduino platform [8] for its projects. In comparison to TinyOS, Arduino is feature-poor: programs are single-threaded C programs for simple sensing and actuation. But for hobbyists, the resulting simplicity is extremely desirable. Building a gumball machine that "only dispenses treats when you knock the secret rhythm on its front panel" (an article in a recent issue of Make magazine) doesn't require static virtualization, network types, and compile-time data race detection.

This increase in learning difficulty had more to do with the novel features of nesC and their increased use than APIs or software implementation. Evolving and larger APIs do not increase programmer difficulty in the same way that language features do. A simple program needs to use a limited number of APIs, and the cognitive effort required scales with program complexity. You see this pattern in language communities, but not operating system ones. For example, consider regular expressions in Perl 5 versus Perl 6. Perl 5 regular expressions are similar to those in many other UNIX tools (sed, shells, etc.), so the learning curve for an experienced UNIX can start very gradually. In contrast, Perl 6 regular expressions introduce programming constructions called grammars and rules that require learning from scratch. While the earliest TinyOS programs were mostly C with a bit of nesC to support components, modern code heavily uses many nesC features, making the learning curve very steep.

The steepness of this learning curve has implications to staffing. Academic projects tend have graduate students as their primary developers. This tension between research and engineering can sometimes be solved by hiring staff software engineers. Language evolution, however, complicated this process considerably for TinyOS. Different groups tried several times to hire TinyOS staff programmers, with mixed results. The first staff hire, made early in TinyOS 1.0, contributed a great deal. But he departed in 2005 to work at a sensor network startup. The second was hired in 2004 during the beginning of TinyOS 2.x development. Intel Research tried hiring a software engineer for 1.x: the hire, after a year, produced a single component which had to be thrown away. The third hire had significant experience in event-driven systems, the gulf between Internet services and TinyOS was too wide and he was unable to contribute. In retrospect, hiring staff early in the project, so they can learn the system as it evolves, was much more successful than doing so late, when it had significant and novel complexity.

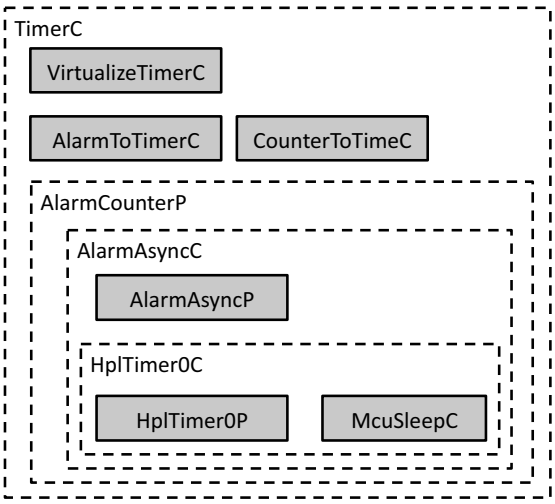



**Figure 3: Component structure for the TinyOS 2.x timer implementation. Grey boxes with solid lines are modules (executable code), while white boxes with dashed lines are configurations, components which connect other components together. AlarmCounterP exists to transform its hardware-independent into specific chip implementation (Atm128AlarmAsyncC).**

One staff member hired late in the project succeeded in contributing because he was an exceptional and unique case. Before starting as a staff engineer, he was one of the largest contributors to TinyOS 2.x, having researched sensor networks while pursuing a Masters degree. The fact that only someone well inside the community could be a significant contributor later on in the project further demonstrates the barrier to entry that OS/language co-design can create.

Other sensor network OSes have emerged to fill the voids left by TinyOS's evolution. Contiki [5], for example, is written entirely in C and provides a more traditional operating system model of a core kernel and applications which compile against it. While TinyOS is more efficient and cleaner, starting with Contiki is much easier. Today a significant fraction of sensor network research builds on top of Contiki rather than TinyOS.

## 6. COMPONENTS

The concept of a component is key to TinyOS's programming model. Components separate interface and implementation, provide data privacy, allow code re-use, and provide sufficient linguistic structure for nesC to perform many useful optimizations. For all of these reasons, using components in embedded software is a tremendous improvement over basic C code.

But while components are generally beneficial, they can be used badly. Early on, TinyOS was intended as a research vehicle. We tried to structure software so that it was easy to extend or modify in a small way. Based on user feedback (in particular, early MAC research such as S-MAC [45]), this structure involved many layered compositions of small, lightweight components. For example, if someone wanted to change the MAC timing behavior of the mica platform (carrier sense, backoff), this involves changing one component. Changing the data encoding/decoding involves changing a different component.

The main goal of this approach was to ease experimentation. But taken to its conclusion, fine-grained components have significant drawbacks we did not foresee. Today, the most heavily used radio driver (for the ChipCon CC2420 [15]) is ≈2400 lines of code[1] and 41 different components. The driver consists of 40 files for 2400 lines of code! In a slightly less extreme example, the timer service, shown in Figure 3, involves 8 components that convert a 32kHz counter with compare and overflow interrupts into a millisecond granularity timer component, which becomes the basis for the statically virtualized timer abstraction (another 3 files). What is ultimately less than a kilobyte of code is spread across 11 different files. In terms of prevention and minimization, this is fine. Each small component is easy to verify and debug, and the interfaces between components are designed to avoid the waste of multiple private copies of the same state.

But as Figure 3 suggests, the drawback of fine-grained components emerges when trying to understand a system for the first time. There are so many tiny pieces of functionality spread across files, with numerous levels of indirection, that keeping track of it all can be a headache. The structural complexity is far beyond what the underlying code complexity requires. In the case of the CC2420, one literally has to have 41 different files open at once to see all the code for just one (admittedly very important) driver. When you are implementing the system, all of it makes sense; but to a new user, it's convoluted and complex. A user interface researcher might say this is not a fundamental problem: a good development tool could make browsing this code easy and intuitive. However, we had neither such a tool nor the expertise to build one. While perhaps not a fundamental problem, it is a real and practical one.

For application-level systems, such as GUI toolkits or the Click modular router [27], fine grained components can make sense. Every application is different, and a very flexible toolkit can greatly speed development. But the tradeoffs for an operating system are very different. In the end, there are very few microcontrollers with

[1]We measure lines of code as the number of lines in a file outside of comments that have a semicolon in them.

which one builds a timer system for using the TinyOS timer library, not that many radios which resemble the CC2420 and not that many variations in its use. These libraries are intended to be the basic APIs of an OS; ultimately, application developers want stability, and so there is very little innovation.

Designing generalized fine-grained abstractions can be valuable if you need to integrate multiple, independent changes. For example, one might want to incorporate an alternative MAC protocol (e.g., Funneling MAC [6]) with an alternative packet retransmission scheme (e.g., Partial Packet Recovery [24]). In practice, however, operating system changes are rarely simply localized and rarely compose easily. While implemented as many small components, those components, for sake of code simplicity, end up being tightly coupled.

Our conclusion is that a well designed and carefully implemented operating system is more helpful than an operating system toolkit or operating system software designed with reuse in mind. Our experience with developing more traditional operating systems supports this conclusion. It is easier to take the Linux boot code and modify it for your needs than to work within a component framework for its generalized boot module. We lost sight of the fact that "code reuse" really means within a system, not necessarily across completely independent systems. As both researchers and software engineers, we want to design generalized abstractions, but an excellent artifact is often more useful than a general architecture.

# 7. COMMUNITY STRUCTURE

TinyOS began as a small research project at UC Berkeley and today has a large, global developer community. Linux's success over HURD in the early 1990s demonstrated that the ability for an open source project to build and maintain an active developer community is as much a result of social interactions and structure as technical concerns.

This section describes how the TinyOS community has evolved socially, focusing on three major considerations: the structure of the community, the relationship between academic and industrial developers, and the effort needed to manage and support users. The prior section described how TinyOS's technical evolution increased its barrier to entry, and this section explores how social mechanisms adopted very late in the project (2007) helped counteract this somewhat.

## 7.1 Historical Progression

The TinyOS community has gone through two major structural changes, reflecting its major revisions: pre-1.0 from 1999-2002, 1.x from 2002-2005 and 2.x from 2005 to present. We present a very brief overview of these changes as background for later observations and also to acknowledge major contributors.

### *7.1.1 Pre-1.0*

Before version 1.0, TinyOS was a small research project at UC Berkeley [23]. All of the major authors were UC Berkeley students, with some students visiting from UCLA and USC contributing a few components, such as for flooding experiments [19]. At this point in time, there was no real separation between TinyOS development and sensor network research. Research meetings at UC Berkeley discussed major design decisions, and close proximity made social interactions about code similar to most research group codebases.

### *7.1.2 Building a community: v1.x*

When version 1.0 was released, TinyOS had a small community of research users through the DARPA NEST project. These users began to contribute code. In addition to students at UC Berkeley, the TinyOS core system[2] developers included researchers and a staff programmer at Intel Labs Berkeley. The Berkeley NEST project group hired a staff member to organize demonstrations, who began to contribute code.

The TinyOS 1.x core system had 37 developers who checked code into the tree. 23 of the developers were from Berkeley: 16 graduate students, 5 undergraduates and two staff members. 6 were from Intel Research Berkeley, 3 were from Technische Universität Berlin, 2 were from Crossbow, Inc., the company that produced the Berkeley hardware designs, and the last 3 were graduate students from Vanderbilt, UCLA, and Harvard.

Although TinyOS 1.x had many users building systems they sought the community to use, most of the core TinyOS development continued to occur at Berkeley. Code in the main TinyOS tree had to go through regression tests for each release. For most research projects, the responsibility of managing formal releases and performing regular tests on someone else's schedule was much more effort than it was worth. Instead, the research community put code in a separate "contributions" directory. While the core TinyOS 1.x tree had 37 contributors, the contrib directory has 110, spread across over 80 project subdirectories, from Funneling MAC [6] to the Capsule flash storage system [31].

### *7.1.3 Expanding globally: v2.x*

The tight collaboration between Berkeley and TU Berlin was the seed for the core TinyOS development community to expand beyond UC Berkeley. This step forward was auspicious: three of the largest TinyOS contribu-

[2]By "core system" we mean the TinyOS code packaged in a release (the `tos/` directory), not PC-side support tools or other non-nesC code.

tors left Berkeley in the spring of 2005. Two started to work full time at their startup company, Moteiv, while one took a faculty position. Setting up a more formal structure would allow all of them to continue to contribute.

A group of the core developers agreed that TinyOS 1.x had numerous unsolvable structural flaws, mostly relating to reliability (e.g., packet transmission queuing as described in Section 4). They formed the TinyOS 2.x working group in October of 2004. The working group realized that one of the major challenges for new users to TinyOS was its lack of any formal design documents. Because every abstraction in TinyOS was up for review and redesign, small subgroups formed and began to define the new interfaces, documenting them in TinyOS Enhancement Proposals (TEPs), a cross between a Python Enhancement Proposal (PEP) and an Internet Request for Comments (RFC).

The first full release of TinyOS 2.0 took two years. When work started, three companies (Moteiv, Arch Rock, and Crossbow) were all significantly involved in the effort and contributed code. By the time 2.0 was released, however, both Crossbow and Moteiv had dropped out of participation. Arch Rock continued to contribute late into 2007.

A small number of institutions dominate academic contributions. After Berkeley-based development transitioned into Arch Rock, Stanford, and Moteiv, Berkeley development dropped to zero until late 2008 and early 2009. The most consistent and significant academic contributor over time is TU Berlin, which not only wrote many parts of the core OS in 2005-6 but also continued development on extensions (e.g., an 802.15.4 MAC in 2008-10). In 2008, Johns Hopkins contributed a network protocol for reprogramming as well as CC2420 security extensions.

One notable aspect of the commit logs is that they are very bursty. Commits generally relate to a library or contribution, and there are very few background commits, e.g., for fixing bugs. The small number of bugs indicates that TinyOS 2.x has successfully followed the prevention principle. From when the TinyOS 2.x tree moved to Google code in July of 2010 until May 2011, there were 16 bug reports over the approximately 80,000 lines of code of the core system.

## 7.2 Industry vs. Academia

TinyOS represents a unique point in the design space of open source projects because it deals with embedded systems yet sees very heavy use by the research community. Because debugging embedded code is very difficult, users have a very strong incentive to use existing code rather than write their own: writing a new device driver is a much more daunting task than writing a new protocol. The research community, however, wants to explore ideas on how to improve important systems and so modify existing codes. These conflicting desires between efficient existing codes to use and extensible codes to conduct research has been a continual tension in the TinyOS development community.

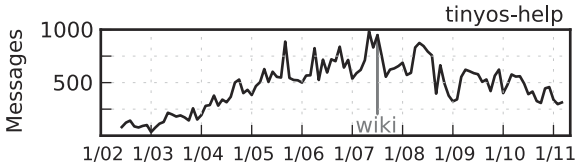



**Figure 4: Traffic on tinyos-help mailing list.**

TinyOS code development has always been primarily academic. Industrial contributions, however, constitute some of the most critical components of the system. For example, the link layer stack for the ChipCon CC2420 radio, the dominant radio used today, has gone through three iterations. The first was an academic rewrite of the TinyOS 1.x stack; the second was a clean reimplementation by Arch Rock; the third was from Rincon Research and included a low-power mode. While the CC2420 code is only 2,400 lines (3% of the codebase), it is one of the most heavily used, experimented with, and important pieces of code.

Initially, developers who left Berkeley for Moteiv and Arch Rock continued to contribute to TinyOS 2.x, as the companies had expressed commitment to an open-source platform. However, the very different timescales of startups and academia proved to be an irreconcilable tension. Both Moteiv and Arch Rock wanted to settle on a "good enough" platform quickly so they could move on to higher-level services they could sell. The academics, in contrast, saw 2.x as the opportunity to "do things right" and establish a design which would minimize future maintenance. The numerous iterations on the design of very low-level systems, such as power management and locking [26], led both Moteiv and Arch Rock to fork the TinyOS tree and use their own private versions of the codebase.

This forking introduced difficult conflicts of interest. For example, Moteiv released Boomerang, a hybrid version of TinyOS halfway between 1.x and 2.x which supported features in newer Moteiv hardware. Meanwhile, members of Moteiv remained involved in TinyOS 2.x discussions. On one hand, they argued that TinyOS 2.x was going in directions contrary to the needs of their customers. On the other, Moteiv had stopped contributing code towards this end, and changing course to follow these suggestions would slow TinyOS 2.x development and cause more people to use Boomerang.

## 7.3 Managing and Supporting Users

Today, TinyOS averages approximately 50-100 downloads/day, or 18,000-36,000 per year (the numbers spike on a release). This number does not include developer downloads via CVS, SVN, or git: it is solely downloads of RPMs, debian packages, and VMWare images. We obtained this count by examining web logs for downloads of these three formats from the TinyOS distribution server, pruning by agent to remove search bots, and then counting the number of unique IP addresses. Unfortunately the bi-monthly rotation of web logs (as well as a server replacement in 2010) prevents us from giving detailed download statistics over time.

Managing a user base so large is difficult, especially because every developer is a volunteer. Computer science graduate students have very little motivation to actively support users. Support is also especially challenging due to the fact that TinyOS is used in many university courses, whose students represent a huge variety of technical ability. Developers typically do not mind answering "interesting" questions or responding to bug reports, but questions on more mundane issues such as Java classpath problems, general C programming questions such as "is there array for TinyOS," or "where do I download TinyOS" become wearying after a few months, let alone a decade.

Tinyos-help, the main help mailing list, started in May 2002. Figure 4 shows the posts per month between then and now. There are two interesting trends: first is the annual dip in traffic around the new year, due to the winter holidays. The second is that messages on the mailing list peaked in June of 2007 with 947 messages that month. Since that time, there has been a steady, downward trend. This downward trend does not correlate with a downward trend in downloads.

What happened in 2007 that made mailing list support easier? In July 2007 the Documentation Working Group formed to move TinyOS from a set of static documentation web pages to a documentation wiki that anyone could modify and improve. Over time, documentation since then has continually improved. Anecdotally, our experience with tinyos-help is that the reduction in traffic since then has not been uniform across types of questions. Traffic on -help has become bimodal, either consisting of the most rudimentary questions by posters who have not bothered to look at the documentation (or search the web), or detailed technical questions. Developers ignore the former and typically respond to the latter. The response to a common, recurring question is typically a pointer to the site-specific Google search for the web accessible tinyos-help archives.

While the process of writing tutorials, API reference documents, and programming manuals is neither glamorous nor exciting, the presence of these materials reduced the long-term effort needed to support a large user community.

# 8. LESSONS LEARNED

In the past decade TinyOS has transitioned from C preprocessor macros maintained by a graduate student at Berkeley to 80,000 lines of code written in a new C dialect by a worldwide community of academic and industrial developers. Arriving at this point involved some good steps and some mistakes. This section tries to answer the question: if we could do it all over with hindsight, what would we do similarly and what would we do differently? Or more generally, how would we recommend growing a systems software research project to be adopted outside academia? We focus on 5 specific decisions TinyOS made: adopting nesC, its focus on software components, its reliance on a research community as users, its collaboration with industry, and how it developed documentation. When possible, we draw parallels between a few other academic software projects.

## 8.1 Good: Language Extensions

Ultimately, the decision to go with nesC was the right one: nesC's language features allowed developers to write robust code that used very few hardware resources. Had we stayed with C, it seems unlikely that TinyOS-based sensor networks would be as advanced as they are today. Chances are another project, realizing the limitations of C, would have tried an alternative language approach. Furthermore, nesC gave us the flexibility to discover novel programming abstractions that are not possible in C and greatly improve system development, such as static virtualization.

## 8.2 Bad: How Language Extensions Evolved

While the decision to use nesC was a good one, how TinyOS used it should have evolved differently. On one hand, eating your own proverbial dog food is important: TinyOS developers built applications and systems, giving them experience with the strengths and weaknesses of the system. On the other hand, doing so led to a distorted perception of what was hard or important. Chasing hard, unsolved problems makes sense from a research standpoint. But from a practical standpoint, making it easier to solve hard problems can simultaneously make it harder to solve the easy ones, and this happened with TinyOS.

In retrospect, it would have been better to split the system design and evolution efforts into two halves. The first half would be to make it easier to build larger and more complex systems. The second half would make it easier to build trivial systems. Motivating systems and networking graduate students to take this second approach would most likely fail. But, for example, sup-

pose TinyOS developers had engaged with work at Stanford [21] or Carnegie Mellon [10] on rapid sensor device prototyping. It could have possibly enabled whole new application domains and use for low-power wireless sensing devices. Arduinos, which have moved to fill this capacity, have very limited network capabilities: who knows what interesting new scientific experiments, art pieces, or toys could have appeared if TinyOS were used instead?

On the other hand, nesC's evolution discovered new and better ways to write efficient, bug-free embedded code. Going forward, the right thing to do is to completely redesign the language, or design a new one, to make these concepts basic language structures rather than complex uses of more general features. For example, one could have a way to define a static virtualization that automatically sets up parameterization, unique, and state management: a single file could define the service, rather than the typical case today of at least 4 files.

## 8.3 Good: Software Components

Components are a significant improvement over basic C code. They provided clean, reusable interfaces, data privacy, and enabled many tools for checking and verifying TinyOS code. They encourage clean system decompositions, which enabled small groups of programmers to build intricate, complex systems, ranging from shooter localization applications [30] to the Tenet programming system [33].

## 8.4 Bad: Software Component Architectures

Faced with such capabilities, however, the inevitable academic tendency is to generalize and define architectures for core services, such as TinyOS' network layer architecture (NLA) [18]. But these generalizations, in practice, turn out to usually be much more effort than they are worth. If there is only a small handful of implementations for any given abstraction (e.g., forwarding policy, link estimator), the structural complexity that generalization adds is detrimental. "Don't generalize; generalizations are generally wrong." [29] In practice, clean, easy-to-understand code without too much structural complexity can be easier to copy and modify. We should have started with fine-grained components, then over time transitioned to more monolithic implementations as they stabilized.

## 8.5 Good: Initial Users

Without the NEST project, TinyOS would not have moved far past Berkeley and a few other schools. NEST motivated Berkeley developers to embark on sometimes boring software engineering projects: others would use their code and so the work had impact. It also led to software from outside Berkeley that others could use, extend, and compare against. Finally it created momentum and interest in the form of social memory and knowledge. When a researcher thinks about using TinyOS, chances are they know another person or group who is already doing so, whom they can learn from.

While it's obvious that getting an initial group of users is critical, how does one do so? There are two basic mechanisms. The first is to promote use internally, among other groups or researchers who might find the system useful. One notably successful example is the Click modular router [27]. Click, originating at MIT, has been used in many research projects from that institution, such as Roofnet [12] and wireless network coding [25]. These demonstrations of Click's success as not only a research project but also a practical tool have helped it now be used by many researchers and companies.

The second approach, which is generally more easily successful, is to have a funding agency give grants to work on the system. For the NEST project, using TinyOS was essentially a requirement. Of course, this has drawbacks as well. Some NEST participants still resent that they had to use TinyOS. Other examples that followed this approach are less extreme, such as DHash++ [17] as part of IRIS, or PlanetLab [34] and Intel.

## 8.6 Bad: Focusing on Experts

In retrospect, focusing on growth within the research community exacerbated TinyOS' focus on technical complexity. The project, just as with technical directions, should have also focused on broadening participation. But in seeking research impact, the project sought impact predominantly with researchers.

While it is possible to achieve impact by having users outside the research community (X from MIT, BSD from Berkeley, Mach from CMU, Xen from Cambridge, and more recently OpenFlow from Stanford, are notable examples), this is especially difficult for embedded software. Embedded systems are often closed, single-vendor, vertically integrated systems where the vendor gives few if any real details on the underlying technology. Anecdotally, through we know of many companies who use TinyOS in products, only a tiny handful will say so on the record.

## 8.7 Bad: Early Industrial Involvement

When effort on TinyOS 2.x started, several companies were involved in the design process. Each of them, however, dropped out within nine months as their development timescale was much, much faster than academia. Frustrated by the long discussions and numerous design iterations, both Moteiv and Arch Rock forked from the main tree to develop their own branches. The frustration was also due to differing goals: both Arch Rock and

Moteiv wanted to focus on the hardware platform they both used, while academic groups representing multiple hardware devices wanted more generality.

What is especially revealing is that many of the early criticisms from Crossbow and Moteiv on the programmability of TinyOS 2.x were, in retrospect, completely correct. While we believe involving industry in early design was a mistake, TinyOS would have benefited from more carefully listening to the requirements industrial collaborators presented. Instead, early industrial partners departed the project in frustration.

## 8.8 Good: Late Industrial Involvement

Once the core design was complete in early 2006, however, companies such as Rincon Research, Handhelds.org, Zolertia, and Shockfish began to join the project and contribute significantly. This code was typically drivers for their platforms, although it also included a few utility libraries. Given a well defined structure and precise, stable interfaces, commercial engineers were willing to participate and contribute without having to accommodate what must at times seem like philosophical debates about hypothetical universes. The OpenFlow project at Stanford lends additional evidence that incorporating commercial contributors later, not earlier, is a better approach than the one TinyOS tried. The original designs of OpenFlow and Xen originated within Stanford and Cambridge. Over time, the projects enlisted industrial partners who are willing to implement, extend, and use the system.

## 8.9 Good: Diverse Documentation

As a user community grows, documentation is absolutely critical to keeping down the support effort needed. Writing documentation can be time consuming, but is worth the long-term time savings in answering questions. TinyOS ultimately gravitated towards three forms of documentation: tutorials, for getting started, TEPs, which are API and implementation references, and a TinyOS programming manual (over 200 pages) that goes into excruciating detail on advanced programming and software engineering techniques. Tutorials acclimate a new user to how to write a program and use some simple functionality; TEPs explain most of the system functionality, for when a user wants to build something new; the programming manual helps when a user wants to write a reasonably large and complex piece of software.

One sometimes frustrating result of good documentation is that you hear very little from users: no news is good news. After the TinyOS documentation wiki started, some developer wondered whether the slow reduction in questions was due to the TinyOS community slowly fading away. But the number of downloads indicates otherwise: more people are downloading TinyOS, but fewer are asking questions about it.

## 8.10 Bad: Only Developer Documentation

It's challenging for someone to write documentation intended for an audience with a vastly different technical background. When TinyOS was early in its evolution and not yet very complex, documentation written by its developers was reasonably accessible to other C programmers. But as the system become more complex and developer expertise increased, tutorials became simultaneously longer and more obtuse.

In retrospect, TinyOS was far too late in transitioning documentation to a wiki. There was a bit of a control concern: if you open documentation to the masses, they might write something incorrect. But generally, for every mistake, there will be ten additions that are correct. If a user thinks a certain piece of documentation is needed, trust that thought. For example, one of the earliest community documentation contributions, the second link on `http:/docs.tinyos.net`, is a tiny page that demonstrates the simplest TinyOS program. We initially thought that something so minor should be in a tutorial, or deeper in the site, but in retrospect realized we should leave it to users to decide.

However, one cannot simply create a wiki and expect users to populate it with content for free. Developers have to heavily seed the documentation effort. Users, like everyone else, are much more motivated to improve something that's there than to create out of whole cloth.

# 9. CONCLUSION

A decade is a long time, especially for an academic project. TinyOS was able to transition from the academic halls of UC Berkeley into a worldwide community of developers and users. Getting to this point involved tens of thousands of hours of work by hundreds of contributors. In retrospect, some decisions that seemed sound at the time had significant negative long-term implications that we did not foresee. For example, while designing language extensions for better operating systems programming is valuable, co-evolving those extensions with the OS can alienate new users, limiting the long-term benefits of the work.

TinyOS has been a critical enabler for wireless sensor network research and engineering, the benefits of which we see in efforts like the IETF developing standards for connecting low-power wireless sensors to the Internet [39]. As computing increasingly pervades society, the ability for universities to transition research into practical, real-world impact and benefits will remain important and valuable. Our hope is that the lessons we learned so may help others trying to do so in the future.

## Acknowledgments

TinyOS is the collaborative work of many developers, too many to list here, who all deserve credit for its success. I'd like to especially acknowledge Jason Hill, David Gay, Cory Sharp, Joe Polastre, Vlado Handziski, Jan Heinrich-Hauer, Kevin Klues, David Moss, Omprakash Gnawali, Jonathan Hui, John Regehr, Matt Welsh, Alec Woo, Robert Szewczyk, Kamin Whitehouse, Philip Buonadonna, Ben Greenstein, and Miklos Maroti. In addition, David Culler's leadership, Eric Brewer's language design insights, and Shankar Sastry's application knowledge all set the trajectory that led to the operating system's longevity and success. Last but certainly not least, TinyOS would never have succeeded without its users, their bug reports, feature requests, and hard work that helped define the early years of sensor network research.

I'd also like to thank the program committees of SOSP 2011 and OSDI 2012. Their reviews provided excellent advice on what aspects of TinyOS's history could be most beneficial to other researchers and engineers.

This work was supported by generous gifts from Microsoft Research, Intel Research, DoCoMo Capital, Foundation Capital, the National Science Foundation under grants #0615308 ("CSR-EHS"), #0627126 ("NeTS-NOSS"), and #0846014 ("CAREER"), as well as a Stanford Terman Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## 10. REFERENCES

[1] Make magazine. `http://makezine.com`.

[2] People Power Company. `http://www.peoplepowerco.com`.

[3] Zolertia Wireless Sensor Networks. `http://www.zolertia.com`.

[4] Mica2 schematics. `http://webs.cs.berkeley.edu/tos/hardware/design/ORCAD_FILES/MICA2/6310-0306-01ACLEAN.pdf`, Mar. 2003.

[5] Adam Dunkels and Björn Grünvall and and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE EmNetS-I)*, 2004.

[6] G.-S. Ahn, S. G. Hong, E. Miluzzo, A. T. Campbell, and F. Cuomo. Funneling-MAC: a Localized, Sink-Oriented MAC for Boosting Fidelity in Sensor Networks. In *Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.

[7] W. Archer, P. Levis, and J. Regehr. Interface Contracts for TinyOS. In *Proceedings of the Sixth International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[8] Arduino Team. Arduino home page. `http://www.arduino.cc`.

[9] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[10] D. Avrahami and S. E. Hudson. Forming Interactivity: A Tool for Rapid Prototyping of Physical Interactive Products. In *Proceedings of the Fourth Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS)*, 2002.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[12] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and Evaluation of an Unplanned 802.11b Mesh Network. In *Proceedings of the Eleventh Annual International Conference on Mobile Computing and Networking (Mobicom)*, 2005.

[13] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative Tracepoints: a Programmable and Application Independent Debugging System for Wireless Sensor Networks. In *Proceedings of the Sixth International Conference on Embedded Network Sensor Systems (SenSys)*, 2008.

[14] O. Chipara, C. Lu, T. C. Bailey, and G.-C. Roman. Reliable Clinical Monitoring Using Wireless Sensor Networks: Experiences in a Step-Down Hospital Unit. In *Proceedings of the Eighth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[15] ChipCon Inc. CC2420 Data Sheet. `http://www.chipcon.com/files/CC2420_Data_Sheet_1_4.pdf`, 2006.

[16] O. S. Consortium. Openflow. `http://www.openflow.org/`.

[17] F. Dabek. *A Distributed Hash Table*. PhD thesis, Cambridge, MA, USA, 2005.

[18] C. T. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A Modular Network Layer for Sensorsets. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[19] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An Empirical Study of Epidemic Algorithms in Large Scale Multihop Wireless Networks. UCLA Computer Science Technical Report UCLA/CSD-TR 02-0013, 2002.

[20] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[21] B. Hartmann, S. R. Klemmer, M. Bernstein, L. Abdulla, B. Burr, A. Robinson-Mosher, and J. Gee. Reflective Physical Prototyping through Integrated Design, Test, and Analysis. In *Proceedings of the Nineteenth Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2006.

[22] J. Hill and D. E. Culler. Mica: a Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24, 2002.

[23] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[24] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2007.

[25] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. XORs in the Air: Practical Wireless Network Coding. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2006.

[26] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proceedings of Twenty-First ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[28] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra,

M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[29] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP)*, 1983.

[30] M. Maroti, G. Simon, A. Ledeczi, and J. Sztipanovits. Shooter localization in urban terrain. *Computer*, 37(8):60–61, Aug. 2004.

[31] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy. Ultra-Low Power Data Storage for Sensor Networks. *ACM Transactions on Sensor Networks*, 5(4):33:1–33:34, 2009.

[32] Micrium. The uC/OS-II Kernel. `http://micrium.com/page/products/rtos/os-ii`.

[33] J. Paek, B. Greenstein, O. Gnawali, K.-Y. Jang, A. Joki, M. Vieira, J. Hicks, D. Estrin, R. Govindan, and E. Kohler. The Tenet Architecture for Tiered Sensor Networks. *ACM Transactions on Sensor Networks*, 6(4):34:1–34:44, 2010.

[34] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.

[36] G. Pottie. Casting the Wireless Sensor Net. MIT Technology Review, July 2003.

[37] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the Sensor Network Debugger. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[38] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An Analysis of a Large Scale Habitat Monitoring Application. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[39] T. Winter and P. Thubert (editors). RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. Internet Network Working Group RFC6550, March 2012.

[40] The eCos project. eCos v2.0 Embedded Operating System. `http://ecos.sourceware.org`.

[41] The FreeRTOS project. FreeRTOS – Free professional grade RTOS. `http://www.freertos.org`.

[42] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis. Visibility: a New Metric for Protocol Design. In *Proceedings of the Fifth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[43] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[44] Xen.org community. The Xen Hypervisor Project. `http://xen.org`.

[45] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *In Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[46] C. Zhang, A. Syed, Y. H. Cho, and J. Heidemann. Steam-Powered Sensing. In *Proceedings of the Ninth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

# Automated Concurrency-Bug Fixing

Guoliang Jin Wei Zhang Dongdong Deng Ben Liblit Shan Lu
University of Wisconsin–Madison
{aliang,wzh,dongdong,liblit,shanlu}@cs.wisc.edu

## Abstract

Concurrency bugs are widespread in multithreaded programs. Fixing them is time-consuming and error-prone. We present CFix, a system that automates the repair of concurrency bugs. CFix works with a wide variety of concurrency-bug detectors. For each failure-inducing interleaving reported by a bug detector, CFix first determines a combination of mutual-exclusion and order relationships that, once enforced, can prevent the buggy interleaving. CFix then uses static analysis and testing to determine where to insert what synchronization operations to force the desired mutual-exclusion and order relationships, with a best effort to avoid deadlocks and excessive performance losses. CFix also simplifies its own patches by merging fixes for related bugs.

Evaluation using four different types of bug detectors and thirteen real-world concurrency-bug cases shows that CFix can successfully patch these cases without causing deadlocks or excessive performance degradation. Patches automatically generated by CFix are of similar quality to those manually written by developers.

## 1 Introduction

### 1.1 Motivation

Concurrency bugs in multithreaded programs have already caused real-world disasters [27, 46] and are a growing threat to software reliability in the multi-core era. Tools to detect data races [12, 50, 68], atomicity violations [7, 13, 30, 31], order violations [16, 33, 66, 69], and abnormal inter-thread data dependencies [53, 70] have been proposed. However, finding bugs is just a start. Software reliability does not improve until bugs are actually fixed.

Bug fixing is time-consuming [38] and error-prone [54]. Concurrency bugs in particular bring unique challenges, such as understanding synchronization problems, selecting and using the right synchronization primitives in the right way, and maintaining performance and readability while adding synchronization into multi-threaded software. A previous study of open-source software [32] finds that it takes 73 days on average to correctly fix a concurrency bug. A study of operating-system patches [65] shows that among common bug types, concurrency bugs are the most difficult to fix correctly. 39% of patches to concurrency bugs in released operating system code are incorrect, a ratio 4 – 6 times higher than that of memory-bug patches.

Fortunately, concurrency bugs may be more amenable to automated repair than sequential bugs. Most concurrency bugs only cause software to fail rarely and nondeterministically. The correct behavior is already present as some safe subset of all possible executions. Thus, such bugs can be fixed by systematically adding synchronization into software and disabling failure-inducing interleavings.

Prior work on AFix demonstrates that this strategy is feasible [20]. AFix uses static analysis and code transformation to insert locks and fix atomicity violations detected by CTrigger [43]. Although promising, AFix only looks at one type of synchronization primitive (mutex locks) and can fix only one type of concurrency bug (atomicity violations) reported by one specific bug detector (CTrigger). In addition, AFix cannot fix a bug when CTrigger reports bug side effects instead of bug root causes.

### 1.2 Contributions

CFix aims to automate the entire process of fixing a wide variety of concurrency bugs, without introducing new functionality problems, degrading performance excessively, or making patches needlessly complex. Guided by these goals, CFix system automates a developer's typical bug fixing process in five steps, shown in Figure 1.

The first step is *bug understanding*. CFix works with a wide variety of concurrency-bug detectors, such as atomicity-violation detectors, order-violation detectors, data race detectors, and abnormal inter-thread data-dependence detectors. These detectors report failure-inducing interleavings that bootstrap the fixing process.

The second step is *fix-strategy design* (Section 2). CFix designs a set of fix strategies for each type of bug report. Each fix strategy includes mutual-exclusion/order relationships[1] that, once enforced, can disable the failure-inducing interleaving. By decomposing every bug report into mutual-exclusion and order problems, CFix addresses the diversity challenge of concurrency bugs and bug detectors. To extend CFix for a new type of bugs, one only

[1] A mutual-exclusion relationship requires one code region to be mutually exclusive with another code region. An order relationship requires that some operation always execute before some other operation.

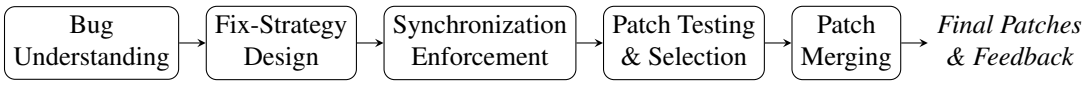



Figure 1: CFix bug fixing process

needs to design new fix strategies and simply reuses other CFix components.

The third step is *synchronization enforcement* (Section 3). Based on the fix strategies provided above, CFix uses static analysis to decide where and how to synchronize program actions using locks and condition variables, and then generates patches using static code transformation. Specifically, CFix uses the existing AFix tool to enforce mutual exclusion, and a new tool OFix to enforce order relationships. To our knowledge, OFix is the first tool that enforces basic order relationships to fix bugs with correctness, performance, and patch simplicity issues all considered. This lets CFix use more synchronization primitives and fix more types of bugs than previous work.

The fourth step is *patch testing and selection* (Section 4). CFix tests patches generated using different fix strategies, and selects the best one considering correctness, performance, and patch simplicity. In this step, CFix addresses the challenge of multi-threaded software testing by leveraging the testing framework of bug detectors and taking advantage of multiple patch candidates, as the testing result of one patch can sometimes imply problems of another. This step also addresses the challenge of bug detectors reporting inaccurate root causes: patches fixing the real root cause are recognizable during testing as having the best correctness and performance.

The fifth step is *patch merging* (Section 5). CFix analyzes and merges related patches. We propose a new merging algorithm for order synchronization operations (i.e., condition-variable signal/wait), and use AFix to merge mutual-exclusion synchronizations (i.e., locks). This step reduces the number of synchronization variables and operations, significantly improving patch simplicity.

Finally, the CFix run-time monitors program execution with negligible overhead and reports deadlocks caused by the patches, if they exist, to guide further patch refinement.

We evaluate CFix using ten software projects, including thirteen different versions of buggy software. Four different concurrency-bug detectors have reported 90 concurrency bugs in total. CFix correctly fixes 88 of these, without introducing new bugs. This corresponds to correctly patching either twelve or all thirteen of the buggy software versions, depending on the bug detectors used. CFix patches have excellent performance: software patched by CFix is at most 1% slower than the original buggy software. Additionally, manual inspection shows that CFix patches are fairly simple, with only a few new synchronization operations added in just the right places.

Overall, this paper makes two major contributions:

Firstly, we design and implement OFix, a tool that enforces two common types of order relationship between two operations identified by call stacks tailored for fixing concurrency bugs. Specifically, OFix focuses on two basic order relationships: either (1) an operation B cannot execute until *all* instances of operation A have executed; or (2) an operation B cannot execute until at least *one* instance of operation A has executed, if operation A executes in this run at all. We refer to these respectively as all*A–B* and first*A–B* relationships. See Sections 3 and 5 for details.

Secondly, we design and implement CFix, a system that assembles a set of bug detecting, synchronization enforcing, and testing techniques to automate the process of concurrency-bug fixing. Our evaluation shows that CFix is effective for a wide variety of concurrency bugs.

## 2 Fix Strategy Design

The focus of CFix is bug fixing; we explicitly do not propose new bug-detection algorithms. Rather, CFix relies on any of several existing detectors to guide bug fixing. We refer to these detectors as CFix's *front end*. We require that the front end provide information about the failure-inducing interleaving (i.e., a specific execution order among bug-related instructions). We do *not* require that the bug detector accurately report bug root causes, which we will demonstrate using real-world examples in Section 6.

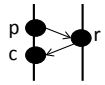
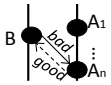
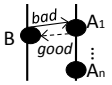
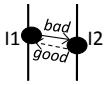
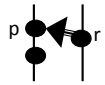
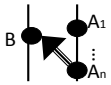
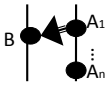
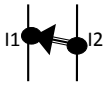
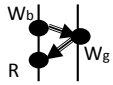
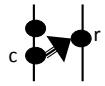
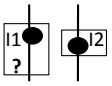
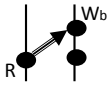
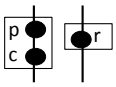
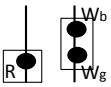
### 2.1 Mutual Exclusion and Ordering

To effectively handle different types of concurrency bugs and bug detectors, CFix decomposes every bug into a combination of mutual-exclusion and order problems. The rationale is that most synchronization primitives either enforce mutual exclusion, such as locks and transactional memories [17, 18, 48], or enforce strict order between two operations, such as condition-variable signals and waits. Lu et al. [32] have shown that atomicity violations and order violations contribute to the root causes of 97% of real-world non-deadlock concurrency bugs.

In this paper, a *mutual-exclusion relationship* refers to the basic relationship as being enforced by AFix [20] among three instructions p, c, and r. Once mutual exclusion is enforced, code between p and c forms a critical section which prevents r from executing at the same time.

An *order relationship* requires that an operation A always execute before another operation B. Note that A and B may each have multiple dynamic instances at run

Table 1: Bug reports and fix strategies. Rectangles denote mutual exclusion regions, wide arrows denote enforced order, and circles illustrate instructions. Vertical lines represent threads 1 and 2. $A_n$ is the $n$th dynamic instance of A.

| | (a) Atomicity Violation | (b) Order Violation all*A–B* | (b) Order Violation first*A–B* | (c) Race | (d) Def-Use Remote-is-Bad | (d) Def-Use Local-is-Bad |
|---|---|---|---|---|---|---|
| Reports: |   |   |   |   |   |   |
| Strategy (1): |   |   |   |   |   |   |
| Strategy (2): |   | N/A | N/A |   |   | N/A |
| Strategy (3): |   | N/A | N/A | N/A |   | N/A |

```
// Thread 1                          // Thread 2
printf("End at %f", Gend); //p       // Gend is uninitialized
...                                  // until here
printf("Take %f", Gend-init); //c    Gend = time(); //r
```

Figure 2: Concurrency bug simplified from FFT. Making Thread 1 mutually exclusive with Thread 2 cannot fix the bug, because r can still execute after p and c.

```
// Thread 1                          // Thread 2
while (...) {                        free(buffer); // B
  tmp = buffer[i]; // A
}
```

Figure 3: Order violation simplified from PBZIP2

time; the desired ordering among these instances could vary in different scenarios. We focus on two basic order relationships: all*A–B* and first*A–B*. When bug fixing needs to enforce an order relationship, unless specifically demanded by the bug report, we try all*A–B* first and move to the less restrictive first*A–B* later if all*A–B* causes deadlocks or timeouts.

### 2.2 Strategies for Atomicity Violations

An atomicity violation occurs when a code region in one thread is unserializably interleaved by accesses from another thread. Many atomicity-violation detectors have been designed [13, 15, 30, 31, 35, 43, 57, 64]. CFix uses CTrigger [43] as an atomicity-violation-detecting front end. Each CTrigger bug report is a triple of instructions $(\mathsf{p},\mathsf{c},\mathsf{r})$ such that software fails almost deterministically when r is executed between p and c, as shown in Table 1(a).

Jin et al. [20] patch each CTrigger bug by making code region p–c mutually exclusive with r. However, this patch may not completely fix a bug: CTrigger may have reported a side effect of a concurrency bug rather than its root cause. Figure 2 shows an example.

Instead of relying on CTrigger to point out the root cause, which is challenging, CFix explores all possible ways to disable the failure-inducing interleaving, as shown in Table 1(a): (1) enforce an order relationship, making r always execute before p; (2) enforce an order relationship, making r always execute after c; or (3) enforce mutual exclusion between p–c and r.

### 2.3 Strategies for Order Violations

An order violation occurs when one operation A should execute before another operation B, but this is not enforced by the program. Order violations contribute to about one third of real-world non-deadlock concurrency bugs [32]. Figures 2 and 3 show two examples simplified from real-world order violations.

Many existing tools detect order violation problems and could work with CFix. This paper uses ConMem [69] as a representative order-violation detector. ConMem discovers buggy interleavings that lead to two types of common order violations: dangling resources and uninitialized reads. For dangling resources, ConMem identifies a resource-use operation A and a resource-destroy operation B, such as those in Figure 3. The original software fails to enforce that all uses of a resource precede all destructions of the

same resource. For uninitialized reads, ConMem finds a read operation B and a write operation A. The original software fails to enforce that at least one instance of A occur before B, leading to uninitialized reads as in Figure 2.

For each of these two types of bugs, CFix has one corresponding fix strategy. As shown in Table 1(b), we enforce an all*A*–*B* order relationship to fix a dangling resource problem, and we enforce a first*A*–*B* order relationship to fix an uninitialized-read problem.

### 2.4 Strategies for Data Races

Race detectors [8, 12, 14, 36, 41, 47, 50, 68] report unsynchronized instructions, including at least one write, that can concurrently access the same variable from different threads. Race-guided testing [22, 39, 51] can identify a race-instruction pair $(I1, I2)$ such that the software fails when I2 executes immediately after I1. For CFix we implement a widely used lock-set/happens-before hybrid race-detection algorithm [2, 52] coupled with a RaceFuzzer-style testing framework [51]. This front end can identify failure-inducing interleavings as shown in Table 1(c).

Table 1(c) also illustrates two possible strategies for fixing a typical data-race bug: (1) force an order relationship, making I2 always execute before I1; or (2) force a mutual-exclusion relationship between I2 and a code region that starts from I1 and ends at a to-be-decided instruction. We use the second strategy only when the front end also reports a data race between I2 and a third instruction I3, I3 comes from the same thread as I1, and software fails when I2 executes right before I3. If all of these constraints hold, we consider both the first strategy and the second strategy that makes I1–I3 mutually exclusive with I2. In all other cases, we only consider the first strategy.

### 2.5 Strategies for Abnormal Def-Use

Some bug detectors identify abnormal inter-thread data-dependence or data-communication patterns that are either rarely observed [33, 53, 66] or able to cause particular types of software failures, such as assertion failures [70]. CFix uses such a tool, ConSeq [70], as a bug-detection front end. Each ConSeq bug report includes two pieces of information: (1) the software fails almost deterministically when a read instruction R uses values defined by a write Wb; and (2) the software is observed to succeed when R uses values defined by another write Wg. Note that ConSeq does not guarantee Wg to be the *only* correct definition of R. Therefore, CFix only uses Wg as a hint.

We refer to the case when R and Wb come from different threads as Remote-is-Bad. Depending on where Wg comes from, there are different ways to fix the bug by enforcing either mutual exclusion or ordering. Table 1(d) shows one situation that is common in practice. We refer to the case when R and Wb come from the same thread as Local-is-Bad. Enforcing orderings is the only strategy to fix this case, as shown in Table 1(d).

### 2.6 Discussion

CFix does not aim to work with deadlock detectors now, because deadlocks have very different properties from other concurrency bugs. Furthermore, there is already a widely accepted way to handle deadlocks in practice: monitor the lock-wait time and restart a thread/process when necessary.

The goal of this work is not to compare different types of bug detectors. In practice, no one bug detector is absolutely better than all others. Different detectors have different strengths and weaknesses in terms of false negatives, false positives, and performance. We leave it to software developers and tool designers to pick bug detectors. CFix simply aims to support all representative types of detectors. CFix can also work with other bug detectors, as long as failure-inducing interleavings and the identity of the involved instructions are provided. We leave extending CFix to more detectors to future work.

## 3 Enforcing an Order Relationship

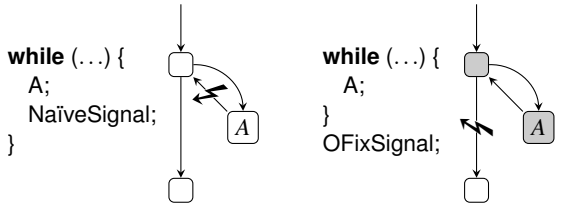
CFix uses AFix [20] to enforce mutual exclusion during patch generation. This section presents OFix, the static analysis and patch-generation component of CFix that enforces orderings, specifically all*A*–*B* and first*A*–*B* orderings. It enforces the desired ordering while making a strong effort to avoid deadlock, excessive performance loss, or needless complexity.

OFix expects bug detectors to describe a run-time operation using two vectors of information: (1) a call stack in the thread that executes that instruction, and (2) a chain of call stacks indicating how that thread has been created, which we call a *thread stack*. We refer to the instruction that performs *A* as an *A instruction*. The thread that executes the *A* instruction is a *signal thread*. All ancestors of the signal thread are called *s-create threads*. Conversely for *B* we have the *B instruction* executed by a *wait thread* whose ancestors are *w-create threads*. We write a *call stack* as $(f_0, i_0) \rightarrow (f_1, i_1) \rightarrow \cdots \rightarrow (f_n, i_n)$. $f_0$ is the starting function of a thread, which could be main or any function passed to a thread creation function. Each $i_k$ before the last is an instruction in $f_k$ that calls function $f_{k+1}$. In a signal or wait thread, the last instruction $i_n$ is the *A* instruction or *B* instruction respectively. In s-create or w-create threads, the last instruction $i_n$ calls a thread-creation function.

In Section 7 we discuss limitations of OFix and the impact on CFix as a whole system, especially those caused by the decision to try only all*A*–*B* and first*A*–*B* orderings and the choice to use call stack as operation identity.

### 3.1 Enforcing an all*A*–*B* Order

It is difficult to statically determine how many instances of *A* will be executed by a program, and hence it is difficult to find the right place to signal the end of *A* and make *B* wait




(a) May signal any number of times, waking the wait thread too early or never

(b) OFix signals exactly once; gray nodes are reaching nodes

Figure 4: Naïve signals versus OFix signals for all$A$–$B$. "↯" marks signal operations on edges.

for all instances of $A$. Consider that $A$ could be executed an unknown number of times in each signal thread; signal threads could be created an unknown number of times by each s-create thread; and s-create threads could also be created an unknown number of times. We address these challenges in four steps: (1) locate places to insert signal operations in signal threads, (2) locate places to insert signal operations in s-create threads, (3) locate places to insert wait operations, and (4) implement the signal and wait operations to coordinate all relevant threads.

**Finding Signal Locations in Signal Threads** A naïve solution that inserts signal operations right after the $A$ instruction could lead to many problems, as shown in Figure 4a. OFix aims to place signal operations so that each signal thread executes exactly one signal operation as soon as it cannot possibly execute more $A$.

Assume that $A$ has call stack $(f_0, i_0) \rightarrow \cdots \rightarrow (f_n, i_n)$ where $i_n$ is the $A$ instruction. OFix analysis starts from $f_0$. When the program can no longer execute $i_0$ in $f_0$, we know that the problematic $A$ call stack can no longer arise, and it is safe to signal. Thus OFix first analyzes the control flow graph (CFG) of $f_0$ to obtain the set of *reaching nodes*: those CFG nodes that can reach $i_0$ in zero or more steps. OFix then inserts a signal operation on each CFG edge that crosses from a reaching node to a non-reaching node.

After placing signals in $f_0$, we may need to continue down the stack to $f_1, f_2$, and so on. The critical question here is whether $i_0$ can call $f_1$ multiple times, i.e., whether $i_0$ is in a loop. If so, then $f_1$ cannot determine when $A$ will be executed for the last time in the thread. Rather, that decision needs to be made in $f_0$, outside of the outermost loop that contains $i_0$. The signal-placement algorithm stops, without continuing down the stack.

Conversely, if $f_1$ can be invoked at most once at $i_0$, we suppress the signal operation that would ordinarily be placed on the edge from $i_0$ to its successor. Instead, we delegate the signaling operation down into $f_1$ and repeat the signal-placement analysis in $f_1$. This process continues, pushing signals deeper down the stack. Signal placement stops when it reaches the end of the call stack or when it finds a call inside a loop.

Note that a function $f_k$ could be invoked under many different call stacks, while we only want $f_k$ to execute a signal operation under a particular call stack. We solve this problem using a function-cloning technique discussed in Section 3.3. All OFix-related transformations are actually applied to cloned functions.

The above strategy has two important properties. We omit proofs due to space constraints. First, each terminating execution of a signal thread signals exactly once, as shown in Figure 4b, because thread execution crosses from reaching nodes to non-reaching nodes exactly once. Second, the signal is performed as early as possible within each function, according to the CFG, and interprocedurally, benefiting from our strategy of pushing signals down the stack. These properties help ensure the correctness of our patches. They also help our patches wake up waiting threads earlier, limiting performance loss and reducing the risk of deadlocks.

**Finding Signal Locations in s-create Threads** Signal operations also need to be inserted into every s-create thread that could spawn more signal threads. Otherwise, we still would not know when we had safely passed beyond the last of all possible $A$ instances. The procedure for s-create threads matches that already used for signal threads. We apply the signal-placement analysis and transformation to each s-create thread in the thread-creation ancestry sequence that eventually leads to creation of the signal thread. This algorithm ensures that each of these ancestral threads signals exactly once immediately when it can no longer create new s-create threads or signal threads.

**Finding Wait Locations in Wait Threads** OFix must insert wait operations that can block the execution of $B$. Assuming that $(f_n, i_n)$ is the last level on the call stack of $B$, OFix creates a clone of $f_n$ that is only invoked under the bug-related call stack and thread stack. We then insert the wait operation immediately before $i_n$ in the clone of $f_n$.

**Implementing Wait and Signal Operations** Our implementation of the signal and wait operations has three parts:

**Part 1:** track the number of threads that will perform signal operations. OFix creates a global counter C in each patch. C is initialized to 1, representing the main thread, and atomically increases by 1 immediately before the creation of every signal thread and s-create thread.

**Part 2:** track how many threads have already signaled. Each signal operation atomically decrements C by 1. Since each signal or s-create thread executes exactly one signal operation, each such thread decreases C exactly once. Figure 5a shows pseudo-code for the signal operation.

**Part 3:** allow a wait thread to proceed once all threads that are going to signal have signaled. This is achieved by

```
mutex_lock(L);
if (--C == 0)
  cond_broadcast(con);
mutex_unlock(L);
```

(a) Signal operation

```
mutex_lock(L);
if (C > 0)
  cond_timedwait(con, L, t);
mutex_unlock(L);
```

(b) Wait operation

Figure 5: Pseudo-code for all$A$–$B$ operations

condition-variable broadcast/wait and the checking of C, as shown in Figure 5b.

These signal and wait operations operate on three variables: one counter, one mutex, and one condition variable. Each of these is statically associated with the order relationship it enforces.

OFix's synchronization operations are not merely semaphore operations. Since we cannot know in advance how many signal operations each wait operation should wait for, OFix relies on a well-managed counter and a condition variable to achieve the desired synchronization.

**Correctness Assessment** OFix makes a best effort to avoid introducing deadlocks by signaling as soon as possible and starting to wait as late as possible. However, it is impossible to statically guarantee that a synchronization enforcement is deadlock free. To mask potential deadlocks introduced by the patch, OFix allows every wait operation to timeout. In fact, deadlocks mostly occur when the bug requires a different fix strategy entirely, making them an important hint to guide CFix patch selection (Section 4).

Barring timeouts, the wait operation guarantees that no $B$ executes before C reaches 0. The signal operations guarantee that C reaches 0 only when all existing signal and s-create threads have signaled, which implies that no more signal or s-create threads can be created, and therefore no more $A$ instances can be executed. Thus, if no wait times out, OFix patch guarantees that no instance of $B$ executes before any instance of $A$.

**Simplicity Optimization** We use the static number of synchronization operations added by a patch as the metric for patch simplicity. In the current implementation, OFix's patches operate on LLVM bitcode [25], but we envision eventually using similar techniques to generate source-code patches. The simplicity of OFix bitcode patches would be a major factor affecting the readability of equivalent source-code patches. OFix's simplicity optimization attempts to reduce the static number of synchronization operations being added.

In the algorithm described above, OFix signal operations are inserted based solely on the calling context of $A$. In fact, a signal operation $s$ is unnecessary if it never executes in the same run as $B$, such as in Figure 6. Since identifying all such $s$ is too complicated for multi-threaded software, OFix focuses on two cases that we find common and especially useful in practice.

```
void main() {
  if (...)
    foo(); // i0
  else
    OFixSignal; // s
}
void foo() { // f1
  pthread_create(Bthread, ...); // iB1
  pthread_create(Athread, ...); // iA1
}
```

(a) Optimization case 1

```
void main() {
  if (...) {
    OFixSignal; // s
    exit(1);
  }
  pthread_create(Bthread, ...); // iB0
  pthread_create(Athread, ...); // iA0
}
```

(b) Optimization case 2

Figure 6: Removing unnecessary OFix signal operations

To ease our discussion, we use $C = (\mathsf{main}, i_0) \rightarrow \cdots \rightarrow (f_k, i_k)$ to denote the longest common prefix of the calling contexts of $A$ and $B$ where none of the call sites $i_0, \ldots, i_k$ is inside a loop. When $i_k$ is a statically-bound call, the next function along the contexts of $A$ and $B$ is the same, denoted as $f_l$. The next level of $B$'s context is denoted as $(f_l, iB_l)$.

**Case 1:** OFix signal operations in $C$ can all be removed, because they never execute in the same run as $B$. To prove this, assume $s$ to be a signal operation inserted in $f_k$. The rationale for optimizing $s$ away is as follows. First, no instance of $B$ will be executed any more once $s$ is executed: program execution cannot reach $C$ any more once it executes $s$, according to how OFix inserts signal operations. Second, no instance of $i_k$, and hence $B$, has executed yet when $s$ is executed. If this were not true, then the signal thread would signal multiple times (once inside the callee of $i_k$ and once at $s$). But this is impossible in OFix patches. Therefore, when $s$ is executed, $B$ cannot have executed yet and will not execute any more. Removing $s$ does not affect the correctness of our patch, as shown in Figure 6a.

**Case 2:** An OFix signal operation $s$ in $f_l$ can be removed if $s$ cannot reach $iB_l$ and $iB_l$ cannot reach $s$. The rationale of this optimization is similar to that of Case 1. Figure 6b shows an example of applying this optimization.

### 3.2 Enforcing a first$A$–$B$ Order

**Basic Design** To guarantee that $B$ waits for the first instance of $A$, we insert a signal operation immediately after the $A$ instruction, and a wait operation immediately before the $B$ instruction. Figure 7 shows the code for first$A$–$B$ synchronization operations. A Boolean flag and a condi-

```
if (!alreadyBroadcast) {
  alreadyBroadcast = true;
  mutex_lock(L);
  cond_broadcast(con);
  mutex_unlock(L);
}
```

(a) Signal operation

```
mutex_lock(L);
if (!alreadyBroadcast)
  cond_timedwait(con, L, t);
mutex_unlock(L);
```

(b) Wait operation

Figure 7: Pseudo-code for first$A$–$B$ operations. It contains a benign race on alreadyBroadcast.

tion variable work together to block the wait thread until at least one signal operation has executed.

**Safety-Net Design** The basic design works if the program guarantees to execute at least one instance of $A$. However, this may not be assured, in which case forcing $B$ to wait for $A$ could hang the wait thread. To address this problem, OFix enhances the basic patch with a safety net: when the program can no longer execute $A$, safety-net signals release the wait thread to continue running.

OFix first checks whether $A$ is guaranteed to execute: specifically, whether $i_k$ post-dominates the entry block of $f_k$ in each level $(f_k, i_k)$ of the $A$ call stack. A safety net is needed only when this is not true.

When a safety net is needed, OFix inserts safety-net signal operations using the all$A$–$B$ algorithm of Section 3.1. That algorithm maintains a counter C as in Figure 5a and guarantees that C drops to 0 only when the program can no longer execute $A$. To allow safety-net signal operations to wake up the wait thread, these operations share the same mutex L and the same condition variable con as those used in the basic patch in Figure 7. Whichever thread decrements C to 0 executes pthread_cond_broadcast to unblock any thread that is blocked at con. That thread also sets alreadyBroadcast to **true** so that any future instances of $B$ will proceed without waiting.

OFix checks whether $B$ is post-dominated by any safety-net signal operation. In that case, the safety net can never help wake up $B$ and is removed entirely. Lastly, OFix applies the two simplicity-optimization algorithms presented in Section 3.1 to remove unnecessary safety-net signals.

Even with the safety net, OFix does not guarantee deadlock freedom. As for all$A$–$B$ patches, OFix uses a timeout in the first$A$–$B$ wait operation to mask potential deadlocks.

### 3.3 Function Cloning

OFix clones functions to ensure that each OFix patch only takes effect under the proper call stack and thread-creation context. All patch-related transformations are applied to cloned functions.

Consider a failure-related call chain $(main, i_0) \rightarrow (f_1, i_1) \rightarrow \cdots \rightarrow (f_n, i_n)$, which chains together the call stacks of all s-create and signal threads (or all w-create and wait threads) through thread-creation function calls. Function cloning starts from the first function $f_k$ on this call chain that can be invoked under more than one calling context. OFix creates a clone $f'_k$ for $f_k$ and modifies $i_{k-1}$ based on the kind of invocation instruction at $i_{k-1}$. If $i_{k-1}$ is a direct function call to $f_k$, OFix simply changes the target of that call instruction to the clone $f'_k$. If $i_{k-1}$ calls a thread-creation function with $f_k$ as the thread-start routine, OFix changes that parameter to $f'_k$. If $f_k$ is invoked through a function pointer, OFix adds code to check the actual value of that function pointer at run time. When the pointer refers to $f_k$, our modified code substitutes $f'_k$ instead. OFix proceeds down the stack in this manner through each level of the call chain to finish the cloning. It is straightforward to prove that the above cloning technique inductively guarantees that OFix always patches under the right context.

OFix repeatedly uses the above cloning technique in various parts of the patching process, with one $f$-clone created for each unique bug-related calling context of $f$.

### 3.4 Discussion

In the algorithms given above, we always consider the calling context of a bug report. We believe this is necessary, especially for bug-detection front-ends that are based on dynamic analysis. OFix can also enforce the ordering between two static instructions, regardless of the call stack. This option can be used when call stacks are unavailable.

OFix achieves context-awareness by cloning functions, which could potentially introduce too much duplicated code. The strategy to start cloning from the first function that can be invoked under more than one calling context ensures that all clones are necessary to achieve context-awareness. Our experience shows that OFix bug fixing in practice does not introduce excessive code duplication, affecting only a small portion of the whole program.

Our current implementation uses POSIX condition variables. We could also use other synchronization primitives, such as pthread_join. Our function cloning technique and the analysis to find signal/wait locations would still be useful: by design, placement and implementation of the synchronization operations are largely orthogonal.

## 4 Patch Testing and Selection

After fix strategies are selected and synchronization relations are enforced, CFix has one or more candidate patches for each bug report. CFix tests these patches as follows.

CFix first checks the correctness of a patch through static analysis and interleaving testing. Considering the huge interleaving space, correctness testing is extremely challenging. CFix first executes the patched software once without external perturbation, referred to as an *RTest*, and then applies a guided testing where the original bug-detection

front end is used, referred to as a *GTest*. A patch is rejected under any of the following scenarios:

**Correctness check 1:** Deadlock discovered by static analysis. If an OFix wait operation is post-dominated by an OFix signal operation that operates on the same condition variable, the patch will definitely introduce deadlocks.

**Correctness check 2:** Failure in RTest. Since multi-threaded software rarely fails without external perturbation, this failure usually suggests an incorrect fix strategy. For example, according to the CTrigger bug report shown in Figure 2, CFix will try a patch that forces r to always execute after c, which causes deterministic failure.

**Correctness check 3:** Failure in GTest. This usually occurs when the patch disables the failure-inducing interleaving among some, but not all, dynamic instances of bug-related instructions.

**Correctness check 4:** Timeout in RTest. A patch timeout could be caused by a masked deadlock or a huge performance degradation of the patch; our timeout threshold is 10 seconds. CFix rejects the patch in both cases, and provides deadlock-detection result to developers.

**Correctness check 5:** Failures of related patches. Interestingly, we can use the correctness of one patch to infer that of a related patch. Consider Figure 2. If an order patch where r is forced to execute after c fails, we infer that a patch that makes r mutually exclusive with p–c is incorrect, because mutual exclusion does not prohibit the interleaving encountered by the order patch.

Rarely, CFix may not find any patch that passes correctness checking. We discuss this in Section 7.

When multiple patches pass correctness checking, CFix compares performance impacts. In our current prototype, CFix discards any patch that is at least 10% slower than some other patch. When a bug has multiple patches passing both correctness and performance checking, CFix picks the patch that introduces the fewest synchronization operations. Note that some patches can be merged and significantly improve simplicity, which we discuss in Section 5. Therefore, given two patches for the same bug report, CFix chooses the one that can be merged with other patches.

CFix includes run-time support to determine whether a timeout within CFix-patched code is caused by deadlock or not. Traditional deadlock-detection algorithms cannot discover the dependency between condition-variable wait threads and signal threads, because they cannot predict which thread will signal in the future. Inspired by previous work [20, 28], CFix addresses this challenge by starting monitoring and deadlock analysis after a timeout. By observing which thread signals on which condition variable as the post-timeout execution continues, CFix can discover circular wait relationships among threads at the moment of the timeout. This strategy imposes no overhead until a patch times out. It can be used for both patch testing and production-run monitoring. The general idea and much of the detail are the same as in the run-time for AFix [20], but we extended the run-time for AFix to support signal and wait on condition variables. Unlike Pulse [28], CFix does not require kernel modification, because it focuses only on deadlocks caused by its own patches.

CFix currently only conducts *RTest* and *GTest* using the failure-inducing inputs reported by the original bug detectors. In practice, this is usually enough to pick a good patch for the targeted bug, as demonstrated by our evaluations. In theory, this may overlook some potential problems, such as deadlock-induced timeout under other inputs and other interleavings. In such cases, we rely on our low-overhead run-time to provide feedback to refine patches.

# 5 Patch Merging

The goals of patch merging are to combine synchronization operations and variables, promote simplicity, and potentially improve performance. Merging is especially useful in practice, because a single synchronization mistake often leads to multiple bug reports within a few lines of code. Fixing these bugs one-by-one would pack many synchronization operations and variables into a few lines of the original program, harming simplicity and performance. Jin et al. [20] presented mutual-exclusion patch merging. In this section, we describe how OFix merges order-enforcing patches.

## 5.1 Patch Merging Guidelines

Patch merging in OFix is governed by four guidelines. Guideline 1: The merged patch must have statically and dynamically fewer signal and wait operations than the unmerged patches. Guideline 2: Each individual bug must still be fixed. Guideline 3: Merging must not add new deadlocks. Guideline 4: Merging should not cause significant performance loss. Note that signal operations cannot be moved earlier, per guideline 2. However, delaying signals too long can hurt performance and introduce deadlocks.

## 5.2 Patch Merging for all*A–B* Orderings

Figure 8 shows a real-world example of merging all*A–B* patches. To understand how the merging works, assume that we have enforced two all*A–B* orderings, $A_1$–$B_1$ and $A_2$–$B_2$, using patches $P_1$ and $P_2$.

OFix considers merging only if $A_1$ and $A_2$ share the same call stack and thread stack, except for the instruction at the last level of the call stacks, denoted as $(f_n, i_n^1)$ and $(f_n, i_n^2)$ for $A_1$ and $A_2$ respectively. Our rationale is to avoid moving signals too far away from their original locations, as this could dramatically affect performance (guideline 4). We do not initially consider $B_1$ and $B_2$: each patch includes just one wait operation, with little simplification potential.

Next OFix determines the locations of signal operations in the merged patch, assuming a merge will take place. To fix the original bugs (guideline 2), a signal thread must

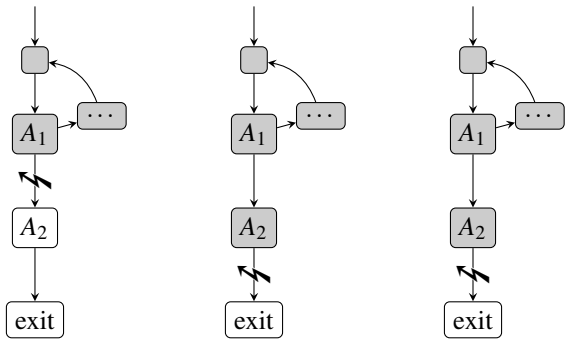
```
  while (1) {
    mutex_lock(L); // A1
    if (...) {
-     OFixSignal1;
      mutex_unlock(L); // A2
-     OFixSignal2;
+     OFixSignal∪;
      return;
    }
    ...
  }
```

(a) Signal thread

```
+ OFixWait∪;
- OFixWait1;
- OFixWait2;
  mutex_destroy(L); // B1,B2
```

(b) Wait thread

Figure 8: all*A*–*B* merging, simplified from PBZIP2. + and – denote additions and removals due to patch merging.




(a) $ReachSet_1$ and $OFixSignal_1$ (b) $ReachSet_2$ and $OFixSignal_2$ (c) $ReachSet_\cup$ and $OFixSignal_\cup$

Figure 9: CFG of the signal thread in Figure 8a. "↯" marks signal operations on edges.

execute a merged signal operation exactly once when it can no longer execute either $A_1$ or $A_2$. This leads to the same signal locations as those in patch $P_1$ and patch $P_2$, except for in $f_n$. If $P_1$ and $P_2$ do not actually place any signal operations in $f_n$, merging is done. Otherwise, we place the merged signal operations in $f_n$, so that $f_n$ signals once it can no longer execute either $i_n^1$ or $i_n^2$. Let $ReachSet_1$ and $ReachSet_2$ be the sets of nodes in $f_n$ that can reach $i_n^1$ and $i_n^2$ respectively. Let $ReachSet_\cup$ be union of $ReachSet_1$ and $ReachSet_2$. Merged signal operations should be inserted on every edge that crosses from the inside to the outside of $ReachSet_\cup$. Figure 9 shows CFGs corresponding to the code in Figure 8a, with the various reaching sets highlighted in gray.

Now that we know where signal operations would be placed if the patches were merged, we reject a merged patch if it cannot reduce the signal operation count, per guideline 1. In fact, one can prove that merging improves simplicity only when $ReachSet_1$ overlaps with $ReachSet_2$.

A final check rejects merging if it delays signal operations so much that deadlocks could be introduced (guideline 3). OFix merges only when there is no blocking operation on any path from where unmerged signal operations are to where merged signal operations would be, thereby guaranteeing not to introduce new deadlocks. Our implementation considers blocking function calls (such as lock acquisitions) and loops conditioned by heap or global variables as blocking operations. To determine whether a function call may block, CFix keeps a whitelist of non-blocking function calls.

```
  if (...) {
    Gend = end; // A1,A2
-   OFixSignal1; // i1
-   OFixSignal2; // i2
+   OFixSignal∪; // i∪
  }
```

(a) Signal thread

```
+ OFixWait∪;
- OFixWait1;
  printf("%d\n", Gend); // B1
- OFixWait2;
  printf("%d\n", Gend-init); // B2
```

(b) Wait thread

Figure 10: first*A*–*B* merging, simplified from FFT. + and – denote additions and removals due to patch merging.

OFix merges $P_1$ and $P_2$, when all of the above checks pass. OFix removes the original signal operations in $P_1$ and $P_2$, and inserts merged signal operations into locations selected above. The simplicity optimizations described in Section 3.1 are reapplied: a merged signal is removed if neither $B_1$ nor $B_2$ would execute whenever it executes.

To merge wait operations, OFix changes the two wait operations, $OFixWait_1$ and $OFixWait_2$, in $P_1$ and $P_2$ to operate on the same synchronization variables as those in the merged signal operations. OFix also has the option to replace $OFixWait_1$ and $OFixWait_2$ with a single wait, $OFixWait_\cup$, located at their nearest common dominator. This option is taken only when $OFixWait_1$ and $OFixWait_2$ share the same call stack and thread stack, and when this replacement will not introduce deadlocks. For example, $OFixWait_1$ and $OFixWait_2$ in Figure 8 pass the above checks and are merged into $OFixWait_\cup$.

This ends the merging process for a single pair of all*A*–*B* patches. The merged patch may itself be a candidate for merging with other patches. Merging continues until no suitable merging candidates remain.

### 5.3 Patch Merging for first*A*–*B* Orderings

Figure 10 provides a real-world example of merging first*A*–*B* patches, highlighting the changes made by merging.

Given two first*A*–*B* orderings, $A_1$–$B_1$ and $A_2$–$B_2$, OFix considers merging their patches only if $A_1$ and $A_2$ share the same call stack and thread stack, except for the last instruction on the call stack. This reflects the same performance concern discussed for all*A*–*B* merging. We denote the basic signal operations used to separately enforce these two orderings as $i^1$ and $i^2$, as shown in Figure 10a.

Table 2: CFix Benchmarks. Not all benchmarks have a report ID. A report ID could be a bug report ID in their corresponding bug database, which is the case for Apache, cherokee, Mozilla, MySQL and Transmission, or a forum post ID, which is the case for HTTrack and ZSNES.

| ID | App.[-ReportID] | LoC | Description |
|---|---|---|---|
| Root Cause: Order Violations | | | |
| OB1 | PBZIP2 | 2.0 K | Several variables are used in one thread after being destroyed/nullified/freed by main. |
| OB2 | x264 | 30 K | One file is read in one thread after being closed by main. |
| OB3 | FFT | 1.2 K | Several statistical variables are accessed in main before being initialized by another thread. |
| OB4 | HTTrack-20247 | 55 K | One pointer is dereferenced in main before being initialized by another thread. |
| OB5 | Mozilla-61369 | 192 K | One pointer is dereferenced in one thread before being initialized by main. |
| OB6 | Transmission-1818 | 95 K | One variable is used in one thread before being initialized by main. |
| OB7 | ZSNES-10918 | 37 K | One mutex variable is used in one thread before being initialized by main. |
| Root Cause: Atomicity Violations | | | |
| AB1 | Apache-25520 | 333 K | Threads write to the same log buffer concurrently, resulting in corrupted logs or crashes. |
| AB2 | MySQL-791 | 681 K | The call to create a new log file is not mutually exclusive with the checking of file status. |
| AB3 | MySQL-3596 | 693 K | The checking and dereference of two pointers are not mutually exclusive with the NULL assignments. |
| AB4 | Mozilla-142651 | 87 K | One memory region could be deallocated by one thread between another thread's two dereferences. |
| AB5 | Cherokee-326 | 83 K | Threads write to the same time string concurrently, resulting in corrupted time strings. |
| AB6 | Mozilla-18025 | 108 K | The checking and dereference of one pointer are not mutually exclusive with the NULL assignment. |

To maintain the $A_1$–$B_1$ and $A_2$–$B_2$ orderings (guideline 2), we should guarantee that the merged basic signal executes when both $A_1$ and $A_2$ have executed. This location, denoted as $i^{\cup}$, is the nearest common post-dominator of $i^1$ and $i^2$ that is also dominated by $i^1$ and $i^2$. OFix abandons the merge if this location $i^{\cup}$ does not exist.

After locating $i^{\cup}$, OFix checks whether merging could cause new deadlocks, checks whether the wait operations can also be merged, inserts safety-net signal operations for a merged patch, and continues merging until no suitable merging candidates remain.

# 6 Experimental Evaluation

## 6.1 Methodology

CFix includes two static analysis and patching components: (1) AFix by Jin et al. [20]; and (2) OFix, newly presented in this paper. Both AFix and OFix are built using LLVM [25]. They apply patches by modifying the buggy program's LLVM bitcode, then compiling this to a native, patched binary executable.

We evaluate CFix on 13 real-world bug cases, representing different types of root causes, from 10 open-source C/C++ server and client applications, as shown in Table 2. We collect similar numbers of bug cases from two categories: (1) bug cases that require atomicity enforcement and (2) bug cases that require order enforcement. For the first category, we use exactly the same set of bug cases as in the AFix [20] evaluation. For the second category, we gather bug cases that have been used in previous papers on ConMem [69], ConSeq [70], and DUI [53]. These three previous papers altogether contain nine bug cases that require order enforcement; we randomly select seven out of these nine. We believe that our bug case set is representative to some extent, although we cannot claim that it represents all concurrency bugs in the real world.

To patch these buggy applications, we follow the five steps described in Section 1. We first apply the four bug-detection front-ends discussed in Section 2 to these applications using the bug-triggering inputs described in the original reports. We refer to these four detectors as the *atomicity-violation front end* (AV), the *order-violation front end* (OV), the *data-race front end* (RA), and the *definition-use front end* (DU). In each case, at least one front-end detects bugs that lead to the failures described in the original reports. CFix then generates, tests, and selects patches for each of the 90 bug reports generated by AV, OV, RA, and DU. It also tries patch merging for cases with multiple bug reports before generating the final patches.

Our experiments evaluate the correctness, performance, and simplicity of CFix's final patches. Our experiments also look at the original buggy program and the program manually fixed by software developers, referred to as *original* and *manual* respectively. For fair comparison, all binaries are generated using identical LLVM settings. All experiments use an eight-core Intel Xeon machine running Red Hat Enterprise Linux 5.

## 6.2 Overall Results

The "Number of Bug Reports" columns of Table 3 show the number of reports from different bug detectors. The reports for each case from each detector are mostly very different from each other and are not just multiple stack traces of the same static instructions.

The "Overall Patch Quality" columns of Table 3 summarize our experimental results. In this table, "✓" indicates clear and complete success: the original concurrency-bug problem is completely fixed, no new bug is observed, and performance degradation is negligible. "-" indicates that

Table 3: Results of OFix patches. The $a$ and $f$ subscripts indicate all*A–B* and first*A–B* bug reports respectively. The $L$ and $R$ subscripts indicate Local-is-Bad and Remote-is-Bad definition-use bug reports respectively. ✓ a good patch; - good patch is not generated; blank: not applicable.

| | Number of Bug Reports | | | | Overall Patch Quality | | | | | | Failure Rates | | Overhead | | # of CFix |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | AV | OV | RA | DU | AV | OV | RA | DU | CFix | Manual | Original | CFix | CFix | Manual | Sync Ops |
| OB1 | 2 | $5_a$ | 4 | | ✓ | ✓ | ✓ | | ✓ | ✓ | 43% | 0% | -0.3% | 1.6% | 5 |
| OB2 | | $1_a$ | | | | ✓ | | | ✓ | ✓ | 65% | 0% | -0.1% | 3.6% | 7 |
| OB3 | 7 | $4_f$ | 10 | $4_L$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 100% | 0% | 0.2% | 0.0% | 5 |
| OB4 | 1 | $1_f$ | 2 | | ✓ | ✓ | ✓ | | ✓ | | 97% | 0% | 0.5% | | 2 |
| OB5 | 1 | | 1 | | ✓ | | ✓ | | ✓ | ✓ | 64% | 0% | 0.0% | 0.0% | 2 |
| OB6 | 1 | $1_f$ | 2 | $1_L$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 93% | 0% | 0.3% | -0.3% | 2 |
| OB7 | | $1_f$ | | | | ✓ | | | ✓ | ✓ | 97% | 0% | 0.2% | | 3 |
| AB1 | 6 | | 6 | | ✓ | | ✓ | | ✓ | ✓ | 52% | 0% | -0.9% | -0.4% | 3 |
| AB2 | 1 | | 2 | $1_R$ | ✓ | | ✓ | ✓ | ✓ | ✓ | 39% | 0% | 0.7% | 0.5% | 5 |
| AB3 | 2 | | 4 | $2_R$ | ✓ | | ✓ | - | ✓ | - | 53% | 0% | -0.0% | 1.0% | 9 |
| AB4 | 1 | | 2 | | ✓ | | ✓ | | ✓ | - | 55% | 0% | -0.5% | 0.0% | 3 |
| AB5 | 4 | | 5 | $1_R$ | ✓ | | ✓ | ✓ | ✓ | ✓ | 68% | 0% | -0.2% | 0.4% | 2 |
| AB6 | 1 | | 2 | $1_R$ | ✓ | | ✓ | ✓ | ✓ | ✓ | 42% | 0% | 0.7% | 0.5% | 5 |

CFix fails to generate a patch that passes its own internal testing. Blanks are cases where no bug is reported by the corresponding front end. For manual patches, "-" means developers submitted intermediate patches that were later found to be incomplete by developers or testers (i.e., the original software failure can still occur with the patch applied); blanks mark cases where developers have not yet provided any patch for the corresponding bug.

Overall, CFix is highly effective. Across all four bug-detection front ends and all 13 benchmarks, CFix successfully fixes all bugs except two reports for one case (AB3) under one front end (DU). CFix's final patches are all of high quality regarding correctness, performance and simplicity, comparable with the final patches designed by software developers. In several cases, CFix patches are even better than the first few patches generated by developers. In the case of HTTrack, CFix creates good patches while the developers have yet to propose any patches at all. Note that Jin et al.'s work on automated atomicity-bug fixing [20] only works with the AV front end and can only fix 6 cases, AB1 through AB6.

### 6.3 Patching for Different Bug Detectors

CFix generates one or more patches using the fixing strategies described in Section 2. Among all front ends, OV has the most straightforward fixing process. OV only detects order violations. It reports six all*A–B* violations and seven first*A–B* violations. CFix generates one ordering patch for each report. All of these patches pass correctness testing.

Fixing AV bug reports is much more challenging. As shown in Table 4, AV finds 27 atomicity violations for 11 benchmarks. Among these, 12 reports from OB1 through OB6 are side effects of order violations, akin to Figure 2. These are examples of bug reports that are different from actual root cause. The software would still fail if we merely enforced mutual exclusions based on these 12 bug reports.

Table 4: Patch testing and selection for AV front end. Subscripts $aO$, $fO$, and $A$ indicate all*A–B* ordering, first*A–B* ordering, and mutual exclusion patches respectively. C1–C5 count correctness rejections as in Section 4. P and R respectively count performance and simplicity rejections.

| | # AV | Rejected Patches | | | | | | | Final |
|---|---|---|---|---|---|---|---|---|---|
| ID | Bugs | C1 | C2 | C3 | C4 | C5 | P | R | Patch |
| OB1 | 2 | 0 | 0 | 0 | 4 | 0 | 1 | 1 | $2_{aO}$ |
| OB3 | 7 | 0 | 7 | 0 | 0 | 7 | 0 | 0 | $7_{aO}$ |
| OB4 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | $1_{fO}$ |
| OB5 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | $1_{aO}$ |
| OB6 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | $1_{aO}$ |
| AB1 | 6 | 12 | 0 | 5 | 7 | 0 | 0 | 0 | $6_A$ |
| AB2 | 1 | 2 | 0 | 1 | 1 | 0 | 0 | 0 | $1_A$ |
| AB3 | 2 | 4 | 0 | 2 | 2 | 0 | 0 | 0 | $2_A$ |
| AB4 | 1 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | $1_A$ |
| AB5 | 4 | 8 | 0 | 2 | 6 | 0 | 0 | 0 | $4_A$ |
| AB6 | 1 | 0 | 0 | 0 | 2 | 0 | 1 | 0 | $1_A$ |

CFix successfully picks patches that match the root causes and completely fixes the 11 benchmarks. Table 4 summarizes this process. For each AV bug report, CFix tries to generate 3–5 patches: one mutual-exclusion patch and two all*A–B* ordering patches, as shown in Table 1(a). If an all*A–B* patch is rejected due to timeouts or deadlocks, the corresponding first*A–B* patch is generated and tested instead. Although testing multi-threaded software is challenging, the static (C1) and dynamic (C2–C5) correctness checks complement each other and help CFix identify and reject bad patches. Most patches that do not reflect root causes are rejected this way, as shown by the C1–C5 columns of Table 4. In a few cases in OB1, OB5, and AB6, the buggy software can be fixed by either mutual-exclusion

or order synchronization. CFix generates both patches and selects the one with the best performance and simplicity.

Fixing RA bugs is also challenging, because a data-race report itself contains no root cause information. CFix successfully generates final patches for RA bug reports as follows. RA finds 19 data races for 5 benchmarks with order-violation root causes. Following Section 2.4, CFix decides that no mutual-exclusion patch is suitable for any of these bugs. The ordering patches generated by OFix all pass correctness testing and are selected as CFix's final patches. RA also finds data races for 6 benchmarks that have mutual-exclusion root causes (AB1–AB6). CFix generates mutual-exclusion patches, all of which pass correctness testing. However, all ordering patches for AB2, AB3, AB4 and some ordering patches for AB1 and AB5 are rejected, because OFix statically determines that these patches would cause deadlocks. Ordering patches for AB6 are rejected due to failures under guided testing. CFix does not generate any ordering patch for 6 data-race bugs in AB1 and AB5, because the RA front-end indicates that software fails as long as the race instructions execute one right after the other regardless of the order between them.

The DU patch-generation process is similar to that for OV in the case of Local-is-Bad reports, and is similar to that for AV in the case of Remote-is-Bad reports. However, CFix fails to generate any patch for AB3. The problem in AB3 is that two reads R1 and R2 should not read values defined by different write instructions. Unfortunately, DU simply reports R1 should not read values from a particular write instruction. CFix statically determines that disabling this data dependence, without considering R2, causes deadlocks, and therefore does not generate a patch.

CFix's final patches are generally identical or have only trivial differences as we switch from one front end to another. The two exceptions are in OB3 and OB4. In OB3, CFix generates a first*A*–*B* patch for an OV-reported bug following the fix strategy design, but an all*A*–*B* patch for three other front ends. In fact the program can only execute one instance of *A* at run time, so these two patches only differ in simplicity. In OB4, the final patch generated for RA and the one generated for AV and OV both correctly fix the reported failure without perceivable performance differences. The RA patch also fixes unreported failures under different inputs. Unless otherwise specified, we use majority vote to select final patches for evaluation results in Table 3 and Section 6.4.

### 6.4 Quality of CFix's Final Patches

**Correctness Results** As discussed in Section 4, CFix's final patches have all passed the guided testing of CFix bug-detection front end, without triggering the previously reported failures. To further test the correctness of CFix's final patches, we insert random sleeps in code regions that are involved in each bug report. The "Failure Rates" columns of Table 3 show the failure rates of the original buggy software and the CFix-patched software through 1,000 testing runs with the same random sleep patterns. As we can see, CFix patches eliminate all of the failures. In addition, the timeouts CFix inserted in its wait operations have never fired in our experiments with these final patches. Thus, our deadlock-avoidance heuristics, while imperfect in theory, perform extremely well in practice.

Manual inspection confirms that these fixes are correct and nontrivial. For example, half of the benchmarks cannot be fixed using locks alone. In OB1 and OB2, either the number of signal threads or the number of *A* instances per thread is not statically bound. Without OFix's careful analyses, naïve patches could easily lead to deadlocks or fail to fix the problem. A naïve first*A*–*B* patch without the safety net would cause FFT to hang nondeterministically.

**Performance Results** The "Overhead" columns of Table 3 show the overheads of both CFix and manual patches relative to the original buggy software. All CFix overheads are below 1%, which is comparable with correct manual patches. These results are averages across 100 non-failing runs of each version of the software with potentially-bug-triggering inputs.

For ordering patches, good performance stems from OFix's efforts to signal as soon as possible and wait as late as possible. This leads to little or no unnecessary delay in the program. For mutual-exclusion patches, good performance is due to short critical sections.

The static analyses in OFix perform well, taking less than one second to generate one patch. We anticipate no scalability problems for larger code bases.

**Simplicity Results** Jin et al. [20] have shown that AFix can provide mutual-exclusion patches with good simplicity. OFix uses patch optimization (Section 3.1) and patch merging (Section 5) to simplify ordering patches. To evaluate these two techniques, Table 5 presents detailed counts of signal operations (numbers followed by "s") and wait operations (numbers followed by "w") in CFix's final patches that are generated by OFix, under different optimization and merging strategies. In the few cases where different final patches are generated under different front ends, we present worst-case results for the final patch with the most synchronization operations.

The "All Opt" column of Table 5 counts synchronization operations with both simplicity optimizations enabled. We find that these numbers are quite moderate when patch merging is also enabled, highlighted in **bold**. Out of seven benchmarks, six can be fixed with no more than five synchronization operations. In the worst case, OB2 can be fixed with six signal operations and one wait operation. Manual inspection confirms that all of the synchronization operations in these final patches are genuinely necessary for our fixing strategy.

Table 5: Number of synchronization operations in patches

| ID | All Opt | Only 1st | Only 2nd | No Opt |
|---|---|---|---|---|
| OB1 merged | **4s**, **1w** | 19s, 1w | 4s, 1w | 19s, 1w |
| OB1 unmerged | 20s, 5w | 95s, 5w | 20s, 5w | 95s, 5w |
| OB2 | **6s**, **1w** | 8s, 1w | 20s, 1w | 22s, 1w |
| OB3 merged | **4s**, **1w** | 17s, 1w | 4s, 1w | 17s, 1w |
| OB3 unmerged | 40s,10w | 170s,10w | 40s, 10w | 170s,10w |
| OB4 merged | **1s**, **2w** | 1s, 2w | 1s, 2w | 1s, 2w |
| OB4 unmerged | 2s, 2w | 2s, 2w | 2s, 2w | 2s, 2w |
| OB5 | **1s**, **1w** | 1s, 1w | 4s, 1w | 4s, 1w |
| OB6 merged | **1s**, **1w** | 1s, 1w | 1s, 1w | 10s, 1w |
| OB6 unmerged | 2s, 2w | 2s, 2w | 2s, 2w | 20s, 2w |
| OB7 | **2s**, **1w** | 2s, 1w | 36s, 1w | 36s, 1w |

OB1 and OB3 respectively have five and ten bugs reported by the corresponding front-end. As a result, their unmerged OFix patches contain twenty five and fifty synchronization operations respectively, severely hurting the code simplicity. Fortunately, only five synchronization operations remain in the merged patches: a very modest number considering the number of bugs fixed.

The "Only 1st" and "Only 2nd" columns of Table 5 show how many synchronization operations would be required if each of the two simplicity optimizations of Section 3.1 were used in isolation, while "No Opt" shows the effect of disabling both optimizations. These large numbers on OB1 and OB3 are caused by unnecessary signal operations before **return** statements as in Figure 6b. Many such statements appear in the command-line option parsing code for these two applications. OB4 is the only benchmark that does not benefit from simplicity optimization. In fact, OFix initially adds 103 safety-net signal operations into OB4. Further analysis of OFix finds that the *B* operation in OB4 is post-dominated by a safety-net signal. OFix therefore removes the entire safety net before optimization is applied, per Section 3.2. For the other benchmarks, the optimizations are quite effective; with neither in place, our patches would have 2.5 to 12 times as many synchronization operations. Each of the two optimizations has its own strengths. OB5 and OB7 benefit from the first optimization; OB1 and OB3 benefit from the second; and OB2 and OB6 benefit from both.

Our current CFix implementation operates directly on LLVM bitcode, not source code. However, these simplicity results suggest that CFix patches are a good starting point for generating clean, readable source-level patches as well.

## 7 Limitations of CFix

Although CFix correctly fixes all bugs that require order enforcement in our evaluation using OFix, OFix is not a universal fixer for all possible bugs that require order enforcement. OFix is restricted by the fact that it only tries two different orderings, as well as by its use of call stacks to identify operations. Therefore, OFix cannot fix bugs that require all*A*–*B* or first*A*–*B* relationships between some, but not all, instances of one call stack and some instances of another call stack.

```
char *buffer[10];

void child(...) { // child-i
  ...
  buffer[i] = malloc(32); // A
  ...
}

void main() {
  for (i = 0; i < 10; ++i) {
    pthread_create(child, ...);
    buffer[i][0] = 'a'; // B
    free(buffer[i]);
  }
}
```

Figure 11: Example that presents a challenge to OFix

Figure 11 shows a bug that OFix cannot fix. Operations *A* and *B* require order enforcement, but share the **for** loop in function main. The ideal way to fix this bug is to force each dynamic instance of *B* ("buffer[i][0] = 'a'" in main) to wait until after the corresponding instance of *A* ("buffer[i] = malloc(32)" in child). The all*A*–*B* strategy cannot fix this bug, because it would make the main thread wait during the first iteration of the loop for all potential instances of *A*, which causes a deadlock-induced timeout. The first*A*–*B* strategy cannot fix this bug either: after the first *B* instance, a later instance of *B* could still happen before the corresponding instance of *A*.

To fix the above bug requires using loop indexes as part of operations' identities and enforcing order relationships accordingly. This type of code pattern is common in server applications with dispatch loops. As a result, OFix has certain limits in those applications.

The bug cases evaluated in Section 6 include bugs whose buggy code regions are contained in one loop: AB1–AB6. The patch testing and selection process of CFix has correctly judged that OFix cannot fix any of these bugs. These six bugs happen to all require atomicity enforcement and are correctly fixed by AFix.

The other two benchmarks in papers on ConMem [69], ConSeq [70], and DUI [53] require order enforcement but are not included in our evaluation. Our preliminary results show that one of them can be patched correctly by CFix. The other cannot, as it is a true order violation but the two operations share a loop as discussed earlier in this section.

CFix may fail to fix a bug in several other scenarios. First, the software may have deep design flaws and not be fixable through synchronization enforcement alone. Second, the bug detector may provide insufficient information for bug fixing, such as AB3 under DU. Third, OFix makes a best effort to avoid deadlocks, by signaling as soon as possible in signal threads and waiting as late as possible in wait threads. When OFix patches suffer deadlocks, CFix concludes that the bug should not be fixed through ordering enforcement. However, this could be wrong in rare cases. For example, complex branch conditions may cause infeasible paths and prevent OFix from identifying earlier

opportunities to signal. While possible in theory, this never occurs in our experiments: OFix successfully generates final patches without deadlocks detected during patch testing. Lastly, CFix patch testing cannot guarantee to catch all problems in a patch, as this is infeasible for any large multi-threaded application. The CFix run-time supports production-run patch monitoring and could potentially be extended to avoid deadlocks at run time [21].

Based on our experience, CFix can work with most concurrency bug detectors that report failure inducing interleavings. Adding a new detector as CFix front end mainly requires a corresponding fix-strategy design, as discussed in Section 2. CFix currently uses four front ends that report no false positives. If a future front end reports false positives or benign races, CFix may enforce some unnecessary synchronizations in the program. This may result in a patch with poor performance or that cannot pass the testing stage in the first place.

In some cases, CFix patches are more complicated than manual patches. These manual patches use non-lock/condition-variable synchronization primitives and sometimes leverage developers' knowledge of special program semantics. For example, some order violations are fixed by swapping the order of original program statements and some are fixed by using pthread_join. Future work can further simplify CFix patches in this direction.

CFix's patch currently operate in terms of LLVM bitcode. This enables quick deployment but makes developers' involvement difficult in the long term. Our evaluation shows CFix can generate compact patches containing few new synchronization operations. This lays a good foundation for eventual production of simple source-level patches. Such a transformation tool will need to consider additional source-level syntax issues that we have not addressed here. We leave such an extension to future work.

## 8 Related Work

As discussed in Section 1, many concurrency-bug detectors have been proposed. These tools aim to identify problems, not to fix them. Therefore, they inevitably leave many challenges for bug fixing, such as figuring out root causes and inserting synchronization operations correctly without unnecessary degradation in performance or simplicity. As a bug-fixing tool, CFix has considered and addressed these challenges, complementing bug detectors.

Techniques have been proposed to insert lock operations into software based on annotations [37, 57], atomic regions inferred from profiling [61], and whole-program serialization analysis [56]. QuickStep [23] automatically selects functions to put into critical sections based on race-detection results during loop parallelization. Recent work by Navabi et al. [40] parallelizes sequential software based on future-style annotations. It automatically inserts barriers to preserve sequential semantics during parallelization.

Compared with the above techniques, CFix is unique in fixing concurrency bugs reported by a wide variety of bug detectors and in synchronizing using both locks and condition variables. CFix addresses unique challenges such as fix-strategy design, simplicity optimization, patch merging, and patch testing. The static analysis conducted by OFix differs from that of Navabi et al. [40] by considering additional issues such as simplicity and performance.

Program synthesis [11, 55, 58] uses verification techniques to generate synchronized programs that satisfy certain specifications. The nature of the problem makes it hard to scale to large, real-world applications. CFix does not try to understand all synchronizations in a program and therefore avoids the associated scalability problems.

Hot-patching tools fix running software. ClearView [45] patches security vulnerabilities by modifying variable values at run time. Its design is not suitable for concurrency bugs. The LOOM system [62] provides a language for developers to specify synchronizations they want to add to a running software and deploys these synchronization changes safely. Similar to CFix, LOOM also does CFG reachability analysis for safety, and has a run-time component to recover from deadlocks. Since LOOM has different design goals from CFix, it does not need to consider issues like working with bug detectors, fix-strategy design, locating synchronization operations, handling statically-unknown numbers of signals, simplicity concerns, patch merging and testing. Tools like CFix can potentially complement LOOM by automatically generating patches for LOOM to deploy.

Run-time tools can help survive some concurrency bugs [7, 21, 24, 26, 34, 49, 59, 66, 67]. Since CFix aims to permanently fix bugs, it has different design constraints and must address unique challenges, such as fixing a wide variety of bugs completely, instead of statistically, with unknown root causes, statically locating synchronization operations while lowering the risk of deadlock, maintaining simplicity, patch testing, and patch selection.

Many record-replay tools [1, 44, 60, 63] and production-run bug detectors [6, 19, 36, 59] have been proposed. They can enable CFix to fix bugs discovered in production runs.

Deterministic systems [3–5, 9, 10, 29, 42] can make some concurrency bugs deterministically happen and some other bugs never occur. This promising approach still faces challenges, such as run-time overhead, integration with system non-determinism, language design, etc. In general, these tools address different problems than CFix. Even for software executed inside a deterministic environment, fixing bugs still requires manual intervention. CFix and these tools can complement each other. Tern [9] and Peregrine [10] proposed precondition computation to enforce specific interleavings for selected inputs. This technique can potentially be used to enable or disable CFix patches for selected inputs.

## 9 Conclusion

CFix is a framework for automatically fixing concurrency bugs. For concurrency bugs reported by a wide variety of detection tools, CFix automatically inserts synchronization operations to enforce the desired orderings/mutual-exclusion and fix the bugs. CFix uses testing to select the best patch among patch candidates, and incorporates optimization and merging algorithms to keep patches simple. Experimental evaluation shows that CFix produces high-quality patches that fix real-world bugs while exhibiting excellent performance. CFix is a significant step forward toward relieving software developers of the time-consuming and error-prone task of fixing concurrency bugs. It can be used to generate patches or patch candidates for developers. Its analysis, testing, and run-time monitoring results can also provide useful feedback to both developers and bug detection tools.

## Acknowledgments

We thank the anonymous reviewers for their invaluable feedback, and our shepherd, Jason Flinn, for his guidance in preparing the final version. We thank the Opera group from UCSD for sharing with us their bug benchmarks. This work is supported in part by DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0701957, CCF-0953478, CCF-1018180, CCF-1054616, and CCF-1217582; and a Claire Boothe Luce faculty fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

## References

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] C. Armour-Brown, J. Fitzhardinge, T. Hughes, N. Nethercote, P. Mackerras, D. Mueller, J. Seward, B. V. Assche, R. Walsh, and J. Weidendorfer. *Valgrind User Manual*. Valgrind project, 3.5.0 edition, Aug. 2009. http://valgrind.org/docs/manual/manual.html.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[7] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[9] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[10] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[11] J. Deshmukh, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Logical concurrency control from sequential proofs. In *ESOP*, 2010.

[12] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[13] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[14] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[16] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[19] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[20] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[21] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[22] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with Portend. In *ASPLOS*, 2012.

[23] D. Kim, S. Misailovic, and M. Rinard. Automatic parallelization with statistical accuracy bounds. Technical Report MIT-CSAIL-TR-2010-007, MIT, 2010. URL http://hdl.handle.net/1721.1/51680.

[24] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD*, 2007.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[26] Z. Letko, T. Vojnar, and B. Křena. AtomRace: data race and atomicity violation detector and healer. In *PADTAD*, 2008.

[27] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162.

[28] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX*, 2005.

[29] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[30] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. In *ASPLOS*, 2006.

[31] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[32] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[33] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[34] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-Aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1), 2009.

[35] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA*, 2010.

[36] D. Marino, M. Musuvathi, and S. Narayanasamy. Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[37] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[38] MySQL. Bug report time to close stats. http://bugs.mysql.com/bugstats.php, Dec. 2011.

[39] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[40] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPOPP*, 2008.

[41] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[42] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[43] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[44] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[45] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[46] K. Poulsen. Software bug contributed to blackout. http://www.securityfocus.com/news/8016, Feb. 2004.

[47] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[48] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[49] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP*, 2009.

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15, 1997.

[51] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[52] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *WBIA*, 2009.

[53] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: DefUse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[54] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *HotDep*, 2007.

[55] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[56] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Automatic atomic region identification in shared memory SPMD programs. In *OOPSLA*, 2010.

[57] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[58] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[59] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, 2011.

[60] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *ASPLOS*, 2011.

[61] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[62] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.

[63] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *FSE*, 2010.

[64] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[65] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[66] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[67] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *MICRO*, 2010.

[68] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[69] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.

[70] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

# All about Eve: Execute-Verify Replication for Multi-Core Servers

Manos Kapritsos*, Yang Wang*, Vivien Quema†, Allen Clement‡, Lorenzo Alvisi*, Mike Dahlin*
*The University of Texas at Austin †Grenoble INP ‡MPI-SWS

**Abstract:** This paper presents Eve, a new Execute-Verify architecture that allows state machine replication to scale to multi-core servers. Eve departs from the traditional agree-execute architecture of state machine replication: replicas first *execute* groups of requests concurrently and then *verify* that they can reach agreement on a state and output produced by a correct replica; if they can not, they roll back and execute the requests sequentially. Eve minimizes divergence using application-specific criteria to organize requests into groups of requests that are unlikely to interfere. Our evaluation suggests that Eve's unique ability to combine execution independence with nondetermistic interleaving of requests enables high-performance replication for multi-core servers while tolerating a wide range of faults, including elusive concurrency bugs.

## 1 Introduction

This paper presents Eve, a new Execute-Verify architecture that allows state machine replication to scale to multi-core servers.

State machine replication (SMR) is a powerful fault tolerance technique [26, 38]. Historically, the essential idea is for replicas to deterministically process the same sequence of requests so that correct replicas traverse the same sequence of internal states and produce the same sequence of outputs.

Multi-core servers pose a challenge to this approach. To take advantage of parallel hardware, modern servers execute multiple requests in parallel. However, if different servers interleave requests' instructions in different ways, the states and outputs of correct servers may diverge even if no faults occur. As a result, most SMR systems require servers to process requests sequentially: a replica finishes executing one request before beginning to execute the next [7, 27, 31, 39, 44, 50].

At first glance, recent efforts to enforce deterministic parallel execution seem to offer a promising approach to overcoming this impasse. Unfortunately, as we detail in the next section, these efforts fall short, not just because of the practical limitations of current implementations (e.g. high overhead [2, 3, 5]) but more fundamentally because, to achieve better performance, many modern replication algorithms do not actually execute operations in the same order at every replica (and sometimes do not even execute the same set of operations) [7, 11, 21, 47].

To avoid these issues, Eve's replication architecture eliminates the requirement that replicas execute requests in the same order. Instead, Eve partitions requests in batches and, after taking lightweight measures to make conflicts within a batch unlikely, it allows different replicas to execute requests within each batch in parallel, speculating that the result of these parallel executions (i.e. the system's important state and output at each replica) will match across enough replicas.

To execute requests in parallel without violating the safety requirements of replica coordination, Eve turns on its head the established architecture of state machine replication. Traditionally, deterministic replicas first *agree* on the order in which requests are to be executed and then *execute* them [7, 26, 27, 31, 38, 50]; in Eve, replicas first speculatively *execute* requests concurrently, and then *verify* that they have agreed on the state and the output produced by a correct replica. If too many replicas diverge so that a correct state/output cannot be identified, Eve guarantees safety and liveness by rolling back and sequentially and deterministically re-executing the requests.

Critical to Eve's performance are mechanisms that ensure that, despite the nondeterminism introduced by allowing parallel execution, replicas seldom diverge, and that, if they do, divergence is efficiently detected and reconciled. Eve minimizes divergence through a *mixer* stage that applies application-specific criteria to produce groups of requests that are unlikely to interfere, and it makes repair efficient through incremental state transfer and fine-grained rollbacks. Note that if the underlying program is correct under unreplicated parallel execution, then delaying agreement until after execution and, when necessary, falling back to sequential re-execution guarantees that replication remains safe and live even if the mixer allows interfering requests in the same group.

Eve's execute-verify architecture is general and ap-

plies to both crash tolerant and Byzantine tolerant systems. In particular, when Eve is configured to tolerate crash faults, it also provides significant protection against concurrency bugs, thus addressing a region of the design space that falls short of Byzantine fault tolerance but that strengthens guarantees compared to standard crash tolerance. Eve's robustness stems from two sources. First, Eve's *mixer* reduces the likelihood of triggering latent concurrency bugs by attempting to run only unlikely-to-interfere requests in parallel [25, 35]. Second, its *execute-verify* architecture allows Eve to detect and recover when concurrency causes executions to diverge, regardless of whether the divergence results from a concurrency bug or from distinct correct replicas making different legal choices.

In essence, Eve refines the assumptions that underlie the traditional implementation of state machine replication. In the agree-execute architecture, the safety requirement that correct replicas agree *on the same state and output* is reduced to the problem of guaranteeing that deterministic replicas process identical sequences of commands (i.e. agree *on the same inputs*). Eve continues to require replicas to be deterministic, but it no longer insists on them executing identical sequences of requests: instead of relying on agreement on inputs, Eve reverts to the weaker original safety requirement that replicas agree on state and output.

The practical consequence of this refinement is that in Eve correct replicas enjoy two properties that prior replica coordination protocols have treated as fundamentally at odds with each other: *nondeterministic interleaving of requests* and *execution independence*. Indeed, it is precisely through the combination of these two properties that Eve improves the state of the art for replicating multi-core servers:

1. *Nondeterministic interleaving of requests lets Eve provide high-performance replication for multi-core servers.* Eve gains performance by avoiding the overhead of enforcing determinism. For example, in our experiments with the TPC-W benchmark, Eve achieves a 6.5*x* speedup over sequential execution that approaches the 7.5*x* speedup of the original unreplicated server. For the same benchmark, Eve achieves a 4.7*x* speedup over the Remus primary-backup system [13] by exploiting its unique ability to allow independent replicas to interleave requests nondeterministically.
2. *Independence lets Eve mask a wide range of faults.* Without independently executing replicas, it is in general impossible to tolerate arbitrary faults. Independence makes Eve's architecture fully general, as our prototype supports tunable fault tolerance [9], retaining traditional state machine replication's ability to be configured to tolerate crash, omission, or Byzantine faults. Notably, we find that execution independence pays dividends even when Eve is configured to tolerate only crash or omission failures by offering the opportunity to mask some concurrency failures. Although we do not claim that our experimental results are general, we find them promising: for the TPC-W benchmark running on the H2 database, executing requests in parallel on an unreplicated server triggered a previously undiagnosed concurrency bug in H2 73 times in a span of 750K requests. Under Eve, our mixer *eliminated* all manifestations of this bug. Furthermore, when we altered our mixer to occassionally allow conflicting requests to be parallelized, Eve detected and corrected the effects of this bug 82% of the times it manifested, because Eve's independent execution allowed the bug to manifest (or not) in different ways on different replicas.

The rest of the paper proceeds as follows. In Section 2 we explain why deterministic multithreaded execution does not solve the problem of replicating multithreaded services. Section 3 describes the system model and Section 4 gives an overview of the protocol. In Section 5 we discuss the execution stage in more detail and in Section 6 we present the agreement protocols used by the verification stage for two interesting configurations and discuss Eve's ability to mask concurrency bugs. Section 7 presents an experimental evaluation of Eve, and Section 8 presents related work. Section 9 concludes the paper.

## 2 Why not deterministic execution?

Deterministic execution of multithreaded programs [2, 3, 5, 30] guarantees that, given the same input, all correct replicas of a multithreaded application will produce identical internal application states and outputs. Although at first glance this approach appears a perfect match for the challenge of multithreaded SMR on multi-core servers, there are two issues that lead us to look beyond it. The first issue [4] is straightforward: current techniques for deterministic multithreading either require hardware support [14, 15, 20] or are too slow (1.2x-10x overhead) [2, 3, 5] for production environments. The second issue originates from the semantic gap that exists between modern SMR protocols and the techniques used to achieve deterministic multithreading.

Seeking opportunities for higher throughput, SMR protocols have in recent years looked for ways to exploit the semantics of the requests processed by the replicas to achieve replica coordination without forcing all replicas to process identical sequences of inputs. For example, many modern SMR systems no longer insist that read requests be performed in the same order at all replicas, since read requests do not modify the state of the replicated application. This *read-only optimization* [7, 9, 24]

is often combined with a second optimization that allows read requests to be executed only at a *preferred quorum* of replicas, rather than at *all* replicas [21]. Several SMR systems [11, 47] use the preferred quorum optimization during failure-free executions also for requests that change the application's state, asking other replicas to execute these requests only if a preferred replica fails.

Unfortunately, deterministic multithreading techniques know nothing of the semantics of the operations they perform. Their ability to guarantee replica coordination of multithreaded servers is based purely on syntactic mechanisms that critically rely on the assumption that all replicas receive identical sequences of inputs: only then can deterministic multithreading ensure that the replicas' states and outputs will be the same. Read-only optimizations and preferred quorum operations violate that assumption, leading correct replicas to diverge. For instance, read-only requests advance a replica's instruction counter and may cause the replica to acquire additional read locks: it is easy to build executions where such low-level differences may eventually cause the application state of correct replicas to diverge [22]. Paradoxically, the troubles of deterministic replication stem from sticking to the letter of the state machine approach [26, 38], at the same time that modern SMR protocols have relaxed its requirements while staying true to its spirit.

## 3 System model

The novel architecture for state machine replication that we propose is fully general: Eve can be applied to coordinate the execution of multithreaded replicas in both synchronous and asynchronous systems and can be configured to tolerate failures of any severity, from crashes to Byzantine faults.

In this paper, we primarily target asynchronous environments where the network can arbitrarily delay, reorder, or lose messages without imperiling safety. For liveness, we require the existence of synchronous intervals during which the network is well-behaved and messages sent between two correct nodes are received and processed with bounded delay. Because synchronous primary-backup with reliable links is a practically interesting configuration [13], we also evaluate Eve in a server-pair configuration that—like primary-backup [6]—relies on timing assumptions for both safety and liveness.

Eve can be configured to produce systems that are *live*, i.e. provide a response to client requests, despite a total of up to $u$ failures, whether of omission or commission, and to ensure that all responses accepted by correct clients are *correct* despite up to $r$ commission failures and any number of omission failures [9]. Commission failures include all failures that are not omission failures. The union of omission and commission failures are Byzantine failures. However, we assume that failures do not break cryptographic primitives; i.e., a faulty node can never produce a correct node's MAC. We denote a message X sent by Y that includes an authenticator (a vector of MACs, one per receiving replica) as $\langle X \rangle_{\vec{\mu}_Y}$.

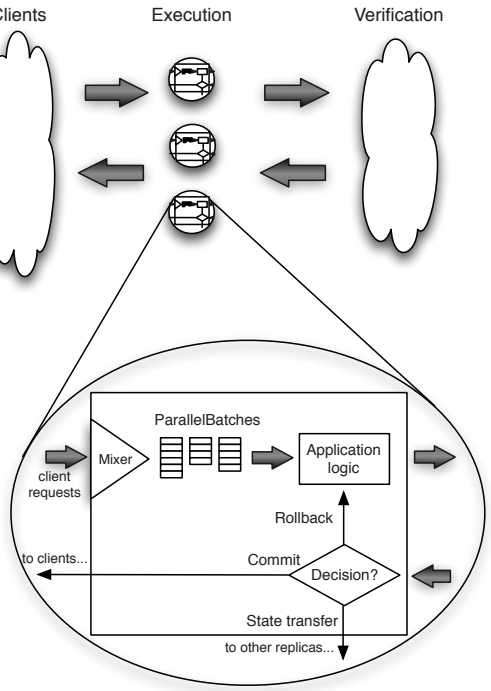



**Figure 1:** Overview of Eve.

## 4 Protocol overview

Figure 1 shows an overview of Eve, whose "execute-then-verify" design departs from the "agree-then-execute" approach of traditional SMR [7, 27, 50].

### 4.1 Execution stage

Eve divides requests in batches, and lets replicas execute requests within a batch in parallel, without requiring them to agree on the order of request execution within a batch. However, Eve takes steps to make it likely that replicas will produce identical final states and outputs for each batch.

**Batching** Clients send their requests to the current primary execution replica. The primary groups requests into batches, assigns each batch a sequence number, and sends them to all execution replicas. Multiple such batches can be in flight at the same time, but they are processed in order. Along with the requests, the primary sends any data needed to consistently process any nondeterministic requests in the batch (e.g. a seed for `random()` calls or a timestamp for `gettimeofday()` calls [7, 9]). The primary however makes no effort to

eliminate the nondeterminism that may arise when multithreaded replicas independently execute their batches.

**Mixing** Each replica runs the same deterministic *mixer* to partition each batch received from the primary into the same ordered sequence of *parallelBatches*—groups of requests that the mixer believes can be executed in parallel with little likelihood that different interleavings will produce diverging results at distinct replicas. For example, if *conflicting* requests $\rho_1$ and $\rho_2$ both modify object $A$, the mixer will place them in different parallelBatches. Section 5.1 describes the mixer in more detail.

**Executing (in parallel)** Each replica executes the parallelBatches in the order specified by the deterministic mixer. After executing all parallelBatches in the $i^{th}$ batch, a replica computes a hash of its application state and of the outputs generated in response to requests in that batch. This hash, along with the sequenceNumber $i$ and the hash for batch $i-1$,[1] constitute a *token* that is sent to the verification stage in order to discern whether the replicas have diverged. Section 5.2 describes how we efficiently and deterministically compute the hash of the final state and outputs.

### 4.2 Verification stage

Eve's execution stage strives to make divergence unlikely, but offers no guarantees: for instance, despite its best effort, the mixer may inadvertently include conflicting requests in the same parallelBatch and cause distinct correct replicas to produce different final states and outputs. It is up to the verification stage to ensure that such divergences cannot affect safety, but only performance: at the end of the verification stage, all correct replicas that have executed the $i^{th}$ batch of requests are guaranteed to have reached the same final state and produced the same outputs.

**Agreement** The verification stage runs an agreement protocol to determine the final state and outputs of all correct replicas after each batch of requests. The input to the agreement protocol (see Section 6) are the tokens received from the execution replicas. The final decision is either *commit* (if enough tokens match) or *rollback* (if too many tokens differ). In particular, the protocol first verifies whether replicas have diverged at all: if all tokens agree, the replicas' common final state and outputs are committed. If there is divergence, the agreement protocol tallies the received tokens, trying to identify a final state and outputs pair reached by enough replicas to guarantee that the pair is the product of a correct replica. If one such pair is found, then Eve ensures that all correct replicas commit to that state and outputs; if not, then the agreement protocol decides to roll back.

**Commit** If the result of the verification stage is *commit*, the execution replicas mark the corresponding sequence number as committed and send the responses for that parallelBatch to the clients.

**Rollback** If the result of the verification stage is *rollback*, the execution replicas roll back their state to the latest committed sequence number and re-execute the batch sequentially to guarantee progress. A rollback may also cause a primary change, to deal with a faulty primary. To guarantee progress, the first batch created by the new primary, which typically includes some subset of the rolled back requests, is executed sequentially by all execution replicas.

A serendipitous consequence of its "execute-verify" architecture is that Eve can often mask replica divergences caused by *concurrency bugs*, i.e. deviations from an application's intended behavior triggered by particular thread interleavings [18]. Some concurrency bugs may manifest as commission failures; however, because such failures are typically triggered probabilistically and are not the result of the actions of a strategic adversary, they can be often masked by configurations of Eve designed to tolerate only omission failures. Of course, as every system that uses redundancy to tolerate failures, Eve is vulnerable to *correlated failures* and cannot mask concurrency failures if too many replicas fail in exactly the same way. This said, Eve's architecture should help, both because the mixer, by trying to avoid parallelizing requests that interfere, makes concurrency bugs less likely and because concurrency bugs may manifest differently (if at all) on different replicas.

## 5 Execution stage

In this section we describe the execution stage in more detail. In particular, we discuss the design of the mixer and the design and implementation of the state management framework that allows Eve to perform efficient state comparison, state transfer, and rollback.

### 5.1 Mixer design

Parallel execution will result in better performance only if divergence is rare. The mission of the mixer is to identify requests that may productively be executed in parallel and to do so with low false negative and false positive rates. False negatives will cause conflicting requests to be executed in parallel, creating the potential for divergence and rollback. False positives will cause requests that could have been successfully executed in parallel to be serialized, reducing the parallelism of the execution. Note however that Eve remains safe and live independent

[1] We include the hash for the previous batch to make sure that the system only accepts valid state transitions. Verification replicas will only accept a token as valid if they have already agreed that there is a committed hash for sequence number $i-1$ that matches the one in the $i^{th}$ token.

| Transaction | Read and write keys |
|---|---|
| getBestSellers | **read**: item, author, order_line |
| getRelated | **read**: item |
| getMostRecentOrder | **read**: customer, cc_xacts, address, country, order_line |
| doCart | **read**: item<br>**write**: shopping_cart_line, shopping_cart |
| doBuyConfirm | **read**: customer, address<br>**write**: order_line, item, cc_xacts, shopping_cart_line |

**Figure 2:** The keys used for the 5 most frequent transactions of the TPC-W workload.

of the false negative and false positive rates of the mixer. A good mixer is just a performance optimization (albeit an important one).

The mixer we use for our experiments parses each request, trying to predict which state it will access: depending on the application, this state can vary from a single file or application-level object to higher level objects such as database rows or tables. Two requests conflict when they access the same object in a read/write or write/write manner. To avoid putting together conflicting requests, the mixer starts with an empty parallelBatch and two (initially empty) hash tables, one for objects being read, the other for objects being written. The mixer then scans in turn each request, mapping the objects accessed in the request to a read or write key, as appropriate. Before adding a request to a parallelBatch, the mixer checks whether that request's keys have read/write or write/write conflicts with the keys already present in the two hash tables. If not, the mixer adds the request to the parallelBatch and adds its keys to the appropriate hash table; when a conflict occurs, the mixer tries to add the request to a different parallelBatch—or creates a new parallelBatch, if the request conflicts with all existing parallelBatches.

In our experiments with the H2 Database Engine and the TPC-W workload, we simply used the names of the tables accessed in read or write mode as read and write keys for each transaction[2] (see Table 2). Note that because the mixer can safely misclassify requests, we need not explicitly capture additional conflicts potentially generated through database triggers or view accesses that may be invisible to us: Eve's verification stage allows us to be safe without being perfect. Moreover, the mixer can be improved over time using feedback from the system (e.g. by logging parallelBatches that caused rollbacks).

Although implementing a perfect mixer might prove tricky for some cases, we expect that a good mixer can be written for many interesting applications and workloads with modest effort. Databases and key-value stores are examples of applications where requests typically identify the application-level objects that will be affected—tables and values respectively. Our experience so far is encouraging. Our TPC-W mixer took 10 student-hours to build, without any prior familiarity with the TPC-W code. As demonstrated in Section 7, this simple mixer achieves good parallelism (acceptably few false positives), and we do not observe any rollbacks (few or no false negatives).

[2]Since H2 does not support row-level locking, we did not implement conflict checks at a granularity finer than a table.

## 5.2 State management

Moving from an agree-execute to an execute-verify architecture puts pressure on the implementation of state checkpointing, comparison, rollback, and transfer. For example, replicas in Eve must compute a hash of the application state reached after executing every batch of requests; in contrast, traditional SMR protocols checkpoint and compare application states much less often (e.g. when garbage collecting the request log).

To achieve efficient state comparison and fine-grained checkpointing and rollback, Eve stores the state using a copy-on-write Merkle tree, whose root is a concise representation of the entire state. The implementation borrows two ideas from BASE [36]. First, it includes only the subset of state that determines the operation of the state machine, omitting other state (such as an IP address or a TCP connection) that can vary across different replicas but has no semantic effect on the state and output produced by the application. Second, it provides an abstraction wrapper on some objects to mask variations across different replicas.

Similar to BASE and other traditional SMR systems such as PBFT, Zyzzyva, and UpRight, where programmers are required to manually annotate which state is to be included in the state machine's checkpoint [7, 9, 24, 36], our current implementation of Eve manually annotates the application code to denote the objects that should be added to the Merkle tree and to mark them as dirty when they get modified.

Compared to BASE, however, Eve faces two novel challenges: maintaining a deterministic Merkle tree structure under parallel execution and parallel hash generation as well as issues related to our choice to implement Eve in Java.

### 5.2.1 Deterministic Merkle trees

To generate the same checksum, different replicas must put the same objects at the same location in their Merkle tree. In single-threaded execution, determinism comes easily by adding an object to the tree when it is created. Determinism is more challenging in multithreaded execution when objects can be created concurrently.

There are two intuitive ways to address the problem. The first option is to make memory allocation synchronized and deterministic. This approach not only negates efforts toward concurrent memory allocation [17, 40],

but is unnecessary, since the allocation order usually does not fundamentally affect replica equivalence. The second option is to generate an ID based on object content and to use it to determine an object's location in the tree; this approach does not work, however, since many objects have the same content, especially at creation time.

Our solution is to postpone adding newly created objects to the Merkle tree until the end of the batch, when they can be added deterministically. Eve scans existing modified objects, and if one contains a reference to an object not yet in the tree, Eve adds that object into the tree's next empty slot and iteratively repeats the process for all newly added objects.

Object scanning is deterministic for two reasons. First, existing objects are already put at deterministic locations in the tree. Second, for a single object, Eve can iterate all its references in a deterministic order. Usually we can use the order in which references are defined in a class. However some classes, like *Hashtable*, do not store their references in a deterministic order; we discuss how to address these classes in Section 5.2.2.

We do not parallelize the process of scanning for new objects, since it has low overhead. We do parallelize hash generation, however: we split the Merkle tree into subtrees and compute their hashes in parallel before combining them to obtain the hash of the Merkle tree's root.

#### 5.2.2 Java Language & Runtime

The choice of implementing our prototype in Java provides us with several desirable features, including an easy way to differentiate references from other data that simplifies the implementation of deterministic scanning; at the same time, it also raises some challenges.

First, objects to which the Merkle tree holds a reference to are not eligible for Java's automatic garbage collection (GC). Our solution is to periodically perform a Merkle-tree-level scan, using a mark-and-sweep algorithm similar to Java's GC, to find unused objects and remove them from the tree. This ensures that those objects can be correctly garbage collected by Java's GC. For the applications we have considered, this scan can be performed less frequently than Java's GC, since objects in the tree tend to be "important" and have a long lifetime. In our experience this scan is not a major source of overhead.

Second, several standard set-like data structures in Java, including instances of the widely-used Hashtable and HashSet classes, are not oblivious to the order in which they are populated. For example, the serialized state of a Java Hashtable object is sensitive to the order in which keys are added and removed. So, while two set-like data structures at different replicas may contain the same elements, they may generate different checksums when added to a Merkle tree: while semantically equivalent, the states of these replicas would instead be seen as having diverged, triggering unnecessary rollbacks.

Our solution is to create wrappers [36] that abstract away semantically irrelevant differences between instances of set-like classes kept at different replicas. The wrappers generate, for each set-like data structure, a deterministic list of all the elements it contains, and, if necessary, a corresponding iterator. If the elements' type is one for which Java already provides a comparator (e.g. Integer, Long, String, etc.), this is easy to do. Otherwise, the elements are sorted using an ordered pair *(requestId, count)* that Eve assigns to each element before adding it to the data structure. Here, *requestId* is the unique identifier of the request responsible for adding the element, and *count* is the number of elements added so far to the data structure by request *requestId*. In practice, we only found the need to generate two wrappers, one for each of the two interfaces (Set and Map) commonly used by Java's set-like data structures.

## 6 Verification stage

The goal of the verification stage is to determine whether enough execution replicas agree on their state and responses after executing a batch of requests. Given that the tokens produced by the execution replicas reflect their current state as well as the state transition they underwent, all the verification stage has to decide is whether enough of these tokens match.

To come to that decision, the verification replicas use an agreement protocol [7, 27] whose details depend largely on the system model. As an optimization, read-only requests are first executed at multiple replicas without involving the verification stage. If enough replies match, the client accepts the returned value; otherwise, the read-only request is reissued and processed as a regular request. We present the protocol for two extreme cases: an asynchronous Byzantine fault tolerant system, and a synchronous primary-backup system. We then discuss how the verification stage can offer some defense against concurrency bugs and how it can be tuned to maximize the number of tolerated concurrency bugs.

### 6.1 Asynchronous BFT

In this section we describe the verification protocol for an asynchronous Byzantine fault tolerant system with $n_E = u + max(u,r) + 1$ execution replicas and $n_V = 2u + r + 1$ verification replicas [8, 9], which allows the system to remain live despite $u$ failures (whether of omission or commission), and safe despite $r$ commission failures and any number of omission failures. Readers familiar with PBFT [7] will find many similarities between these two protocols; this is not surprising, since both protocols attempt to perform agreement among $2u + r + 1$ replicas ($3f + 1$ in PBFT terminology). The main differences between these protocols stem from two factors. First, in

PBFT the replicas try to agree on the output of a single node—the primary. In Eve the object of agreement is the behavior of a collection of replicas—the execution replicas. Therefore, in Eve verification replicas use a quorum of tokens from the execution replicas as their "proposed" value. Second, in PBFT the replicas try to agree on the inputs to the state machine (the incoming requests and their order). Instead, in Eve replicas try to agree on the outputs of the state machine (the application state and the responses to the clients). As such, in the view change protocol (which space considerations compel us to discuss in full detail elsewhere [22]) the existence of a certificate for a given sequence number is enough to commit that sequence number to the next view—a prefix of committed sequence numbers is no longer required.

When an execution replica executes a batch of requests (i.e. a sequence of parallelBatches), it sends a $\langle \text{VERIFY}, \upsilon, n, T, e \rangle_{\vec{\mu}_e}$ message to all verification replicas, where $\upsilon$ is the current view number, $n$ is the batch sequence number, $T$ is the computed token for that batch, and $e$ is the sending execution replica. Recall that $T$ contains the hash of both batch $n$ and of batch $n-1$: a verification replica accepts a VERIFY message for batch $n$ only if it has previously committed a hash for batch $n-1$ that matches the one stored in $T$.

When a verification replica receives $max(u,r)+1$ VERIFY messages with matching tokens, it marks this sequence number as *preprepared* and sends a $\langle \text{PREPARE}, \upsilon, n, T, v \rangle_{\vec{\mu}_v}$ message to all other verification replicas. Similarly when a verification replica receives $n_V - u$ matching PREPARE messages, it marks this sequence number as *prepared* and sends a $\langle \text{COMMIT}, \upsilon, n, T, v \rangle_{\vec{\mu}_v}$ to all other verification replicas. Finally, when a verification replica receives $n_V - u$ matching COMMIT messages, it marks this sequence number as *committed* and sends a $\langle \text{VERIFY-RESPONSE}, \upsilon, n, T, v \rangle_{\vec{\mu}_v}$ message to all execution replicas. Note that the view number $\upsilon$ is the same as that of the VERIFY message; this indicates that agreement was reached and no view change was necessary.

If agreement can not be reached, either because of diverging replicas, asynchrony, or because of a Byzantine execution primary, the verification replicas initiate a view change.[3] During the view change, the verification replicas identify the highest sequence number (and corresponding token) that has been prepared by at least $n_V - u$ replicas and start the new view with that token. They send a $\langle \text{VERIFY-RESPONSE}, \upsilon+1, n, T, v, f \rangle_{\vec{\mu}_v}$ message to all execution replicas, where $f$ is a flag that indicates that the next batch should be executed sequentially to ensure progress. Note that in this case the view number has increased; this indicates that agreement was not reached and a rollback to sequence number $n$ is required.

[3] The view change is triggered when the commit throughput is lower than expected, similar to [10].

**Commit, State transfer and Rollback** Upon receipt of $r+1$ matching VERIFY-RESPONSE messages, an execution replica $e$ distinguishes three cases:

**Commit** If the view number has not increased and the agreed-upon token matches the one $e$ previously sent, then $e$ marks that sequence number as stable, garbage-collects any portions of the state that have now become obsolete, and releases the responses computed from the requests in this batch to the corresponding clients.

**State transfer** If the view number has not increased, but the token does not match the one $e$ previously sent, it means that this replica has diverged from the agreed-upon state. To repair this divergence, it issues a state transfer request to other replicas. This transfer is incremental: rather than transferring the entire state, Eve transfers only the part that has changed since the last stable sequence number. Incremental transfer, which uses the Merkle tree to identify what state needs to be transferred, allows Eve to rapidly bring slow and diverging replicas up-to-date.

**Rollback** If the view number has increased, this means that agreement could not be reached. Replica $e$ discards any unexecuted requests and rolls back its state to the sequence number indicated by the token $T$, while verifying that its new state matches the token (else it initiates a state transfer). The increased view number also implicitly rotates the execution primary. The replicas start receiving batches from the new primary and, since the flag $f$ was set, execute the first batch sequentially to ensure progress.

## 6.2 Synchronous primary-backup

A system configured for synchronous primary-backup has only two replicas that are responsible for both execution and verification. The primary receives client requests and groups them into batches. When a batch $\mathcal{B}$ is formed, it sends a $\langle \text{EXECUTE-BATCH}, n, \mathcal{B}, ND \rangle$ message to the backup, where $n$ is the batch sequence number and $ND$ is the data needed for consistent execution of nondeterministic calls such as `random()` and `gettimeofday()`. Both replicas apply the mixer to the batch, execute the resulting parallelBatches, and compute the state token, as described in Section 4. The backup sends its token to the primary, which compares it to its own token. If the tokens match, the primary marks this sequence number as stable and releases the responses to the clients. If the tokens differ, the primary rolls back its state to the latest stable sequence number and noti-

fies the backup to do the same. To ensure progress, they execute the next batch sequentially.

If the primary crashes, the backup is eventually notified and assumes the role of the primary. As long as the old primary is unavailable, the new primary will keep executing requests on its own. After a period of unavailability, a replica uses incremental state transfer to bring its state up-to-date before processing any new requests.

### 6.3 Tolerating concurrency bugs

A happy consequence of the execute-verify architecture is that even when configured with the minimum number of replicas required to tolerate $u$ omission faults, Eve provides some protection against concurrency bugs.

Concurrency bugs can lead to both omission faults (e.g., a replica could get stuck) and commission faults (e.g., a replica could produce an incorrect output or transition to an incorrect state). However, faults due to concurrency bugs have an important property that in general cannot be assumed for Byzantine faults: they are easy to repair. If Eve detects a concurrency fault, it can repair the fault via rollback and sequential re-execution.

**Asynchronous case** When configured with $r = 0$, Eve provides the following guarantee:

**Theorem 1.** *When configured with $n_{exec} = 2u + 1$ and $r = 0$, asynchronous Eve is safe, live, and correct despite up to u concurrency or omission faults.*

Note that safety and liveness refer to the requirements of state machine replication—that the committed state and outputs at correct replicas match and that requests eventually commit. Correctness refers to the state machine itself; a committed state is correct if it is a state that can be reached by the state machine in a fault-free run.

**Proof sketch:** The system is always safe and correct because the verifier requires $u + 1$ matching execution tokens to commit a batch. If there are at most $u$ concurrency faults and no other commission faults, then every committed batch has at least one execution token produced by a correct replica.

The system is live because if a batch fails to gather $u + 1$ matching tokens, the verifier forces the execution replicas to roll back and sequentially re-execute. During sequential execution deterministic correct replicas do not diverge; so, re-execution suffers at most $u$ omission faults and produces at least $u + 1$ matching execution tokens, allowing the batch to commit. □

When more than $u$ correlated concurrency faults produce exactly the same state and output, Eve still provides the safety and liveness properties of state machine replication, but can no longer guarantee correctness.

**Synchronous case** When configured with just $u + 1$ execution replicas, Eve can continue to operate with 1 replica if $u$ replicas fail by omission. In such configurations, Eve does not have spare redundancy and can not mask concurrency faults at the one remaining replica.

**Extra protection during good intervals** During *good intervals* when there are no replica faults or timeouts other than those caused by concurrency bugs, Eve uses spare redundancy to boost its best-effort protection against concurrency bugs to $n_E - 1$ execution replicas in both the synchronous and asynchronous cases.

For example, in the synchronous primary-backup case, when both execution replicas are alive, the primary receives both execution responses, and if they do not match, it orders a rollback and sequential re-execution. Thus, during a good interval this configuration masks one-replica concurrency failures. We expect this to be the common case.

In both the synchronous and asynchronous case Eve, when configured for $r = 0$, enters *extra protection mode* (EPM) after $k$ consecutive batches for which all $n_E$ execution replicas provided matching, timely responses. While Eve is in EPM, after the verifiers receive the minimum number of execution responses necessary for progress, they continue to wait for up to a short timeout to receive all $n_E$ responses. If the verifiers receive all $n_E$ matching responses, they commit the response. Otherwise, they order a rollback and sequential re-execution. Then, if they receive $n_E$ matching responses within a short timeout, they commit the response and remain in EPM. Conversely, if sequential re-execution does not produce $n_E$ matching and timely responses, they suspect a non-concurrency failure and exit EPM to ensure liveness by allowing the system to make progress with fewer matching responses.

## 7 Evaluation

Our evaluation tries to answer the following questions:

- What is the throughput gain that Eve provides compared to a traditional sequential execution approach?
- How does Eve's performance compare to an unreplicated multithreaded execution and alternative replication approaches?
- How is Eve's performance affected by the mixer and by other workload characteristics?
- How well does Eve mask concurrency bugs?

We address these questions by using a key-value store application and the H2 Database Engine. We implemented a simple key-value store application to perform microbenchmark measurements of Eve's sensitivity to various parameters. Specifically, we vary the amount of execution time required per request, the size of the application objects and the accuracy of our mixer, in terms of both false positives and false negatives. For the H2

Database Engine we use an open-source implementation of the TPC-W benchmark [42, 43]. For brevity, we will present the results of the browsing workload, which has more opportunities for concurrency.

Our current prototype omits some of the features described above. Specifically, although we implement the extra protection mode optimization from Section 6.3 for synchronous primary-backup replication, we do not implement it for our asynchronous configurations. Also, our current implementation does not handle applications that include objects for which Java's *finalize* method modifies state that needs to be consistent across replicas. Finally, our current prototype only supports in-memory application state.

We run our microbenchmarks on an Emulab testbed with 14x 4-core Intel Xeon @2.4 GHz, 4x 8-core Intel Xeon @2.66 GHz, and 2x 8-core hyper-threaded Intel Xeon @1.6 GHz, connected with a 1 Gb Ethernet. We were able to get limited access to 3x 16-core AMD Opteron @3.0 GHz and 2x 8-core Intel Xeon L5420 @2.5 Ghz. We use the AMD machines as execution replicas to run the TPC-W benchmark on the H2 Database Engine for both the synchronous primary-backup and the asynchronous BFT configuration (Figure 3). For the asynchronous BFT configuration we use 3 execution and 4 verifier nodes, which are sufficient to tolerate 1 Byzantine fault ($u = 1, r = 1$). The L5420 machines are running Xen and we use them to perform our comparison with Remus (Figure 10 and Figure 11).

## 7.1 H2 Database with TPC-W

Figures 3 demonstrates the performance of Eve for the H2 Database Engine [19] with the TPC-W browsing workload [42, 43]. We report the throughput of Eve using an asynchronous BFT configuration (*Eve-BFT*) and a synchronous active primary-backup configuration (*Eve-PrimaryBackup*). We compare against the throughput achieved by an unreplicated server that uses sequential execution regardless of the number of available hardware threads (*sequential*). Note that this represents an upper bound of the performance achievable by previous replication systems that use sequential execution [7, 9, 27, 31]. We also compare against the performance of an unreplicated server that uses parallel execution.

With 16 execution threads, Eve achieves a speedup of 6.5x compared to sequential execution. That approaches the 7.5x speedup achieved by an unreplicated H2 Database server using 16 threads.

In both configurations and across all runs and for all data points, Eve never needs to roll back. This suggests that our simple mixer never parallelized requests it should have serialized. At the same time, the good speedup indicates that it was adequately aggressive in identifying opportunities for parallelism.

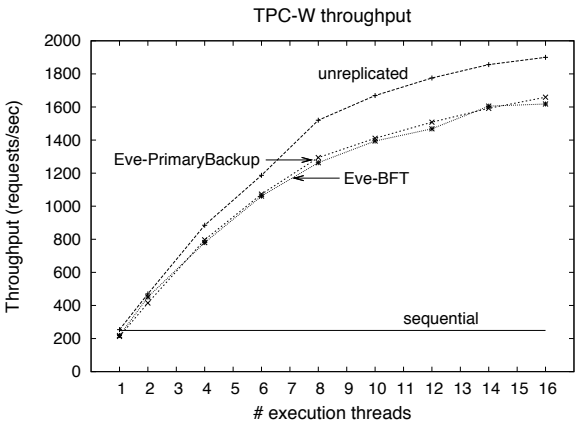



**Figure 3:** The throughput of Eve running the TPC-W browsing workload on the H2 Database Engine.

## 7.2 Microbenchmarks

In this section, we use a simple key-value store application to measure how various parameters affect Eve's performance. Due to lack of space, we only show the graphs for the primary-backup configuration; the results for asynchronous replication are similar. Except when noted, the default workload consumes 1 ms of execution time per request, each request updates one application object, and the application object size is 1 KB.

Figure 4 shows the impact of varying the CPU demand of each request. We observe that heavier workloads (10 ms of execution time per request) scale well, up to 12.5x on 16 threads compared to sequential execution. As the workload gets lighter, the overhead of Eve becomes more pronounced. Speedups fall to 10x for 1 ms/request and to 3.3x for 0.1 ms/request. The 3.3x scaling is partially an artifact of our inability to fully load the server with lightweight requests. In our workload generator, clients have 1 outstanding request at a time, thus requiring a high number of clients to saturate the servers; this causes our servers to run out of sockets before they are fully loaded. We measure our server CPU utilization during this experiment to be about 30%.

In Figure 4 we plot throughput speedup, so that trends are apparent. For reference, the absolute peak throughputs in requests per second are 25.2K, 10.0K, 1242 for the 0.1 ms, 1 ms, 10 ms lines, respectively.

The next experiment explores the impact of the application object size on the system throughput. We run the experiment using object sizes of 10 B, 1 KB, and 10 KB. Figure 5 shows the results. While the achieved throughput scales well for object sizes of 10 B and 1 KB, its scalability decreases for larger objects (10 KB). This is an artifact of the hashing library we use, as it first copies the object before computing its hash: for large objects, this memory copy limits the achievable throughput. Note that in this figure we plot throughput speedup rather than

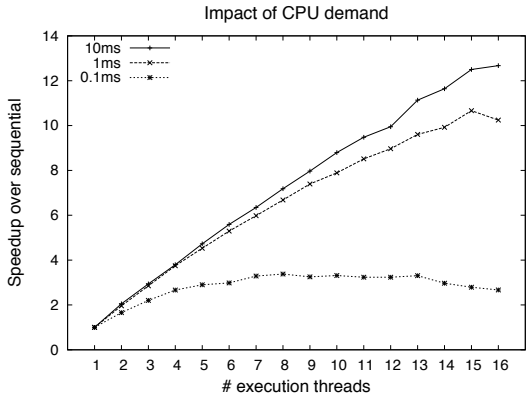



**Figure 4:** The impact of CPU demand per request on Eve's throughput speedup.

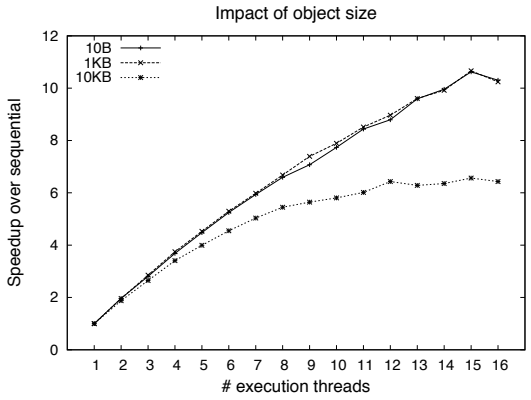



**Figure 5:** The impact of application object size on Eve's throughput speedup.

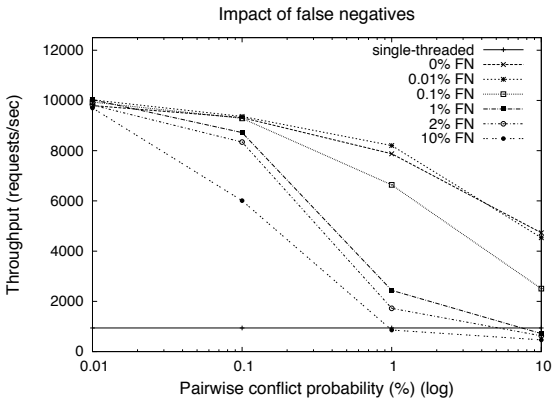



**Figure 6:** The impact of conflict probability and false negative rate on Eve's throughput.

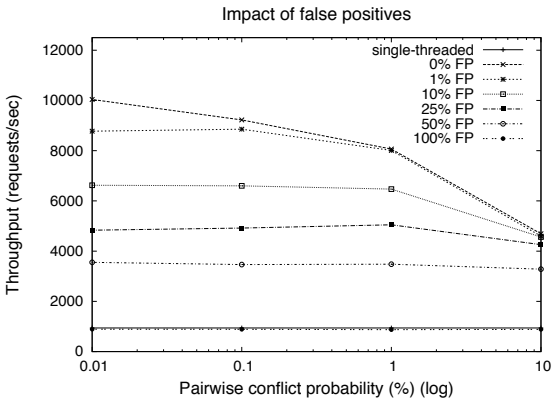



**Figure 7:** The impact of conflict probability and false positive rate on Eve's throughput.

absolute throughput to better indicate the trends across workloads. For reference, the absolute peak throughput values in requests per second are 10.0K, 10.0K, 5.6K for the 10 B, 1 KB, 10 KB lines, respectively.

Next, we evaluate Eve's sensitivity to inaccurate mixers. Specifically, we explore the limits of tolerance to false negatives (misclassifying conflicting requests as non-conflicting) and false positives (misclassifying non-conflicting requests as conflicting). The effect of these parameters is measured as a function of the pairwise conflict probability: the probability that two requests have a conflict. In practice, we achieve this by having each request modify one object and then varying the number of application objects. For example, to produce a 1% conflict chance, we create 100 objects. Similarly, a 1% false negative rate means that each pair of conflicting requests has a 1% chance of being classified as non-conflicting.

Figure 6 shows the effect of false negatives on throughput. First notice that, even for 0% false negatives, the throughput drops as the pairwise conflict chance increases due to the decrease of available parallelism. For example, if a batch has 100 requests and each request has a 10% chance of conflicting with each other request, then a perfect mixer is likely to divide the batch into about 10 parallelBatches, each with about 10 requests.

When we add false negatives, we add rollbacks, and the number of rollbacks increases with both the underlying conflict rate and the false negative rate. Notice that the impact builds more quickly than one might expect because there is essentially a birthday "paradox"—if we have a 1% conflict rate and a 1% false negative rate, then the probability that any pair of conflicting requests be misclassified is 1 in 10000. But in a batch of 100 requests, each of these requests has about a 1% chance of being party to a conflict, which means there is about a 39% chance that a batch of 100 requests contain an undetected conflict. Furthermore, with a 1% conflict rate, the batch will be divided into only a few parallelBatches, so there is a good chance that conflicting requests will land in the same parallelBatch. In fact, in this case we measure 1 rollback per 7 parallelBatches executed. Despite this high conflict rate and this high number of rollbacks, Eve achieves a speedup of 2.6x compared to sequential execution.

Figure 7 shows the effect of false positives on throughput. As expected, increased false positive ratios can lead

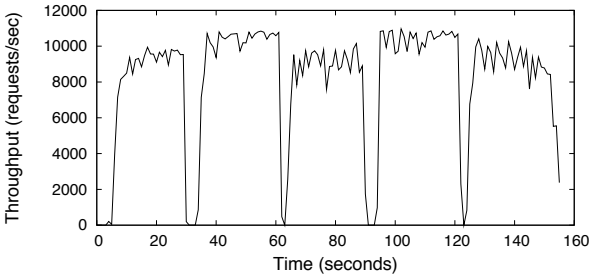



**Figure 8:** Throughput during node crash and recovery for an Eve primary-backup configuration.

to lower throughput, but the effect is not as significant as for false negatives. The reason is simple: false positives reduce the opportunities for parallel execution, but they don't incur any additional overhead.

From these experiments, we conclude that Eve does require a good mixer to achieve good performance. This requirement does not particularly worry us. We found it easy to build a mixer that (to the best of our knowledge) detects all conflicts and still allows for a good amount of parallelism. Others have had similar experience [25]. Although creating perfect mixers may be difficult in some cases, we speculate that it will often be feasible to construct mixers with the low false negative rates and modest false positive rates needed by Eve.

## 7.3 Failure and recovery

In Figure 8, we demonstrate Eve's ability to mask and recover from failures. In the primary-backup configuration we run an experiment where we kill the primary node $n_1$ at $t = 30$ seconds and recover it at $t = 60$ seconds (by which time the secondary $n_2$ has become the new primary). We then kill the new secondary ($n_1$) at $t = 90$ seconds and recover it at $t = 120$ seconds. We observe that after the first failure the throughput drops to zero until the backup realizes that the primary is dead after a timeout of 4 seconds.[4] The backup then assumes the role of the primary and starts processing requests. The throughput during this period is higher because the new primary knows that the other node is crashed and does not send any messages to it. At $t = 60$, the first node recovers, and the throughput drops to zero for about one second while the newly recovered node catches up. Then the throughput returns to its original value. The process repeats when $n_1$ crashes again at $t = 90$ seconds and recovers at $t = 120$ seconds.

## 7.4 Concurrency faults

To evaluate Eve's ability to mask concurrency faults, we use a primary-backup configuration with 16 execution threads and run the TPC-W browsing workload on the H2 Database Engine with various mixers. H2 has a previously undiagnosed concurrency bug in which a row counter is not incremented properly when multiple requests access the same table in *read_uncommitted* mode. Our standard mixer completely masks this bug because it does not let requests that modify the same table execute in parallel. By introducing less accurate mixers we explore how well Eve's second line of defense—parallel execution—works in masking this bug.

Figure 9 shows the number of times that the bug manifested in one or both replicas. When the bug manifests only in one replica, Eve detects that the replicas have diverged and repairs the damage by rolling back and re-executing sequentially. If the bug happens to manifest in both replicas in the same way, Eve will not detect it.

The first column shows the results when there is a trivial aggressive mixer that places all requests of batch $i$ in the same parallelBatch. In this case, all requests that arrive together in a batch are allowed to execute in parallel. Naturally, this case has the highest number of bug manifestations. We observe that even when the mixer does no filtering at all, Eve masks 82% of the instances where the bug manifests. In the remaining 18% of the cases, the bug manifested in the same way in both replicas and was not corrected by Eve. In columns 2 through 4, we introduce mixers with high rates of false negatives. This results in fewer manifestations of the bug, with Eve still masking the majority of those manifestations. In the fifth column, we show results for our original mixer, which (to the best of our knowledge) does not introduce false negatives. In this case, the bug does not manifest at all.

Although we do not claim that these results are general, we find them promising.

## 7.5 Remus

Remus [13] is a primary-backup system that uses Virtual Machines (VMs) to send modified state from the primary to the backup. An advantage of this approach is that it is simple and requires no modifications to the application. A drawback of this approach is that it aggressively utilizes network resources to keep the backup consistent with the primary. The issue is aggravated by two properties of Remus. First, Remus does not make fine-grain distinctions between state that is required for the state machine and temporary state. Second, Remus operates on the VM level, which forces it to send entire pages, rather than just the modified objects. Also, because Remus is using passive replication, it tolerates a narrower range of faults than Eve. Our experiments show that, despite Eve's stronger guarantees, it outperforms Remus by a factor of $4.7x$, while using two orders of magnitude less network bandwidth.

Figure 10 shows the throughput achieved by Remus and Eve on the browsing workload of the TPC-W bench-

[4] One could use a fast failure detector [29] to achieve sub-second detection.

| | Group all | 1% FN | 0.5% FN | 0.1% FN | Original Mixer |
|---|---|---|---|---|---|
| Times bug manifested | 73 | 51 | 29 | 4 | 0 |
| Fixed with rollback | 60 | 38 | 18 | 3 | 0 |
| All identical (not masked) | 13 | 13 | 11 | 1 | 0 |
| Throughput | 1104 | 1233 | 1240 | 1299 | 1322 |

**Figure 9:** Effectiveness of Eve in masking concurrency bugs when various mixers are used.

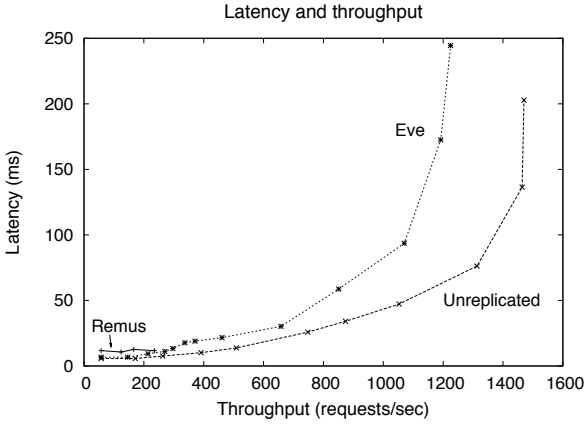



**Figure 10:** The latency and throughput of Remus and Eve running the H2 Database Engine on Xen. Both systems use a 2-node configuration. The workload is the browsing workload of the TPC-W benchmark.

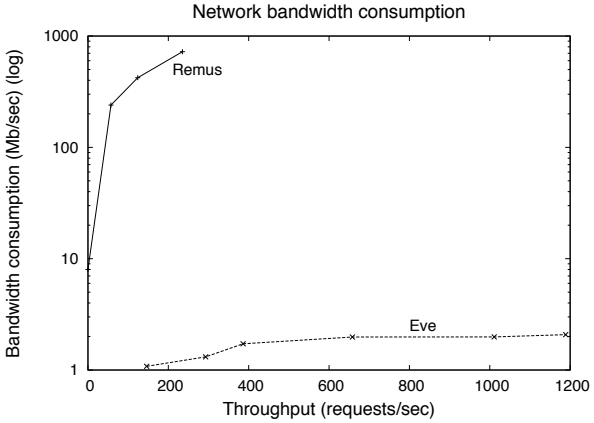



**Figure 11:** The bandwidth consumption of Remus and Eve for the experiment shown in Figure 10.

mark. We also show the latency and throughput of the unreplicated system for the same workload. Both systems run the H2 Database Engine on Xen and using a 2-node (primary-backup) configuration. Remus achieves a maximum throughput of 235 requests per second, while Eve peaks at 1225 requests per second. Remus crashes for loads higher than 235 requests per second, as its bandwidth requirements approach the capacity of the network, as Figure 11 shows. In contrast with Remus, Eve executes requests independently at each replica and does not need to propagate state modifications over the network. The practical consequence is that Eve uses significantly less bandwidth, achieves higher throughput, and provides stronger guarantees compared to a passive replication approach like Remus.

### 7.6 Latency and batching

Figure 10 provides some insight in Eve's tradeoff between latency and throughput. When Eve is not saturated, its latency is only marginally higher than that of an unreplicated server. As the load increases, Eve's latency increases somewhat, until it finally spikes up at the saturation point, at a throughput of 1225 requests per second; the unreplicated server's latency spikes up at around 1470 requests per second. To keep its latency low while maintaing a high peak throughput, Eve uses a dynamic batching scheme: the batch size decreases when the demand is low (providing good latency), and increases when the system starts becoming saturated, in order to leverage as much parallelism as possible.

## 8 Related Work

Vandiver et al. [45] describe a Byzantine-tolerant semi-active replication scheme for transaction processing systems. Their system supports concurrent execution of queries but its scope is limited: it applies to the subset of transaction processing systems that use strict two-phase locking (2PL). A recent paper suggests that it may be viable to enforce deterministic concurrency control in transactional systems [41], but the general case remains hard. Kim et al. [23] recently proposed applying this idea to a transactional operating system. This approach assumes that all application state is manageable by the kernel and does not handle in-memory application state.

One alternative is to use a replication technique other than state machine replication. *Semi-active replication* [34] weakens state machine replication with respect to both determinism and execution independence: *one* replica, the primary, executes nondeterministically and logs all the nondeterministic actions it performs. All other replicas then execute by deterministically reproducing the primary's choices. In this context, one may hope to be able to leverage the large body of work on deterministic multiprocessor replay [1, 12, 16, 28, 32, 33, 37, 46, 48, 49]. Unfortunately, relaxing the requirement of independent execution makes these systems vulnerable to commission failures. Also, similar to determin-

istic multithreaded execution approaches, record and replay approaches assume that the same input is given to all replicas. As discussed in Section 2 this assumption is violated in modern replication systems.

The Remus primary-backup system [13] takes a different approach: the backup does not execute requests, but instead passively absorbs state updates from the primary: since execution occurs only at the primary, the costs and difficulty of coordinating parallel execution are sidestepped. These advantages however come at a significant price in terms of fault coverage: Remus can only tolerate omission failures—all commission failures, including common failures such as concurrency bugs, are beyond its reach. Like Remus, Eve neither tracks nor eliminates nondeterminism, but it manages to do so without forsaking fault coverage; further, despite its stronger guarantees, Eve outperforms Remus by a factor of $4.7x$ and uses two orders of magnitude less network bandwidth (see Section 7.5) because it can ensure that the states of replicas converge without requiring the transfer of all modified state.

One of the keys to Eve's ability to combine independent execution with nondeterministic interleaving of requests is the use of the mixer, which allows replicas to execute requests concurrently with low chance of interference. Kotla et al. [25] use a similar mechanism to improve the throughput of BFT replication systems. However, since they still assume a traditional agree-execute architecture, the safety of their system depends on the assumption that the criteria used by the mixer never mistakenly parallelize conflicting requests: a single unanticipated conflict can lead to a safety violation.

Both Eve and Zyzzyva [24] allow speculative execution that precedes completion of agreement, but the assumptions on which Eve and Zyzzyva rest are fundamentally different. Zyzzyva depends on correct nodes being deterministic, so that agreement on inputs is enough to guarantee agreement on outputs: hence, a replica need only send (a hash of) the sequence of requests it has executed to convey its state to a client. In contrast, in Eve there is no guarantee that correct replicas, even if they have executed the same batch of requests, will be in the same state, as the mixer may have incorrectly placed conflicting requests in the same parallelBatch.

We did contemplate an Eve implementation in which verification is not performed within the logical boundaries of the replicated service but, as in Zyzzyva, it is moved to the clients to reduce overhead. For example, a server's reply to a client's request could contain not just the response, but also the root of the Merkle tree that encodes the server's state. However, since agreement is not a bottleneck for the applications we consider, we ultimately chose to heed the lessons of Aardvark [10] and steer away from the corner cases that such an implementation would have introduced.

# 9 Conclusion

Eve is a new execute-verify architecture that allows state machine replication to scale to multi-core servers. By revisiting the role of determinism in replica coordination, Eve enables new SMR protocols that for the first time allow replicas to interleave requests nondeterministically and execute independently. This unprecedented combination is critical to both Eve's scalability and to its generality, as Eve can be configured to tolerate both omission and commission failures in both synchronous and asynchronous settings. As an added bonus, Eve's unconventional architecture can be easily tuned to provide low-cost, best-effort protection against concurrency bugs.

# Acknowledgements

We thank our shepherd Robert Morris, and the OSDI reviewers for their insightful comments. This work was supported by NSF grants NSF-CiC-FRCC-1048269 and CNS-0720649.

# References

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 2010.

[4] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[5] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *CDCCA*, 1992.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[8] A. Clement. *UpRight Fault Tolerance*. PhD thesis, The University of Texas at Austin, Dec. 2010.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *SOSP*, 2009.

[10] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[11] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[15] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. In *ASPLOS*, 2011.

[16] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay for multiprocessor virtual machines. In *VEE*, 2008.

[17] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD, April 2006.

[18] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, 2010.

[19] H2. The H2 home page. http://www.h2database.com.

[20] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. In *HPCA*, 2011.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX*, 2010.

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers (extended version). Technical Report TR-12-23, Department of Computer Science, The University of Texas at Austin, September 2012.

[23] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter. Improving server applications with system transactions. In *EuroSys*, 2012.

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[25] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *DSN*, 2004.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[27] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[28] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In *ASPLOS*, 2010.

[29] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *SOSP*, 2011.

[30] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[31] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *OSDI*, 2008.

[32] J. T. Pablo Montesinos, Luis Ceze. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

[33] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessor. In *SOSP*, 2009.

[34] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4. *ACM OSR*, 1991.

[35] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.

[36] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In *SOSP*, 2001.

[37] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM TCS*, 1999.

[38] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 1990.

[39] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In *NSDI*, 2009.

[40] Sun Microsystems, Inc. Memory management in the Java HotSpot virtual machine, April 2006.

[41] A. Thomson and D. J. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[42] TPC-W. Open-source TPC-W implementation. http://pharm.ece.wisc.edu/tpcw.shtml.

[43] Transaction Processing Performance Council. The TPC-W home page. http://www.tpc.org/tpcw.

[44] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM*, 1996.

[45] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, 2007.

[46] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *ASPLOS*, 2011.

[47] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT. In *Eurosys*, 2011.

[48] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[49] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MOBS*, 2007.

[50] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, 2003.

# Spanner: Google's Globally-Distributed Database

*James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford*

*Google, Inc.*

## Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

## 1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

provides externally consistent [16] reads and writes, and globally-consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent MapReduce executions [12], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

These features are enabled by the fact that Spanner assigns globally-meaningful commit timestamps to transactions, even though transactions may be distributed. The timestamps reflect serialization order. In addition, the serialization order satisfies external consistency (or equivalently, linearizability [20]): if a transaction $T_1$ commits before another transaction $T_2$ starts, then $T_1$'s commit timestamp is smaller than $T_2$'s. Spanner is the first system to provide such guarantees at global scale.

The key enabler of these properties is a new TrueTime API and its implementation. The API directly exposes clock uncertainty, and the guarantees on Spanner's timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty. Google's cluster-management software provides an implementation of the TrueTime API. This implementation keeps uncertainty small (generally less than 10ms) by using multiple modern clock references (GPS and atomic clocks).

Section 2 describes the structure of Spanner's implementation, its feature set, and the engineering decisions that went into their design. Section 3 describes our new TrueTime API and sketches its implementation. Section 4 describes how Spanner uses TrueTime to implement externally-consistent distributed transactions, lock-free read-only transactions, and atomic schema updates. Section 5 provides some benchmarks on Spanner's performance and TrueTime behavior, and discusses the experiences of F1. Sections 6, 7, and 8 describe related and future work, and summarize our conclusions.

## 2 Implementation

This section describes the structure of and rationale underlying Spanner's implementation. It then describes the *directory* abstraction, which is used to manage replication and locality, and is the unit of data movement. Finally, it describes our data model, why Spanner looks like a relational database instead of a key-value store, and how applications can control data locality.

A Spanner deployment is called a *universe*. Given that Spanner manages data globally, there will be only a handful of running universes. We currently run a test/playground universe, a development/production universe, and a production-only universe.

Spanner is organized as a set of *zones*, where each zone is the rough analog of a deployment of Bigtable servers [9]. Zones are the unit of administrative deployment. The set of zones is also the set of locations across which data can be replicated. Zones can be added to or removed from a running system as new datacenters are brought into service and old ones are turned off, respectively. Zones are also the unit of physical isolation: there may be one or more zones in a datacenter, for example, if different applications' data must be partitioned across different sets of servers in the same datacenter.

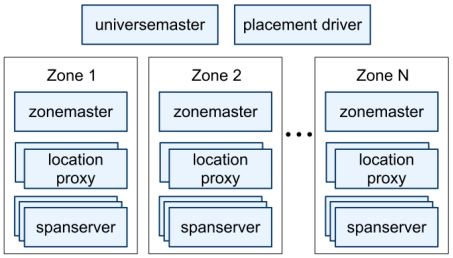



Figure 1: Spanner server organization.

Figure 1 illustrates the servers in a Spanner universe. A zone has one *zonemaster* and between one hundred and several thousand *spanservers*. The former assigns data to spanservers; the latter serve data to clients. The per-zone *location proxies* are used by clients to locate the spanservers assigned to serve their data. The *universe master* and the *placement driver* are currently singletons. The universe master is primarily a console that displays status information about all the zones for interactive debugging. The placement driver handles automated movement of data across zones on the timescale of minutes. The placement driver periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load. For space reasons, we will only describe the spanserver in any detail.

### 2.1 Spanserver Software Stack

This section focuses on the spanserver implementation to illustrate how replication and distributed transactions have been layered onto our Bigtable-based implementation. The software stack is shown in Figure 2. At the bottom, each spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*. A tablet is similar to Bigtable's tablet abstraction, in that it implements a bag of the following mappings:

$$(\text{key:string}, \text{timestamp:int64}) \rightarrow \text{string}$$

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multi-version database than a key-value store. A

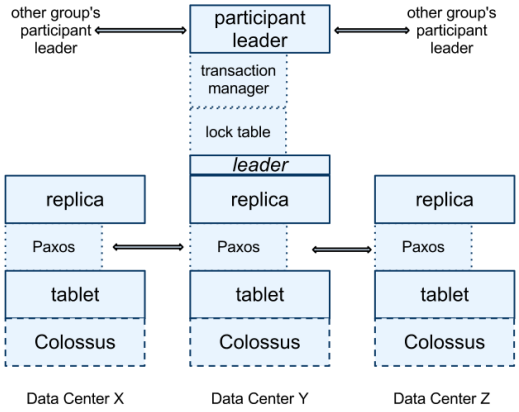



Figure 2: Spanserver software stack.

tablet's state is stored in set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System [15]).

To support replication, each spanserver implements a single Paxos state machine on top of each tablet. (An early Spanner incarnation supported multiple Paxos state machines per tablet, which allowed for more flexible replication configurations. The complexity of that design led us to abandon it.) Each state machine stores its metadata and log in its corresponding tablet. Our Paxos implementation supports long-lived leaders with time-based leader leases, whose length defaults to 10 seconds. The current Spanner implementation logs every Paxos write twice: once in the tablet's log, and once in the Paxos log. This choice was made out of expediency, and we are likely to remedy this eventually. Our implementation of Paxos is pipelined, so as to improve Spanner's throughput in the presence of WAN latencies; but writes are applied by Paxos in order (a fact on which we will depend in Section 4).

The Paxos state machines are used to implement a consistently replicated bag of mappings. The key-value mapping state of each replica is stored in its corresponding tablet. Writes must initiate the Paxos protocol at the leader; reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date. The set of replicas is collectively a *Paxos group*.

At every replica that is a leader, each spanserver implements a *lock table* to implement concurrency control. The lock table contains the state for two-phase locking: it maps ranges of keys to lock states. (Note that having a long-lived Paxos leader is critical to efficiently managing the lock table.) In both Bigtable and Spanner, we designed for long-lived transactions (for example, for report generation, which might take on the order of minutes), which perform poorly under optimistic concurrency control in the presence of conflicts. Operations

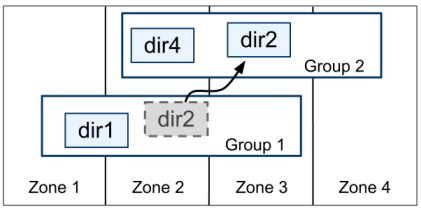



Figure 3: Directories are the unit of data movement between Paxos groups.

that require synchronization, such as transactional reads, acquire locks in the lock table; other operations bypass the lock table.

At every replica that is a leader, each spanserver also implements a *transaction manager* to support distributed transactions. The transaction manager is used to implement a *participant leader*; the other replicas in the group will be referred to as *participant slaves*. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the *coordinator leader*, and the slaves of that group as *coordinator slaves*. The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).

## 2.2 Directories and Placement

On top of the bag of key-value mappings, the Spanner implementation supports a bucketing abstraction called a *directory*, which is a set of contiguous keys that share a common prefix. (The choice of the term *directory* is a historical accident; a better term might be *bucket*.) We will explain the source of that prefix in Section 2.3. Supporting directories allows applications to control the locality of their data by choosing keys carefully.

A directory is the unit of data placement. All data in a directory has the same replication configuration. When data is moved between Paxos groups, it is moved directory by directory, as shown in Figure 3. Spanner might move a directory to shed load from a Paxos group; to put directories that are frequently accessed together into the same group; or to move a directory into a group that is closer to its accessors. Directories can be moved while client operations are ongoing. One could expect that a 50MB directory can be moved in a few seconds.

The fact that a Paxos group may contain multiple directories implies that a Spanner tablet is different from

a Bigtable tablet: the former is not necessarily a single lexicographically contiguous partition of the row space. Instead, a Spanner tablet is a container that may encapsulate multiple partitions of the row space. We made this decision so that it would be possible to colocate multiple directories that are frequently accessed together.

*Movedir* is the background task used to move directories between Paxos groups [14]. Movedir is also used to add or remove replicas to Paxos groups [25], because Spanner does not yet support in-Paxos configuration changes. Movedir is not implemented as a single transaction, so as to avoid blocking ongoing reads and writes on a bulky data move. Instead, movedir registers the fact that it is starting to move data and moves the data in the background. When it has moved all but a nominal amount of the data, it uses a transaction to atomically move that nominal amount and update the metadata for the two Paxos groups.

A directory is also the smallest unit whose geographic-replication properties (or *placement*, for short) can be specified by an application. The design of our placement-specification language separates responsibilities for managing replication configurations. Administrators control two dimensions: the number and types of replicas, and the geographic placement of those replicas. They create a menu of named options in these two dimensions (e.g., *North America, replicated 5 ways with 1 witness*). An application controls how data is replicated, by tagging each database and/or individual directories with a combination of those options. For example, an application might store each end-user's data in its own directory, which would enable user *A*'s data to have three replicas in Europe, and user *B*'s data to have five replicas in North America.

For expository clarity we have over-simplified. In fact, Spanner will shard a directory into multiple *fragments* if it grows too large. Fragments may be served from different Paxos groups (and therefore different servers). Movedir actually moves fragments, and not whole directories, between groups.

## 2.3 Data Model

Spanner exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general-purpose transactions. The move towards supporting these features was driven by many factors. The need to support schematized semi-relational tables and synchronous replication is supported by the popularity of Megastore [5]. At least 300 applications within Google use Megastore (despite its relatively low performance) because its data model is simpler to manage than Bigtable's, and because of its support for synchronous replication across datacenters. (Bigtable only supports eventually-consistent replication across datacenters.) Examples of well-known Google applications that use Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine. The need to support a SQL-like query language in Spanner was also clear, given the popularity of Dremel [28] as an interactive data-analysis tool. Finally, the lack of cross-row transactions in Bigtable led to frequent complaints; Percolator [32] was in part built to address this failing. Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings [9, 10, 19]. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.

The application data model is layered on top of the directory-bucketed key-value mappings supported by the implementation. An application creates one or more *databases* in a universe. Each database can contain an unlimited number of schematized *tables*. Tables look like relational-database tables, with rows, columns, and versioned values. We will not go into detail about the query language for Spanner. It looks like SQL with some extensions to support protocol-buffer-valued fields.

Spanner's data model is not purely relational, in that rows must have names. More precisely, every table is required to have an ordered set of one or more primary-key columns. This requirement is where Spanner still looks like a key-value store: the primary keys form the name for a row, and each table defines a mapping from the primary-key columns to the non-primary-key columns. A row has existence only if some value (even if it is NULL) is defined for the row's keys. Imposing this structure is useful because it lets applications control data locality through their choices of keys.

Figure 4 contains an example Spanner schema for storing photo metadata on a per-user, per-album basis. The schema language is similar to Megastore's, with the additional requirement that every Spanner database must be partitioned by clients into one or more hierarchies of tables. Client applications declare the hierarchies in database schemas via the `INTERLEAVE IN` declarations. The table at the top of a hierarchy is a *directory table*. Each row in a directory table with key *K*, together with all of the rows in descendant tables that start with *K* in lexicographic order, forms a directory. `ON DELETE CASCADE` says that deleting a row in the directory table deletes any associated child rows. The figure also illustrates the interleaved layout for the example database: for

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

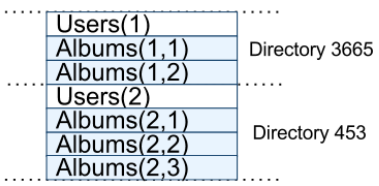



Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

example, `Albums(2,1)` represents the row from the `Albums` table for `user_id 2, album_id 1`. This interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables, which is necessary for good performance in a sharded, distributed database. Without it, Spanner would not know the most important locality relationships.

## 3 TrueTime

| Method | Returns |
|---|---|
| *TT.now*() | *TTinterval*: [*earliest*, *latest*] |
| *TT.after*($t$) | true if $t$ has definitely passed |
| *TT.before*($t$) | true if $t$ has definitely not arrived |

Table 1: TrueTime API. The argument $t$ is of type *TTstamp*.

This section describes the TrueTime API and sketches its implementation. We leave most of the details for another paper: our goal is to demonstrate the power of having such an API. Table 1 lists the methods of the API. TrueTime explicitly represents time as a *TTinterval*, which is an interval with bounded time uncertainty (unlike standard time interfaces that give clients no notion of uncertainty). The endpoints of a *TTinterval* are of type *TTstamp*. The *TT.now*() method returns a *TTinterval* that is guaranteed to contain the absolute time during which *TT.now*() was invoked. The time epoch is analogous to UNIX time with leap-second smearing. Define the instantaneous error bound as $\epsilon$, which is half of the interval's width, and the average error bound as $\bar{\epsilon}$. The *TT.after*() and *TT.before*() methods are convenience wrappers around *TT.now*().

Denote the absolute time of an event $e$ by the function $t_{abs}(e)$. In more formal terms, TrueTime guarantees that for an invocation $tt = TT.now()$, $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$, where $e_{now}$ is the invocation event.

The underlying time references used by TrueTime are GPS and atomic clocks. TrueTime uses two forms of time reference because they have different failure modes. GPS reference-source vulnerabilities include antenna and receiver failures, local radio interference, correlated failures (e.g., design faults such as incorrect leap-second handling and spoofing), and GPS system outages. Atomic clocks can fail in ways uncorrelated to GPS and each other, and over long periods of time can drift significantly due to frequency error.

TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine. The majority of masters have GPS receivers with dedicated antennas; these masters are separated physically to reduce the effects of antenna failures, radio interference, and spoofing. The remaining masters (which we refer to as *Armageddon masters*) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master. All masters' time references are regularly compared against each other. Each master also cross-checks the rate at which its reference advances time against its own local clock, and evicts itself if there is substantial divergence. Between synchronizations, Armageddon masters advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift. GPS masters advertise uncertainty that is typically close to zero.

Every daemon polls a variety of masters [29] to reduce vulnerability to errors from any one master. Some are GPS masters chosen from nearby datacenters; the rest are GPS masters from farther datacenters, as well as some Armageddon masters. Daemons apply a variant of Marzullo's algorithm [27] to detect and reject liars, and synchronize the local machine clocks to the non-liars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted.

Between synchronizations, a daemon advertises a slowly increasing time uncertainty. $\epsilon$ is derived from conservatively applied worst-case local clock drift. $\epsilon$ also depends on time-master uncertainty and communication delay to the time masters. In our production environment, $\epsilon$ is typically a sawtooth function of time, varying from about 1 to 7 ms over each poll interval. $\bar{\epsilon}$ is therefore 4 ms most of the time. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at 200 microseconds/second, which together account

| Operation | Timestamp Discussion | Concurrency Control | Replica Required |
|---|---|---|---|
| Read-Write Transaction | § 4.1.2 | pessimistic | leader |
| Read-Only Transaction | § 4.1.4 | lock-free | leader for timestamp; any for read, subject to § 4.1.3 |
| Snapshot Read, client-provided timestamp | — | lock-free | any, subject to § 4.1.3 |
| Snapshot Read, client-provided bound | § 4.1.3 | lock-free | any, subject to § 4.1.3 |

Table 2: Types of reads and writes in Spanner, and how they compare.

for the sawtooth bounds from 0 to 6 ms. The remaining 1 ms comes from the communication delay to the time masters. Excursions from this sawtooth are possible in the presence of failures. For example, occasional time-master unavailability can cause datacenter-wide increases in $\epsilon$. Similarly, overloaded machines and network links can result in occasional localized $\epsilon$ spikes.

# 4 Concurrency Control

This section describes how TrueTime is used to guarantee the correctness properties around concurrency control, and how those properties are used to implement features such as externally consistent transactions, lock-free read-only transactions, and non-blocking reads in the past. These features enable, for example, the guarantee that a whole-database audit read at a timestamp $t$ will see exactly the effects of every transaction that has committed as of $t$.

Going forward, it will be important to distinguish writes as seen by Paxos (which we will refer to as *Paxos writes* unless the context is clear) from Spanner client writes. For example, two-phase commit generates a Paxos write for the prepare phase that has no corresponding Spanner client write.

## 4.1 Timestamp Management

Table 2 lists the types of operations that Spanner supports. The Spanner implementation supports *read-write transactions*, *read-only transactions* (predeclared snapshot-isolation transactions), and *snapshot reads*. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as read-only transactions. Both are internally retried (clients need not write their own retry loops).

A read-only transaction is a kind of transaction that has the performance benefits of snapshot isolation [6]. A read-only transaction must be predeclared as not having any writes; it is not simply a read-write transaction without any writes. Reads in a read-only transaction execute at a system-chosen timestamp without locking, so that incoming writes are not blocked. The execution of the reads in a read-only transaction can proceed on any replica that is sufficiently up-to-date (Section 4.1.3).

A snapshot read is a read in the past that executes without locking. A client can either specify a timestamp for a snapshot read, or provide an upper bound on the desired timestamp's staleness and let Spanner choose a timestamp. In either case, the execution of a snapshot read proceeds at any replica that is sufficiently up-to-date.

For both read-only transactions and snapshot reads, commit is inevitable once a timestamp has been chosen, unless the data at that timestamp has been garbage-collected. As a result, clients can avoid buffering results inside a retry loop. When a server fails, clients can internally continue the query on a different server by repeating the timestamp and the current read position.

### 4.1.1 Paxos Leader Leases

Spanner's Paxos implementation uses timed leases to make leadership long-lived (10 seconds by default). A potential leader sends requests for timed *lease votes*; upon receiving a quorum of lease votes the leader knows it has a lease. A replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration. Define a leader's *lease interval* as starting when it discovers it has a quorum of lease votes, and as ending when it no longer has a quorum of lease votes (because some have expired). Spanner depends on the following disjointness invariant: for each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's. Appendix A describes how this invariant is enforced.

The Spanner implementation permits a Paxos leader to abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner constrains when abdication is permissible. Define $s_{max}$ to be the maximum timestamp used by a leader. Subsequent sections will describe when $s_{max}$ is advanced. Before abdicating, a leader must wait until $TT.after(s_{max})$ is true.

### 4.1.2 Assigning Timestamps to RW Transactions

Transactional reads and writes use two-phase locking. As a result, they can be assigned timestamps at any time

when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit.

Spanner depends on the following monotonicity invariant: within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders. A single leader replica can trivially assign timestamps in monotonically increasing order. This invariant is enforced across leaders by making use of the disjointness invariant: a leader must only assign timestamps within the interval of its leader lease. Note that whenever a timestamp $s$ is assigned, $s_{max}$ is advanced to $s$ to preserve disjointness.

Spanner also enforces the following external-consistency invariant: if the start of a transaction $T_2$ occurs after the commit of a transaction $T_1$, then the commit timestamp of $T_2$ must be greater than the commit timestamp of $T_1$. Define the start and commit events for a transaction $T_i$ by $e_i^{start}$ and $e_i^{commit}$; and the commit timestamp of a transaction $T_i$ by $s_i$. The invariant becomes $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$. The protocol for executing transactions and assigning timestamps obeys two rules, which together guarantee this invariant, as shown below. Define the arrival event of the commit request at the coordinator leader for a write $T_i$ to be $e_i^{server}$.

**Start** The coordinator leader for a write $T_i$ assigns a commit timestamp $s_i$ no less than the value of *TT.now*().*latest*, computed after $e_i^{server}$. Note that the participant leaders do not matter here; Section 4.2.1 describes how they are involved in the implementation of the next rule.

**Commit Wait** The coordinator leader ensures that clients cannot see any data committed by $T_i$ until *TT.after*($s_i$) is true. Commit wait ensures that $s_i$ is less than the absolute commit time of $T_i$, or $s_i < t_{abs}(e_i^{commit})$. The implementation of commit wait is described in Section 4.2.1. Proof:

$$
\begin{aligned}
s_1 &< t_{abs}(e_1^{commit}) && \text{(commit wait)} \\
t_{abs}(e_1^{commit}) &< t_{abs}(e_2^{start}) && \text{(assumption)} \\
t_{abs}(e_2^{start}) &\le t_{abs}(e_2^{server}) && \text{(causality)} \\
t_{abs}(e_2^{server}) &\le s_2 && \text{(start)} \\
s_1 &< s_2 && \text{(transitivity)}
\end{aligned}
$$

#### 4.1.3 Serving Reads at a Timestamp

The monotonicity invariant described in Section 4.1.2 allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read. Every replica tracks a value called *safe time* $t_{safe}$ which is the maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp $t$ if $t <= t_{safe}$.

Define $t_{safe} = min(t_{safe}^{Paxos}, t_{safe}^{TM})$, where each Paxos state machine has a safe time $t_{safe}^{Paxos}$ and each transaction manager has a safe time $t_{safe}^{TM}$. $t_{safe}^{Paxos}$ is simpler: it is the timestamp of the highest-applied Paxos write. Because timestamps increase monotonically and writes are applied in order, writes will no longer occur at or below $t_{safe}^{Paxos}$ with respect to Paxos.

$t_{safe}^{TM}$ is $\infty$ at a replica if there are zero prepared (but not committed) transactions—that is, transactions in between the two phases of two-phase commit. (For a participant slave, $t_{safe}^{TM}$ actually refers to the replica's leader's transaction manager, whose state the slave can infer through metadata passed on Paxos writes.) If there are any such transactions, then the state affected by those transactions is indeterminate: a participant replica does not know yet whether such transactions will commit. As we discuss in Section 4.2.1, the commit protocol ensures that every participant knows a lower bound on a prepared transaction's timestamp. Every participant leader (for a group $g$) for a transaction $T_i$ assigns a prepare timestamp $s_{i,g}^{prepare}$ to its prepare record. The coordinator leader ensures that the transaction's commit timestamp $s_i >= s_{i,g}^{prepare}$ over all participant groups $g$. Therefore, for every replica in a group $g$, over all transactions $T_i$ prepared at $g$, $t_{safe}^{TM} = min_i(s_{i,g}^{prepare}) - 1$ over all transactions prepared at $g$.

#### 4.1.4 Assigning Timestamps to RO Transactions

A read-only transaction executes in two phases: assign a timestamp $s_{read}$ [8], and then execute the transaction's reads as snapshot reads at $s_{read}$. The snapshot reads can execute at any replicas that are sufficiently up-to-date.

The simple assignment of $s_{read} = TT.now().latest$, at any time after a transaction starts, preserves external consistency by an argument analogous to that presented for writes in Section 4.1.2. However, such a timestamp may require the execution of the data reads at $s_{read}$ to block if $t_{safe}$ has not advanced sufficiently. (In addition, note that choosing a value of $s_{read}$ may also advance $s_{max}$ to preserve disjointness.) To reduce the chances of blocking, Spanner should assign the oldest timestamp that preserves external consistency. Section 4.2.2 explains how such a timestamp can be chosen.

### 4.2 Details

This section explains some of the practical details of read-write transactions and read-only transactions elided earlier, as well as the implementation of a special transaction type used to implement atomic schema changes.

It then describes some refinements of the basic schemes as described.

#### 4.2.1 Read-Write Transactions

Like Bigtable, writes that occur in a transaction are buffered at the client until commit. As a result, reads in a transaction do not see the effects of the transaction's writes. This design works well in Spanner because a read returns the timestamps of any data read, and uncommitted writes have not yet been assigned timestamps.

Reads within read-write transactions use wound-wait [33] to avoid deadlocks. The client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data. While a client transaction remains open, it sends keepalive messages to prevent participant leaders from timing out its transaction. When a client has completed all reads and buffered all writes, it begins two-phase commit. The client chooses a coordinator group and sends a commit message to each participant's leader with the identity of the coordinator and any buffered writes. Having the client drive two-phase commit avoids sending data twice across wide-area links.

A non-coordinator-participant leader first acquires write locks. It then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions (to preserve monotonicity), and logs a prepare record through Paxos. Each participant then notifies the coordinator of its prepare timestamp.

The coordinator leader also first acquires write locks, but skips the prepare phase. It chooses a timestamp for the entire transaction after hearing from all other participant leaders. The commit timestamp $s$ must be greater or equal to all prepare timestamps (to satisfy the constraints discussed in Section 4.1.3), greater than $TT.now().latest$ at the time the coordinator received its commit message, and greater than any timestamps the leader has assigned to previous transactions (again, to preserve monotonicity). The coordinator leader then logs a commit record through Paxos (or an abort if it timed out while waiting on the other participants).

Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until $TT.after(s)$, so as to obey the commit-wait rule described in Section 4.1.2. Because the coordinator leader chose $s$ based on $TT.now().latest$, and now waits until that timestamp is guaranteed to be in the past, the expected wait is at least $2 * \bar{\epsilon}$. This wait is typically overlapped with Paxos communication. After commit wait, the coordinator sends the commit timestamp to the client and all other participant leaders. Each participant leader logs the transaction's outcome through Paxos. All participants apply at the same timestamp and then release locks.

#### 4.2.2 Read-Only Transactions

Assigning a timestamp requires a negotiation phase between all of the Paxos groups that are involved in the reads. As a result, Spanner requires a *scope* expression for every read-only transaction, which is an expression that summarizes the keys that will be read by the entire transaction. Spanner automatically infers the scope for standalone queries.

If the scope's values are served by a single Paxos group, then the client issues the read-only transaction to that group's leader. (The current Spanner implementation only chooses a timestamp for a read-only transaction at a Paxos leader.) That leader assigns $s_{read}$ and executes the read. For a single-site read, Spanner generally does better than $TT.now().latest$. Define $LastTS()$ to be the timestamp of the last committed write at a Paxos group. If there are no prepared transactions, the assignment $s_{read} = LastTS()$ trivially satisfies external consistency: the transaction will see the result of the last write, and therefore be ordered after it.

If the scope's values are served by multiple Paxos groups, there are several options. The most complicated option is to do a round of communication with all of the groups's leaders to negotiate $s_{read}$ based on $LastTS()$. Spanner currently implements a simpler choice. The client avoids a negotiation round, and just has its reads execute at $s_{read} = TT.now().latest$ (which may wait for safe time to advance). All reads in the transaction can be sent to replicas that are sufficiently up-to-date.

#### 4.2.3 Schema-Change Transactions

TrueTime enables Spanner to support atomic schema changes. It would be infeasible to use a standard transaction, because the number of participants (the number of groups in a database) could be in the millions. Bigtable supports atomic schema changes in one datacenter, but its schema changes block all operations.

A Spanner schema-change transaction is a generally non-blocking variant of a standard transaction. First, it is explicitly assigned a timestamp in the future, which is registered in the prepare phase. As a result, schema changes across thousands of servers can complete with minimal disruption to other concurrent activity. Second, reads and writes, which implicitly depend on the schema, synchronize with any registered schema-change timestamp at time $t$: they may proceed if their timestamps precede $t$, but they must block behind the schema-change transaction if their timestamps are after $t$. Without TrueTime, defining the schema change to happen at $t$ would be meaningless.

| replicas | latency (ms) | | | throughput (Kops/sec) | | |
|---|---|---|---|---|---|---|
| | write | read-only transaction | snapshot read | write | read-only transaction | snapshot read |
| 1D | 9.4±.6 | — | — | 4.0±.3 | — | — |
| 1 | 14.4±1.0 | 1.4±.1 | 1.3±.1 | 4.1±.05 | 10.9±.4 | 13.5±.1 |
| 3 | 13.9±.6 | 1.3±.1 | 1.2±.1 | 2.2±.5 | 13.8±3.2 | 38.5±.3 |
| 5 | 14.4±.4 | 1.4±.05 | 1.3±.04 | 2.8±.3 | 25.3±5.2 | 50.0±1.1 |

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

#### 4.2.4 Refinements

$t_{safe}^{TM}$ as defined above has a weakness, in that a single prepared transaction prevents $t_{safe}$ from advancing. As a result, no reads can occur at later timestamps, even if the reads do not conflict with the transaction. Such false conflicts can be removed by augmenting $t_{safe}^{TM}$ with a fine-grained mapping from key ranges to prepared-transaction timestamps. This information can be stored in the lock table, which already maps key ranges to lock metadata. When a read arrives, it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts.

*LastTS*() as defined above has a similar weakness: if a transaction has just committed, a non-conflicting read-only transaction must still be assigned $s_{read}$ so as to follow that transaction. As a result, the execution of the read could be delayed. This weakness can be remedied similarly by augmenting *LastTS*() with a fine-grained mapping from key ranges to commit timestamps in the lock table. (We have not yet implemented this optimization.) When a read-only transaction arrives, its timestamp can be assigned by taking the maximum value of *LastTS*() for the key ranges with which the transaction conflicts, unless there is a conflicting prepared transaction (which can be determined from fine-grained safe time).

$t_{safe}^{Paxos}$ as defined above has a weakness in that it cannot advance in the absence of Paxos writes. That is, a snapshot read at $t$ cannot execute at Paxos groups whose last write happened before $t$. Spanner addresses this problem by taking advantage of the disjointness of leader-lease intervals. Each Paxos leader advances $t_{safe}^{Paxos}$ by keeping a threshold above which future writes' timestamps will occur: it maintains a mapping *MinNextTS*$(n)$ from Paxos sequence number $n$ to the minimum timestamp that may be assigned to Paxos sequence number $n + 1$. A replica can advance $t_{safe}^{Paxos}$ to *MinNextTS*$(n) - 1$ when it has applied through $n$.

A single leader can enforce its *MinNextTS*() promises easily. Because the timestamps promised by *MinNextTS*() lie within a leader's lease, the disjointness invariant enforces *MinNextTS*() promises across leaders. If a leader wishes to advance *MinNextTS*() beyond the end of its leader lease, it must first extend its lease. Note that $s_{max}$ is always advanced to the highest value in *MinNextTS*() to preserve disjointness.

A leader by default advances *MinNextTS*() values every 8 seconds. Thus, in the absence of prepared transactions, healthy slaves in an idle Paxos group can serve reads at timestamps greater than 8 seconds old in the worst case. A leader may also advance *MinNextTS*() values on demand from slaves.

## 5 Evaluation

We first measure Spanner's performance with respect to replication, transactions, and availability. We then provide some data on TrueTime behavior, and a case study of our first client, F1.

### 5.1 Microbenchmarks

Table 3 presents some microbenchmarks for Spanner. These measurements were taken on timeshared machines: each spanserver ran on scheduling units of 4GB RAM and 4 cores (AMD Barcelona 2200MHz). Clients were run on separate machines. Each zone contained one spanserver. Clients and zones were placed in a set of datacenters with network distance of less than 1ms. (Such a layout should be commonplace: most applications do not need to distribute all of their data worldwide.) The test database was created with 50 Paxos groups with 2500 directories. Operations were standalone reads and writes of 4KB. All reads were served out of memory after a compaction, so that we are only measuring the overhead of Spanner's call stack. In addition, one unmeasured round of reads was done first to warm any location caches.

For the latency experiments, clients issued sufficiently few operations so as to avoid queuing at the servers. From the 1-replica experiments, commit wait is about 5ms, and Paxos latency is about 9ms. As the number of replicas increases, the latency stays roughly constant with less standard deviation because Paxos executes in parallel at a group's replicas. As the number of replicas increases, the latency to achieve a quorum becomes less sensitive to slowness at one slave replica.

For the throughput experiments, clients issued sufficiently many operations so as to saturate the servers'

| participants | latency (ms) | |
|---|---|---|
| | mean | 99th percentile |
| 1 | 17.0 ±1.4 | 75.0 ±34.9 |
| 2 | 24.5 ±2.5 | 87.6 ±35.9 |
| 5 | 31.5 ±6.2 | 104.5 ±52.2 |
| 10 | 30.0 ±3.7 | 95.6 ±25.4 |
| 25 | 35.5 ±5.6 | 100.4 ±42.7 |
| 50 | 42.7 ±4.1 | 93.7 ±22.9 |
| 100 | 71.4 ±7.6 | 131.2 ±17.6 |
| 200 | 150.5 ±11.0 | 320.3 ±35.1 |

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

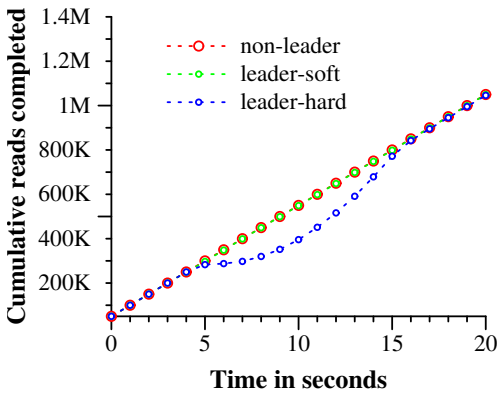



Figure 5: Effect of killing servers on throughput.

CPUs. Snapshot reads can execute at any up-to-date replicas, so their throughput increases almost linearly with the number of replicas. Single-read read-only transactions only execute at leaders because timestamp assignment must happen at leaders. Read-only-transaction throughput increases with the number of replicas because the number of effective spanservers increases: in the experimental setup, the number of spanservers equaled the number of replicas, and leaders were randomly distributed among the zones. Write throughput benefits from the same experimental artifact (which explains the increase in throughput from 3 to 5 replicas), but that benefit is outweighed by the linear increase in the amount of work performed per write, as the number of replicas increases.

Table 4 demonstrates that two-phase commit can scale to a reasonable number of participants: it summarizes a set of experiments run across 3 zones, each with 25 spanservers. Scaling up to 50 participants is reasonable in both mean and 99th-percentile, and latencies start to rise noticeably at 100 participants.

## 5.2 Availability

Figure 5 illustrates the availability benefits of running Spanner in multiple datacenters. It shows the results of three experiments on throughput in the presence of datacenter failure, all of which are overlaid onto the same time scale. The test universe consisted of 5 zones $Z_i$, each of which had 25 spanservers. The test database was sharded into 1250 Paxos groups, and 100 test clients constantly issued non-snapshot reads at an aggregrate rate of 50K reads/second. All of the leaders were explicitly placed in $Z_1$. Five seconds into each test, all of the servers in one zone were killed: *non-leader* kills $Z_2$; *leader-hard* kills $Z_1$; *leader-soft* kills $Z_1$, but it gives notifications to all of the servers that they should handoff leadership first.

Killing $Z_2$ has no effect on read throughput. Killing $Z_1$ while giving the leaders time to handoff leadership to a different zone has a minor effect: the throughput drop is not visible in the graph, but is around 3-4%. On the other hand, killing $Z_1$ with no warning has a severe effect: the rate of completion drops almost to 0. As leaders get re-elected, though, the throughput of the system rises to approximately 100K reads/second because of two artifacts of our experiment: there is extra capacity in the system, and operations are queued while the leader is unavailable. As a result, the throughput of the system rises before leveling off again at its steady-state rate.

We can also see the effect of the fact that Paxos leader leases are set to 10 seconds. When we kill the zone, the leader-lease expiration times for the groups should be evenly distributed over the next 10 seconds. Soon after each lease from a dead leader expires, a new leader is elected. Approximately 10 seconds after the kill time, all of the groups have leaders and throughput has recovered. Shorter lease times would reduce the effect of server deaths on availability, but would require greater amounts of lease-renewal network traffic. We are in the process of designing and implementing a mechanism that will cause slaves to release Paxos leader leases upon leader failure.

## 5.3 TrueTime

Two questions must be answered with respect to TrueTime: is $\epsilon$ truly a bound on clock uncertainty, and how bad does $\epsilon$ get? For the former, the most serious problem would be if a local clock's drift were greater than 200us/sec: that would break assumptions made by TrueTime. Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems. As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends.

Figure 6 presents TrueTime data taken at several thousand spanserver machines across datacenters up to 2200

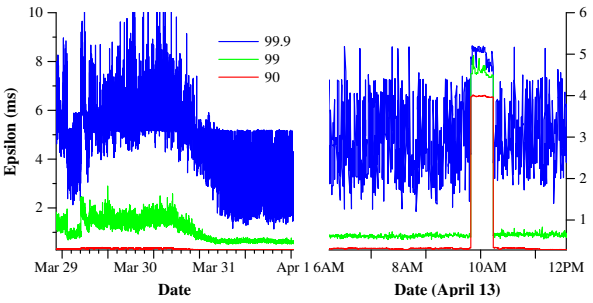



Figure 6: Distribution of TrueTime $\epsilon$ values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

km apart. It plots the 90th, 99th, and 99.9th percentiles of $\epsilon$, sampled at timeslave daemons immediately after polling the time masters. This sampling elides the sawtooth in $\epsilon$ due to local-clock uncertainty, and therefore measures time-master uncertainty (which is generally 0) plus communication delay to the time masters.

The data shows that these two factors in determining the base value of $\epsilon$ are generally not a problem. However, there can be significant tail-latency issues that cause higher values of $\epsilon$. The reduction in tail latencies beginning on March 30 were due to networking improvements that reduced transient network-link congestion. The increase in $\epsilon$ on April 13, approximately one hour in duration, resulted from the shutdown of 2 time masters at a datacenter for routine maintenance. We continue to investigate and remove causes of TrueTime spikes.

## 5.4 F1

Spanner started being experimentally evaluated under production workloads in early 2011, as part of a rewrite of Google's advertising backend called F1 [35]. This backend was originally based on a MySQL database that was manually sharded many ways. The uncompressed dataset is tens of terabytes, which is small compared to many NoSQL instances, but was large enough to cause difficulties with sharded MySQL. The MySQL sharding scheme assigned each customer and all related data to a fixed shard. This layout enabled the use of indexes and complex query processing on a per-customer basis, but required some knowledge of the sharding in application business logic. Resharding this revenue-critical database as it grew in the number of customers and their data was extremely costly. The last resharding took over two years of intense effort, and involved coordination and testing across dozens of teams to minimize risk. This operation was too complex to do regularly: as a result, the team had to limit growth on the MySQL database by storing some data in external Bigtables, which compromised transactional behavior and the ability to query across all data.

| # fragments | # directories |
|---|---|
| 1 | >100M |
| 2–4 | 341 |
| 5–9 | 5336 |
| 10–14 | 232 |
| 15–99 | 34 |
| 100–500 | 7 |

Table 5: Distribution of directory-fragment counts in F1.

The F1 team chose to use Spanner for several reasons. First, Spanner removes the need to manually reshard. Second, Spanner provides synchronous replication and automatic failover. With MySQL master-slave replication, failover was difficult, and risked data loss and downtime. Third, F1 requires strong transactional semantics, which made using other NoSQL systems impractical. Application semantics requires transactions across arbitrary data, and consistent reads. The F1 team also needed secondary indexes on their data (since Spanner does not yet provide automatic support for secondary indexes), and was able to implement their own consistent global indexes using Spanner transactions.

All application writes are now by default sent through F1 to Spanner, instead of the MySQL-based application stack. F1 has 2 replicas on the west coast of the US, and 3 on the east coast. This choice of replica sites was made to cope with outages due to potential major natural disasters, and also the choice of their frontend sites. Anecdotally, Spanner's automatic failover has been nearly invisible to them. Although there have been unplanned cluster failures in the last few months, the most that the F1 team has had to do is update their database's schema to tell Spanner where to preferentially place Paxos leaders, so as to keep them close to where their frontends moved.

Spanner's timestamp semantics made it efficient for F1 to maintain in-memory data structures computed from the database state. F1 maintains a logical history log of all changes, which is written into Spanner itself as part of every transaction. F1 takes full snapshots of data at a timestamp to initialize its data structures, and then reads incremental changes to update them.

Table 5 illustrates the distribution of the number of fragments per directory in F1. Each directory typically corresponds to a customer in the application stack above F1. The vast majority of directories (and therefore customers) consist of only 1 fragment, which means that reads and writes to those customers' data are guaranteed to occur on only a single server. The directories with more than 100 fragments are all tables that contain F1 secondary indexes: writes to more than a few fragments

| operation | latency (ms) | | count |
|---|---|---|---|
| | mean | std dev | |
| all reads | 8.7 | 376.4 | 21.5B |
| single-site commit | 72.3 | 112.8 | 31.2M |
| multi-site commit | 103.0 | 52.2 | 32.1M |

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

of such tables are extremely uncommon. The F1 team has only seen such behavior when they do untuned bulk data loads as transactions.

Table 6 presents Spanner operation latencies as measured from F1 servers. Replicas in the east-coast data centers are given higher priority in choosing Paxos leaders. The data in the table is measured from F1 servers in those data centers. The large standard deviation in write latencies is caused by a pretty fat tail due to lock conflicts. The even larger standard deviation in read latencies is partially due to the fact that Paxos leaders are spread across two data centers, only one of which has machines with SSDs. In addition, the measurement includes every read in the system from two datacenters: the mean and standard deviation of the bytes read were roughly 1.6KB and 119KB, respectively.

## 6 Related Work

Consistent replication across datacenters as a storage service has been provided by Megastore [5] and DynamoDB [3]. DynamoDB presents a key-value interface, and only replicates within a region. Spanner follows Megastore in providing a semi-relational data model, and even a similar schema language. Megastore does not achieve high performance. It is layered on top of Bigtable, which imposes high communication costs. It also does not support long-lived leaders: multiple replicas may initiate writes. All writes from different replicas necessarily conflict in the Paxos protocol, even if they do not logically conflict: throughput collapses on a Paxos group at several writes per second. Spanner provides higher performance, general-purpose transactions, and external consistency.

Pavlo et al. [31] have compared the performance of databases and MapReduce [12]. They point to several other efforts that have been made to explore database functionality layered on distributed key-value stores [1, 4, 7, 41] as evidence that the two worlds are converging. We agree with the conclusion, but demonstrate that integrating multiple layers has its advantages: integrating concurrency control with replication reduces the cost of commit wait in Spanner, for example.

The notion of layering transactions on top of a replicated store dates at least as far back as Gifford's dissertation [16]. Scatter [17] is a recent DHT-based key-value store that layers transactions on top of consistent replication. Spanner focuses on providing a higher-level interface than Scatter does. Gray and Lamport [18] describe a non-blocking commit protocol based on Paxos. Their protocol incurs more messaging costs than two-phase commit, which would aggravate the cost of commit over widely distributed groups. Walter [36] provides a variant of snapshot isolation that works within, but not across datacenters. In contrast, our read-only transactions provide a more natural semantics, because we support external consistency over all operations.

There has been a spate of recent work on reducing or eliminating locking overheads. Calvin [40] eliminates concurrency control: it pre-assigns timestamps and then executes the transactions in timestamp order. H-Store [39] and Granola [11] each supported their own classification of transaction types, some of which could avoid locking. None of these systems provides external consistency. Spanner addresses the contention issue by providing support for snapshot isolation.

VoltDB [42] is a sharded in-memory database that supports master-slave replication over the wide area for disaster recovery, but not more general replication configurations. It is an example of what has been called NewSQL, which is a marketplace push to support scalable SQL [38]. A number of commercial databases implement reads in the past, such as MarkLogic [26] and Oracle's Total Recall [30]. Lomet and Li [24] describe an implementation strategy for such a temporal database.

Farsite derived bounds on clock uncertainty (much looser than TrueTime's) relative to a trusted clock reference [13]: server leases in Farsite were maintained in the same way that Spanner maintains Paxos leases. Loosely synchronized clocks have been used for concurrency-control purposes in prior work [2, 23]. We have shown that TrueTime lets one reason about global time across sets of Paxos state machines.

## 7 Future Work

We have spent most of the last year working with the F1 team to transition Google's advertising backend from MySQL to Spanner. We are actively improving its monitoring and support tools, as well as tuning its performance. In addition, we have been working on improving the functionality and performance of our backup/restore system. We are currently implementing the Spanner schema language, automatic maintenance of secondary indices, and automatic load-based resharding. Longer term, there are a couple of features that we plan to in-

vestigate. Optimistically doing reads in parallel may be a valuable strategy to pursue, but initial experiments have indicated that the right implementation is non-trivial. In addition, we plan to eventually support direct changes of Paxos configurations [22, 34].

Given that we expect many applications to replicate their data across datacenters that are relatively close to each other, TrueTime $\epsilon$ may noticeably affect performance. We see no insurmountable obstacle to reducing $\epsilon$ below 1ms. Time-master-query intervals can be reduced, and better clock crystals are relatively cheap. Time-master query latency could be reduced with improved networking technology, or possibly even avoided through alternate time-distribution technology.

Finally, there are obvious areas for improvement. Although Spanner is scalable in the number of nodes, the node-local data structures have relatively poor performance on complex SQL queries, because they were designed for simple key-value accesses. Algorithms and data structures from DB literature could improve single-node performance a great deal. Second, moving data automatically between datacenters in response to changes in client load has long been a goal of ours, but to make that goal effective, we would also need the ability to move client-application processes between datacenters in an automated, coordinated fashion. Moving processes raises the even more difficult problem of managing resource acquisition and allocation between datacenters.

## 8 Conclusions

To summarize, Spanner combines and extends on ideas from two research communities: from the database community, a familiar, easy-to-use, semi-relational interface, transactions, and an SQL-based query language; from the systems community, scalability, automatic sharding, fault tolerance, consistent replication, external consistency, and wide-area distribution. Since Spanner's inception, we have taken more than 5 years to iterate to the current design and implementation. Part of this long iteration phase was due to a slow realization that Spanner should do more than tackle the problem of a globally-replicated namespace, and should also focus on database features that Bigtable was missing.

One aspect of our design stands out: the linchpin of Spanner's feature set is TrueTime. We have shown that reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics. In addition, as the underlying system enforces tighter bounds on clock uncertainty, the overhead of the stronger semantics decreases. As a community, we should no longer depend on loosely synchronized clocks and weak time APIs in designing distributed algorithms.

## Acknowledgements

Many people have helped to improve this paper: our shepherd Jon Howell, who went above and beyond his responsibilities; the anonymous referees; and many Googlers: Atul Adya, Fay Chang, Frank Dabek, Sean Dorward, Bob Gruber, David Held, Nick Kline, Alex Thomson, and Joel Wein. Our management has been very supportive of both our work and of publishing this paper: Aristotle Balogh, Bill Coughran, Urs Hölzle, Doron Meyer, Cos Nicolaou, Kathy Polizzi, Sridhar Ramaswany, and Shivakumar Venkataraman.

We have built upon the work of the Bigtable and Megastore teams. The F1 team, and Jeff Shute in particular, worked closely with us in developing our data model and helped immensely in tracking down performance and correctness bugs. The Platforms team, and Luiz Barroso and Bob Felderman in particular, helped to make TrueTime happen. Finally, a lot of Googlers used to be on our team: Ken Ashcraft, Paul Cychosz, Krzysztof Ostrowski, Amir Voskoboynik, Matthew Weaver, Theo Vassilakis, and Eric Veach; or have joined our team recently: Nathan Bales, Adam Beberg, Vadim Borisov, Ken Chen, Brian Cooper, Cian Cullinan, Robert-Jan Huijsman, Milind Joshi, Andrey Khorlin, Dawid Kuroczko, Laramie Leavitt, Eric Li, Mike Mammarella, Sunil Mushran, Simon Nielsen, Ovidiu Platon, Ananth Shrinivas, Vadim Suvorov, and Marcel van der Holst.

## References

[1] Azza Abouzeid et al. "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads". *Proc. of VLDB*. 2009, pp. 922–933.

[2] A. Adya et al. "Efficient optimistic concurrency control using loosely synchronized clocks". *Proc. of SIGMOD*. 1995, pp. 23–34.

[3] Amazon. *Amazon DynamoDB*. 2012.

[4] Michael Armbrust et al. "PIQL: Success-Tolerant Query Processing in the Cloud". *Proc. of VLDB*. 2011, pp. 181–192.

[5] Jason Baker et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". *Proc. of CIDR*. 2011, pp. 223–234.

[6] Hal Berenson et al. "A critique of ANSI SQL isolation levels". *Proc. of SIGMOD*. 1995, pp. 1–10.

[7] Matthias Brantner et al. "Building a database on S3". *Proc. of SIGMOD*. 2008, pp. 251–264.

[8] A. Chan and R. Gray. "Implementing Distributed Read-Only Transactions". *IEEE TOSE* SE-11.2 (Feb. 1985), pp. 205–212.

[9] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". *ACM TOCS* 26.2 (June 2008), 4:1–4:26.

[10] Brian F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". *Proc. of VLDB*. 2008, pp. 1277–1288.

[11] James Cowling and Barbara Liskov. "Granola: Low-Overhead Distributed Transaction Coordination". *Proc. of USENIX ATC*. 2012, pp. 223–236.

[12] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: a flexible data processing tool". *CACM* 53.1 (Jan. 2010), pp. 72–77.

[13] John Douceur and Jon Howell. *Scalable Byzantine-Fault-Quantifying Clock Synchronization*. Tech. rep. MSR-TR-2003-67. MS Research, 2003.

[14] John R. Douceur and Jon Howell. "Distributed directory service in the Farsite file system". *Proc. of OSDI*. 2006, pp. 321–334.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". *Proc. of SOSP*. Dec. 2003, pp. 29–43.

[16] David K. Gifford. *Information Storage in a Decentralized Computer System*. Tech. rep. CSL-81-8. PhD dissertation. Xerox PARC, July 1982.

[17] Lisa Glendenning et al. "Scalable consistency in Scatter". *Proc. of SOSP*. 2011.

[18] Jim Gray and Leslie Lamport. "Consensus on transaction commit". *ACM TODS* 31.1 (Mar. 2006), pp. 133–160.

[19] Pat Helland. "Life beyond Distributed Transactions: an Apostate's Opinion". *Proc. of CIDR*. 2007, pp. 132–141.

[20] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". *ACM TOPLAS* 12.3 (July 1990), pp. 463–492.

[21] Leslie Lamport. "The part-time parliament". *ACM TOCS* 16.2 (May 1998), pp. 133–169.

[22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. "Reconfiguring a state machine". *SIGACT News* 41.1 (Mar. 2010), pp. 63–73.

[23] Barbara Liskov. "Practical uses of synchronized clocks in distributed systems". *Distrib. Comput.* 6.4 (July 1993), pp. 211–219.

[24] David B. Lomet and Feifei Li. "Improving Transaction-Time DBMS Performance and Functionality". *Proc. of ICDE* (2009), pp. 581–591.

[25] Jacob R. Lorch et al. "The SMART way to migrate replicated stateful services". *Proc. of EuroSys*. 2006, pp. 103–115.

[26] MarkLogic. *MarkLogic 5 Product Documentation*. 2012.

[27] Keith Marzullo and Susan Owicki. "Maintaining the time in a distributed system". *Proc. of PODC*. 1983, pp. 295–305.

[28] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". *Proc. of VLDB*. 2010, pp. 330–339.

[29] D.L. Mills. *Time synchronization in DCNET hosts*. Internet Project Report IEN–173. COMSAT Laboratories, Feb. 1981.

[30] Oracle. *Oracle Total Recall*. 2012.

[31] Andrew Pavlo et al. "A comparison of approaches to large-scale data analysis". *Proc. of SIGMOD*. 2009, pp. 165–178.

[32] Daniel Peng and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications". *Proc. of OSDI*. 2010, pp. 1–15.

[33] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. "System level concurrency control for distributed database systems". *ACM TODS* 3.2 (June 1978), pp. 178–198.

[34] Alexander Shraer et al. "Dynamic Reconfiguration of Primary/Backup Clusters". *Proc. of USENIX ATC*. 2012, pp. 425–438.

[35] Jeff Shute et al. "F1 — The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business". *Proc. of SIGMOD*. May 2012, pp. 777–778.

[36] Yair Sovran et al. "Transactional storage for geo-replicated systems". *Proc. of SOSP*. 2011, pp. 385–400.

[37] Michael Stonebraker. *Why Enterprises Are Uninterested in NoSQL*. 2010.

[38] Michael Stonebraker. *Six SQL Urban Myths*. 2010.

[39] Michael Stonebraker et al. "The end of an architectural era: (it's time for a complete rewrite)". *Proc. of VLDB*. 2007, pp. 1150–1160.

[40] Alexander Thomson et al. "Calvin: Fast Distributed Transactions for Partitioned Database Systems". *Proc. of SIGMOD*. 2012, pp. 1–12.

[41] Ashish Thusoo et al. "Hive — A Petabyte Scale Data Warehouse Using Hadoop". *Proc. of ICDE*. 2010, pp. 996–1005.

[42] VoltDB. *VoltDB Resources*. 2012.

# A Paxos Leader-Lease Management

The simplest means to ensure the disjointness of Paxos-leader-lease intervals would be for a leader to issue a synchronous Paxos write of the lease interval, whenever it would be extended. A subsequent leader would read the interval and wait until that interval has passed.

TrueTime can be used to ensure disjointness without these extra log writes. The potential $i$th leader keeps a lower bound on the start of a lease vote from replica $r$ as $v_{i,r}^{leader} = \mathit{TT.now}().\mathit{earliest}$, computed before $e_{i,r}^{send}$ (defined as when the lease request is sent by the leader). Each replica $r$ grants a lease at lease $e_{i,r}^{grant}$, which happens after $e_{i,r}^{receive}$ (when the replica receives a lease request); the lease ends at $t_{i,r}^{end} = \mathit{TT.now}().\mathit{latest} + 10$, computed after $e_{i,r}^{receive}$. A replica $r$ obeys the **single-vote** rule: it will not grant another lease vote until $\mathit{TT.after}(t_{i,r}^{end})$ is true. To enforce this rule across different incarnations of $r$, Spanner logs a lease vote at the granting replica before granting the lease; this log write can be piggybacked upon existing Paxos-protocol log writes.

When the $i$th leader receives a quorum of votes (event $e_i^{quorum}$), it computes its lease interval as $\mathit{lease}_i = [\mathit{TT.now}().\mathit{latest}, \mathit{min}_r(v_{i,r}^{leader}) + 10]$. The lease is deemed to have expired at the leader when $\mathit{TT.before}(\mathit{min}_r(v_{i,r}^{leader}) + 10)$ is false. To prove disjointness, we make use of the fact that the $i$th and $(i+1)$th leaders must have one replica in common in their quorums. Call that replica $r0$. Proof:

$$
\begin{aligned}
\mathit{lease}_i.\mathit{end} &= \mathit{min}_r(v_{i,r}^{leader}) + 10 && \text{(by definition)}\\
\mathit{min}_r(v_{i,r}^{leader}) + 10 &\le v_{i,r0}^{leader} + 10 && \text{(min)}\\
v_{i,r0}^{leader} + 10 &\le t_{abs}(e_{i,r0}^{send}) + 10 && \text{(by definition)}\\
t_{abs}(e_{i,r0}^{send}) + 10 &\le t_{abs}(e_{i,r0}^{receive}) + 10 && \text{(causality)}\\
t_{abs}(e_{i,r0}^{receive}) + 10 &\le t_{i,r0}^{end} && \text{(by definition)}\\
t_{i,r0}^{end} &< t_{abs}(e_{i+1,r0}^{grant}) && \text{(single-vote)}\\
t_{abs}(e_{i+1,r0}^{grant}) &\le t_{abs}(e_{i+1}^{quorum}) && \text{(causality)}\\
t_{abs}(e_{i+1}^{quorum}) &\le \mathit{lease}_{i+1}.\mathit{start} && \text{(by definition)}
\end{aligned}
$$

# Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary

Cheng Li[†], Daniel Porto[*†], Allen Clement[†], Johannes Gehrke[‡], Nuno Preguiça[*], Rodrigo Rodrigues[*]
[†]Max Planck Institute for Software Systems (MPI-SWS), [*]CITI / DI-FCT-Universidade Nova de Lisboa, [‡]Cornell University

**Abstract:** Online services distribute and replicate state across geographically diverse data centers and direct user requests to the closest or least loaded site. While effectively ensuring low latency responses, this approach is at odds with maintaining cross-site consistency. We make three contributions to address this tension. First, we propose RedBlue consistency, which enables blue operations to be fast (and eventually consistent) while the remaining red operations are strongly consistent (and slow). Second, to make use of fast operation whenever possible and only resort to strong consistency when needed, we identify conditions delineating when operations can be blue and must be red. Third, we introduce a method that increases the space of potential blue operations by breaking them into separate generator and shadow phases. We built a coordination infrastructure called Gemini that offers RedBlue consistency, and we report on our experience modifying the TPC-W and RUBiS benchmarks and an online social network to use Gemini. Our experimental results show that RedBlue consistency provides substantial performance gains without sacrificing consistency.

## 1 Introduction

Scaling services over the Internet to meet the needs of an ever-growing user base is challenging. In order to improve user-perceived latency, which directly affects the quality of the user experience [32], services replicate system state across geographically diverse sites and direct users to the closest or least loaded site.

To avoid paying the performance penalty of synchronizing concurrent actions across data centers, some systems, such as Amazon's Dynamo [9], resort to weaker consistency semantics like eventual consistency where state can temporarily diverge. Others, such as Yahoo!'s PNUTS [8], avoid state divergence by requiring all operations that update the service state to be funneled through a primary site and thus incurring increased latency.

This paper addresses the inherent tension between performance and meaningful consistency. A first step towards this goal is to allow multiple levels of consistency to coexist [19, 34, 35]: some operations can be executed optimistically, without synchronizing with concurrent actions at other sites, while others require a stronger consistency level and thus require cross-site synchronization. However, this places a high burden on the developer of the service, who must decide which operations to assign which consistency levels. This requires reasoning about the consistency semantics of the overall system to ensure that the behaviors that are allowed by the different consistency levels satisfy the specification of the system.

In this paper we make the following three contributions to address this tension.

1. We propose a novel consistency definition called RedBlue consistency. The intuition behind RedBlue consistency is that blue operations execute locally and are lazily replicated in an eventually consistent manner [9, 25, 38, 26, 12, 33, 34]. Red operations, in contrast, are serialized with respect to each other and require immediate cross-site coordination. RedBlue consistency preserves causality by ensuring that dependencies established when an operation is invoked at its primary site are preserved as the operation is incorporated at other sites.
2. We identify the conditions under which operations must be colored red and may be colored blue in order to ensure that application invariants are never violated and that all replicas converge on the same final state. Intuitively, operations that commute with all other operations and do not impact invariants may be blue.
3. We observe that the commutativity requirement limits the space of potentially blue operations. To address this, we decompose operations into two components: (1) a generator operation that identifies the changes the original operation should make, but has no side effects itself, and (2) a shadow operation that performs the identified changes and is replicated to all sites. Only shadow operations are colored red or blue. This allows for a fine-grained classification of operations and broadens the space of potentially blue operations.

We built a system called Gemini that coordinates RedBlue replication, and use it to extend three applications to be RedBlue consistent: the TPC-W and RUBiS benchmarks and the Quoddy social network. Our evaluation using microbenchmarks and the three applications shows that RedBlue consistency provides substantial latency and throughput benefits. Furthermore, our experience with modifying these applications indicates that shadow operations can be created with modest effort.

The rest of the paper is organized as follows: we po-

| Consistency level | Example systems | Immediate response | State convergence | Single value | General operations | Stable histories | Classification strategy |
|---|---|---|---|---|---|---|---|
| Strong | RSM [20, 31] | no | yes | yes | yes | yes | N/A |
| Timeline/snapshot | PNUTS [8], Megastore [3] | reads only | yes | yes | yes | yes | N/A |
| Fork | SUNDR [24] | all ops | no | yes | yes | yes | N/A |
| Eventual | Bayou [38], Depot [26] | all ops | yes | no | yes | yes | N/A |
| | Sporc [12], CRDT [33] | all ops | yes | yes | no | yes | N/A |
| | Zeno [34], COPS [25] | weak/all ops | yes | yes | yes | no | no / N/A |
| Multi | PSI [35] | cset | yes | yes | partial | yes | no |
| | lazy repl. [19], Horus [39] | immed./causal ops | yes | yes | yes | yes | no |
| RedBlue | Gemini | Blue ops | yes | yes | yes | yes | yes |

Table 1: Tradeoffs in geo-replicated systems and various consistency levels.

sition our work in comparison to existing proposals in §2. We define RedBlue consistency and introduce shadow operations along with a set of principles of how to use them in §4 and §5. We describe our prototype system in §6, and report on the experience transitioning three application benchmarks to be RedBlue consistent in §7. We analyze experimental results in §8 and conclude in §9.

## 2 Background and related work

**Target end-to-end properties.** To frame the discussion of existing systems that may be used for geo-replication, we start by informally stating some desirable properties that such solutions should support. The first property consists of ensuring a good user experience by providing **low latency** access to the service [32]. Providing low latency access implies that operations should proceed after contacting a small number of replicas, but this is at odds with other requirements that are often sacrificed by consistency models that privilege low latency. The first such requirement is preserving **causality**, both in terms of the monotonicity of user requests within a session and preserving causality across clients, which is key to enabling natural semantics [28]. Second, it is important that all replicas that have executed the same set of operations are in the same state, i.e., that they exhibit **state convergence**. Third, we want to avoid marked deviations from the conventional, single server semantics. In particular, operations should return a **single value**, precluding solutions that return a set of values corresponding to the outcome of multiple concurrent updates; the system should provide a set of **stable histories**, meaning that user actions cannot be undone; and it should provide support for **general operations**, not restricting the type of operations that can be executed. Fourth, the behavior of the system must obey its specification. This specification may be defined as a set of **invariants** that must be preserved, e.g., that no two users receive the same user id when registering. Finally, and orthogonally to the tension between low latency and user semantics, it is important for all operations executed at one replica to be propagated to all remaining replicas, a property we call **eventual propagation**.

Table 1 summarizes several proposals of consistency definitions, which strike different balances between the requirements mentioned above. While other consistency definitions exist, we focus on the ones most closely related to the problem of offering fast and consistent responses in geo-replicated systems.

**Strong vs. weak consistency.** On the strong consistency side of the spectrum there are definitions like linearizability [17], where the replicated system behaves like a single server that serializes all operations. This, however, requires coordination among replicas to agree on the order in which operations are executed, with the corresponding overheads that are amplified in geo-replication scenarios. Somewhat more efficient are timeline consistency in PNUTS [8] and snapshot consistency in Megastore [3]. These systems ensure that there is a total order for updates to the service state, but give the option of reading a consistent but dated view of the service. Similarly, Facebook has a primary site that handles updates and a secondary site that acts as a read-only copy [23]. This allows for fast reads executed at the closest site but writes still pay a penalty for serialization. Fork consistency [24, 27] relaxes strong consistency by allowing users to observe distinct causal histories. The primary drawback of fork consistency is that once replicas have forked, they can never be reconciled. Such approach is useful when building secure systems but is not appropriate in the context of geo-replication.

Eventual consistency [38] is on the other end of the spectrum. Eventual consistency is a catch-all phrase that covers any system where replicas may diverge in the short term as long as the divergence is eventually repaired and may or may not include causality. (See Saito and Shapiro [30] for a survey.) In practice, as shown in Table 1, systems that embrace eventual consistency have limitations. Some systems waive the stable history property, either by rolling back operations and re-executing them in a different order at some of the replicas [34], or by resorting to a last writer wins strategy, which often results in loss of one of the concurrent updates [25]. Other systems expose multiple values from divergent branches in operations replies either directly to the client [26, 9] or to an application-specific conflict resolution procedure [38]. Finally, some systems restrict operations by assuming that all operations in the system commute [12, 33], which might require the programmer

to rewrite or avoid using some operations.

**Coexistence of multiple consistency levels.** The solution we propose for addressing the tension between low latency and strongly consistent responses is to allow different operations to run with different consistency levels. Existing systems that used a similar approach include Horus [39], lazy replication [19], Zeno [34], and PSI [35]. However, none of these proposals guide the service developer in choosing between the available consistency levels. In particular, developers must reason about whether their choice leads to the desired service behavior, namely by ensuring that invariants are preserved and that replica state does not diverge. This can be challenging due to difficulties in identifying behaviors allowed by a specific consistency level and understanding the interplay between operations running at different levels. Our research addresses this challenge, namely by defining a set of conditions that precisely determine the appropriate consistency level for each operation.

**Other related work.** Consistency rationing [18] allows consistency guarantees to be associated with data instead of operations, and the consistency level to be automatically switched at runtime between weak consistency and serializability based on specified policies. TACT [41] consistency bounds the amount of inconsistency of data items in an application-specific manner, using the following metrics: numerical error, order error and staleness. In contrast to these models, the focus of our work is not on adapting the consistency levels of particular data items at runtime, but instead on systematically partitioning the space of operations according to their actions and the desired system semantics.

One of the central aspects of our work is the notion of shadow operations, which increase operation commutativity by decoupling the decision of the side effects from their application to the state. This enables applications to make more use of fast operations. Some prior work also aims at increasing operation commutativity: Weihl exploited commutativity-based concurrency control for abstract data types [40]; operational transformation [10, 12] extends non-commutative operations with a transformation that makes them commute; Conflict-free Replicated Data Types (CRDTs) [33] design operations that commute by construction; Gray [15] proposed an open nested transaction model that uses commutative compensating transactions to revert the effects of aborted transactions without rolling back the transactions that have seen their results and already committed; delta transactions [36] divide a transaction into smaller pieces that commute with each other to reduce the serializability requirements. Our proposal of shadow operations can be seen as an extension to these concepts, providing a different way of broadening the scope of potentially commutative operations. There exist other proposals that also decouple the execution into two parts, namely two-tier replication [16] and CRDT downstreams [33]. In contrast to these proposals, for each operation, we may generate different shadow operations based on the specifics of the execution. Also, shadow operations can run under different consistency levels, which is important because commutativity is not always sufficient to ensure safe weakly consistent operation.

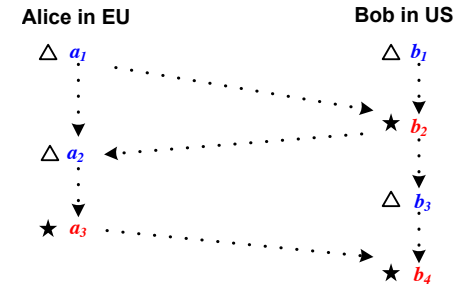
## 3 System model

We assume a distributed system with state fully replicated across $k$ sites denoted $site_0 \ldots site_{k-1}$. We follow the traditional deterministic state machine model, where there is a set of possible states $\mathcal{S}$ and a set of possible operations $O$, each replica holds a copy of the current system state, and upon applying an operation each replica deterministically transitions to the next state and possibly outputs a corresponding reply.

In our notation, $S \in \mathcal{S}$ denotes a system state, and $u, v \in O$ denote operations. We assume there exists an initial state $S_0$. If operation $u$ is applied against a system state $S$, it produces another system state $S'$; we will also denote this by $S' = S + u$. We say that a pair of operations $u$ and $v$ *commute* if $\forall S \in \mathcal{S}, S+u+v = S+v+u$. The system maintains a set of application-specific invariants. We say that state $S$ is *valid* if it satisfies all these invariants. Each operation $u$ is initially submitted at one site which we call $u$'s *primary site* and denote $site(u)$; the system then later replicates $u$ to the other sites.

## 4 RedBlue consistency

In this section we introduce RedBlue consistency, which allows replicated systems to be fast as possible and consistent when necessary. "Fast" is an easy concept to understand—it equates to providing low latency responses to user requests. "Consistent" is more nuanced—consistency models technically restrict the state that operations can observe, which can be translated to an order that operations can be applied to a system. Eventual consistency [25, 38, 26, 12], for example, permits operations to be partially ordered and enables fast systems—sites can process requests locally without coordinating with each other—but sacrifices the intuitive semantics of serializing updates. In contrast, linearizability [17] or serializability [5] provide strong consistency and allow for systems with intuitive semantics—in effect, all sites process operations in the same order—but require significant coordination between sites, precluding fast operation. RedBlue consistency is based on an explicit division of operations into blue operations whose order of execution can vary from site to site, and red operations that must be executed in the same order at all sites.




(a) RedBlue order $O$ of operations


(b) Causal serializations of $O$

Figure 1: RedBlue order and causal serializations for a system spanning two sites. Operations marked with ★ are red; operations marked with △ are blue. Dotted arrows in (a) indicate dependencies between operations.

## 4.1 Defining RedBlue consistency

The definition of RedBlue consistency has two components: (1) A RedBlue order, which defines a partial order of operations, and (2) a set of local causal serializations, which define site-specific total orders in which the operations are locally applied.

**Definition 1** (RedBlue order)**.** *Given a set of operations $U = B \cup R$, where $B \cap R = \emptyset$, a* RedBlue order *is a partial order $O = (U, \prec)$ with the restriction that $\forall u, v \in R$ such that $u \neq v$, $u \prec v$ or $v \prec u$ (i.e., red operations are totally ordered).*

Recall that each site is a deterministic state machine that processes operations in a serial order. The serial order executed by site $i$ is a causal serialization if it is compatible with the global RedBlue order and ensures causality for all operations initially executed at site $i$. A replicated system with $k$ sites is then RedBlue consistent if every site applies a causal serialization of the same global RedBlue order $O$.

**Definition 2** (Causal serialization)**.** *Given a site $i$, $O_i = (U, <)$ is an* $i$-causal serialization *(or short, a* causal serialization*) of RedBlue order $O = (U, \prec)$ if (a) $O_i$ is a linear extension of $O$ (i.e., $<$ is a total order compatible with the partial order $\prec$), and (b) for any two operations $u, v \in U$, if site$(v) = i$ and $u < v$ in $O_i$, then $u \prec v$.*

```
1  float balance, interest = 0.05;
2  func deposit( float money ):
3     balance = balance + money;
4  func withdraw ( float money ):
5     if ( balance - money >= 0 ) then:
6        balance = balance - money;
7     else print "failure";
8  func accrueinterest():
9     float delta = balance × interest;
10    balance = balance + delta;
```

Figure 2: Pseudocode for the bank example.

**Definition 3** (RedBlue consistency)**.** *A replicated system is* $O$-RedBlue consistent *(or short, RedBlue consistent) if each site $i$ applies operations according to an $i$-causal serialization of RedBlue order $O$.*

Figure 1 shows a RedBlue order and a pair of causal serializations of that RedBlue order. In systems where every operation is labeled red, RedBlue consistency is equivalent to serializability [5]; in systems where every operation is labeled blue, RedBlue consistency allows the same set of behaviors as eventual consistency [38, 25, 26]. It is important to note that while RedBlue consistency constrains possible orderings of operations at each site and thus the states the system can reach, it does not ensure *a priori* that the system achieves all the end-to-end properties identified in §2, in particular, state convergence and invariant preservation, as discussed next.

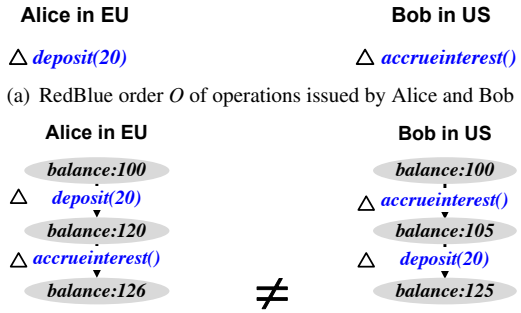
## 4.2 State convergence and a RedBlue bank

In order to understand RedBlue consistency it is instructive to look at a concrete example. For this example, consider a simple bank with two users: Alice in the EU and Bob in the US. Alice and Bob share a single bank account where they can deposit or withdraw funds and where a local bank branch can accrue interest on the account (pseudocode for the operations can be found in Figure 2). Let the `deposit` and `accrueinterest` operations be blue. Figure 3 shows a RedBlue order of deposits and interest accruals made by Alice and Bob and causal serializations applied at both branches of the bank.

State convergence is important for replicated systems. Intuitively a pair of replicas is state convergent if, after processing the same set of operations, they are in the same state. In the context of RedBlue consistency we formalize state convergence as follows:

**Definition 4** (State convergence)**.** *A RedBlue consistent system is state convergent if all causal serializations of the underlying RedBlue order $O$ reach the same state $S$.*

The bank example as described is not state convergent. The root cause is not surprising: RedBlue consistency allows sites to execute blue operations in different orders but two blue operations in the example correspond to non-commutative operations—addition (`dep-`




(a) RedBlue order $O$ of operations issued by Alice and Bob

(b) Causal serializations of $O$ leading to diverged state

Figure 3: A RedBlue consistent account with initial balance of $100.

osit) and multiplication (accrueinterest). A sufficient condition to guarantee state convergence in a RedBlue consistent system is that every blue operation is *globally commutative*, i.e., it commutes with all other operations, blue or red.

**Theorem 1.** *Given a RedBlue order O, if all blue operations are globally commutative, then any O-RedBlue consistent system is state convergent.*[1]

The banking example and Theorem 1 highlight an important tension inherent to RedBlue consistency. On the one hand, low latency requires an abundance of blue operations. On the other hand, state convergence requires that blue operations commute with all other operations, blue or red. In the next section we introduce a method for addressing this tension by increasing commutativity.

# 5 Replicating side effects

In this section, we observe that while operations themselves may not be commutative, *we can often make the changes they induce on the system state commute.* Let us illustrate this issue within the context of the RedBlue bank from §4.2. We can make the deposit and accrueinterest operations commute by first computing the amount of interested accrued and then treating that value as a deposit.

## 5.1 Defining shadow operations

The key idea is to split each original application operation $u$ into two components: a *generator operation* $g_u$ with no side effects, which is executed only at the primary site against some system state $S$ and produces a *shadow operation* $h_u(S)$, which is executed at every site (including the primary site). The generator operation decides which state transitions should be made while the shadow operation applies the transitions in a state-independent manner.

The implementation of generator and shadow operations must obey some basic correctness requirements. Generator operations, as mentioned, must not have any side effects. Furthermore, shadow operations must produce the same effects as the corresponding original operation when executed against the original state $S$ used as an argument in the creation of the shadow operation.

**Definition 5** (Correct generator / shadow operations)**.** *The decomposition of operation u into generator and shadow operations is correct if for all states S, the generator operation $g_u$ has no effect and the generated shadow operation $h_u(S)$ has the same effect as u, i.e., for any state S: $S+g_u = S$ and $S+h_u(S) = S+u$.*

Note that a trivial decomposition of an original operation $u$ into generator and shadow operations is to let $g_u$ be a no-op and let $h_u(S) = u$ for all $S$.

In practice, as exemplified in §7, separating the decision of which transition to make from the act of applying the transition allows many objects and their associated usage in shadow operations to form an abelian group and thus dramatically increase the number of commutative (i.e., blue) operations in the system. Unlike previous approaches [16, 33], for a given original operation, our solution allows its generator operation to generate state-specific shadow operations with different properties, which can then be assigned different colors in the RedBlue consistency model.

## 5.2 Revisiting RedBlue consistency

Decomposing operations into generator and shadow components requires us to revisit the foundations of RedBlue consistency. In particular, only shadow operations are included in a RedBlue order while the causal serialization for site $i$ additionally includes the generator operations initially executed at site $i$. The causal serialization must ensure that generator operations see the same state that is associated with the generated shadow operation and that shadow operations appropriately inherit all dependencies from their generator operation.

We capture these subtleties in the following revised definition of causal serializations. Let $U$ be the set of shadow operations executed by the system and $V_i$ be the generator operations executed at site $i$.

**Definition 6** (Causal serialization–revised)**.** *Given a site i, $O_i = (U \cup V_i, <)$ is an i-causal serialization of RedBlue order $O = (U, \prec)$ if*

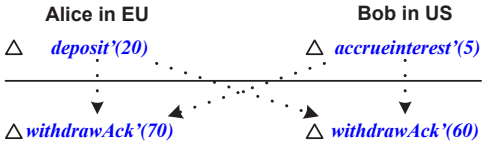
- *$O_i$ is a total order;*
- *$(U, <)$ is a linear extension of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, S is the state obtained after applying the sequence of shadow operations preceding $g_v$ in $O_i$;*
- *For any $g_v \in V_i$ and $h_u(S) \in U$, $h_u(S) < g_v$ in $O_i$ iff $h_u(S) \prec h_v(S')$ in O.*

Note that shadow operations appear in every causal serialization, while generator operations appear only in the causal serialization of the initially executing site.

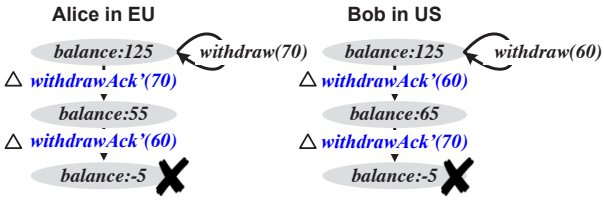
[1] All proofs can be found in a separate technical report [22].

```
1  func deposit' ( float money ):
2    balance = balance + money;
3  func withdrawAck' ( float money ):
4    balance = balance - money;
5  func withdrawFail' ():
6    /* no-op */
7  func accrueinterest' ( float delta ):
8    balance = balance + delta;
```

Figure 4: Pseudocode for shadow bank operations.




(a) RedBlue order $O$ of banking shadow operations




(b) Invalid but convergent causal serializations of $O$

Figure 5: A RedBlue consistent bank with only blue operations. The starting balance of $125 is the result of applying shadow operations above the solid line to an initial balance of $100. Loops indicate generator operations.

## 5.3 Shadow banking and invariants

Figure 4 shows the shadow operations for the banking example. Note that the `withdraw` operation maps to two distinct shadow operations that may be labeled as blue or red independently—`withdrawAck'` and `withdrawFail'`.

Figure 5 illustrates that shadow operations make it possible for all operations to commute, provided that we can identify the underlying abelian group. This does not mean, however, that it is safe to label all operations blue.

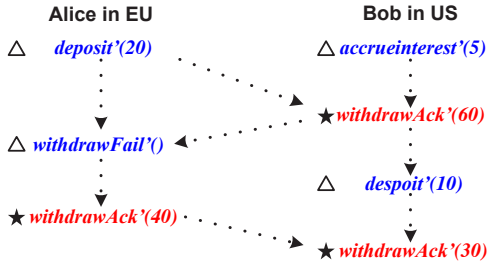
In this example, such a labeling would allow Alice and Bob to successfully withdraw $70 and $60 at their local branches, thus ending up with a final balance of $-5. This violates the fundamental invariant that a bank balance should never be negative.

To determine which operations can be safely labeled blue, we begin by defining that a shadow operation is invariant safe if, when applied to a valid state, it always transitions the system into another valid state.

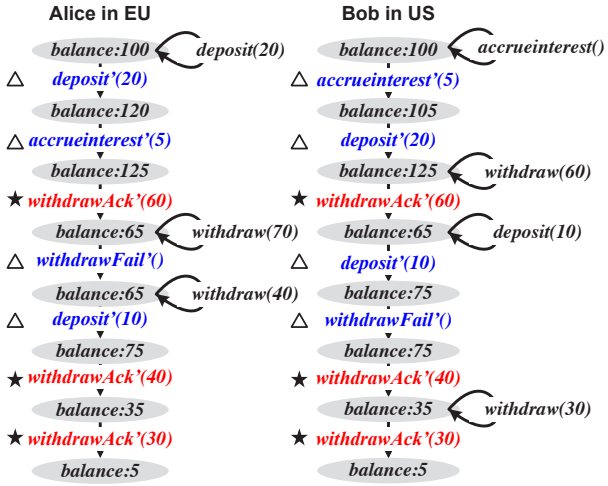
**Definition 7** (Invariant safe)**.** *Shadow operation $h_u(S)$ is invariant safe if for all valid states $S$ and $S'$, the state $S' + h_u(S)$ is also valid.*

The following theorem states that in a RedBlue consistent system with appropriate labeling, each replica transitions only through valid states.

**Theorem 2.** *If all shadow operations are correct and all blue shadow operations are invariant safe and globally commutative, then for any execution of that system that is RedBlue consistent, no site is ever in an invalid state.*




(a) RedBlue order $O$ of banking shadow operations




(b) Convergent and invariant preserving causal serializations of $O$

Figure 6: A RedBlue consistent bank with correctly labeled shadow operations and initial balance of $100.

**What can be blue? What must be red?** The combination of Theorems 1 and 2 leads to the following procedure for deciding which shadow operations can be blue or must be red if a RedBlue consistent system is to provide both state convergence and invariant preservation:

1. For any pair of non-commutative shadow operations $u$ and $v$, label both $u$ and $v$ red.
2. For any shadow operation $u$ that may result in an invariant being violated, label $u$ red.
3. Label all non-red shadow operations blue.

Applying this decision process to the bank example leads to a labeling where `withdrawAck'` is red and the remaining shadow operations are blue. Figure 6 shows a RedBlue order with appropriately labeled shadow operations and causal serializations for the two sites.

## 5.4 Discussion

Shadow operations introduce some surprising anomalies to a user experience. Notably, while the effect of every user action is applied at every site, the final system state is not guaranteed to match the state resulting from a serial ordering of the original operations. The important thing to keep in mind is that the decisions

made always make sense in the context of the *local* view of the system: when Alice accrues interest in the EU, the amount of interest accrued is based on the balance that Alice observes at that moment. If Bob concurrently makes a deposit in the US and subsequently observes that interest has been accrued, the amount of interest *will not* match the amount that Bob would accrue based on the balance as he currently observes it.

Shadow operations always provide for a coherent sequence of state transitions that reflects the effects demanded by user activity; while this sequence of state transitions is coherent (and convergent), the state transitions are chosen based on the locally observable state when/where the user activity initiated and not the system state when they are applied.

# 6 Gemini design & implementation

We implemented the Gemini storage system to provide RedBlue consistency. The prototype consists of 10K lines of java code and uses MySQL as its storage backend. Each Gemini site consists of four components: a storage engine, a proxy server, a concurrency coordinator, and a data writer. A multi-site deployment is constructed by replicating the single site components across multiple sites.

The basic flow of user requests through the system is straightforward. A user issues requests to a *proxy server* located at the closest site. The proxy server processes a request by executing an appropriate application transaction, which is implemented as a single Gemini operation, comprising multiple data accesses; individual data accesses within a generator operation execute in a temporary private scratchpad, providing a virtual private copy of the service state. The original data lies in a *storage engine*, which provides a standard storage interface. In our implementation, the storage engine is a relational database, and scratchpad operations are executed against a temporary table. Upon completion of the generator operation, the proxy server sends the produced shadow operation on to the *concurrency coordinator* to admit or reject this operation according to RedBlue consistency. The concurrency coordinator notifies the proxy server if the operation is accepted or rejected. Additionally, accepted shadow operations are appended to the end of the local causal serialization and propagated to remote sites and to the local *data writer* for execution against the storage engine. When a shadow operation is rejected, the proxy server re-executes the generator operation and restarts the process.

## 6.1 Optimistic concurrency control

Gemini relies on optimistic concurrency control (OCC) [5] to run generator operations without blocking.

Gemini uses timestamps to determine if operations can complete successfully. Timestamps are logical clocks [20] of the form $\langle\langle b_0, b_1, \ldots, b_{k-1}\rangle, r\rangle$, where $b_i$ is the local count of shadow operations initially executed by site $i$ and $r$ is the global count of red shadow operations. To ensure that different sites do not choose the same red sequence number (i.e., all red operations are totally-ordered) we use a simple token passing scheme: only the coordinator in possession of a unique red token is allowed to increase the counter $r$ and approve red operations. In the current prototype, a coordinator holds onto the red token for up to 1 second before passing it along.

When a generator operation completes, the coordinator must determine if the operation (a) reads a coherent system snapshot and (b) obeys the ordering constraints of a causal serialization, as described in §5. To do this, the coordinator checks the timestamps of the data items readd and written by the completing operation, and compares them to the timestamps associated with operations completing concurrently and the remote shadow operations that were being applied simultaneously at that site.

Upon successful completion of a generator operation the coordinator assigns the corresponding shadow operation a timestamp that is component-wise equal to the latest operation that was incorporated at its site, and increments its blue and, if this shadow operations is red, the red component of the logical timestamp. This timestamp determines the position of the shadow operation in the RedBlue order, with the normal rules that determine that two operations are partially ordered if one is equal to or dominates the other in all components. It also allows sites to know when it is safe to incorporate remote shadow operations: they must wait until all shadow operations with smaller timestamps have already been incorporated in the local state of the site. When a remote shadow operation is applied at a site, it is assigned a new timestamp that is the entry-wise max of the timestamp assigned to the shadow operation in the initial site and the local timestamps of accessed data objects. This captures dependencies that span local and remote operations.

**Read-only shadow operations.** As a performance optimization, a subset of blue shadow operations can be marked read-only. Read-only shadow operations receive special treatment from the coordinator: once the generator operation passes the coherence and causality checks, the proxy is notified that the shadow operation has been accepted but the shadow operation is *not* incorporated into the local serialization or global RedBlue order.

## 6.2 Failure handling

The current Gemini prototype is designed to demonstrate the performance potential of RedBlue consistency in geo-replicated environments and as such is not implemented to tolerate faults of either a local (i.e., within a site) or catastrophic (i.e., of an entire site) nature. Addressing these concerns is orthogonal to the primary con-

tributions of this paper, nonetheless we briefly sketch mechanisms that could be employed to handle faults.

**Isolated component failure.** The Gemini architecture consists of four main components at each site, each representing a single point of failure. Standard state machine replication techniques [20, 31] can be employed to make each component robust to failures.

**Site failure.** Our Gemini prototype relies on a simple token exchange for coordinating red epochs. To avoid halting the system upon a site failure, a fault tolerant consensus protocol like Paxos [21] can regulate red tokens.

**Operation propagation.** Gemini relies on each site to propagate its own local operations to all remote sites. A pair-wise network outage or failure of a site following the replication of a operation to some but not all of the sites could prevent sites from exchanging operations that depend on the partially replicated operation. This can be addressed using standard techniques for exchanging causal logs [26, 2, 38, 28] or reliable multicast [13].

**Cross-session monotonicity.** The proxy that each user connects to enforces the monotonicity of user requests within a session [37]. However, a failure of that proxy, or the user connecting to a different site may result in a subset of that user's operations not carrying over. This can be addressed by allowing the user to specify a "last-read" version when starting a new session or requiring the user to cache all relevant requests [26] in order to replay them when connecting to a new site.

# 7 Case studies

In this section we report on our experience in modifying three existing applications—the TPC-W shopping cart benchmark [7], the RUBiS auction benchmark [11], and the Quoddy social networking application [14]—to work with RedBlue consistency. The two main tasks to fulfill this goal are (1) decomposing the application into generator and shadow operations and (2) labeling the shadow operations appropriately.

**Writing generator and shadow operations.** Each of the three case study applications executes MySQL database transactions as part of processing user requests, generally one transaction per request. We map these application level transactions to the original operations and they also serve as a starting point for the generator operations. For shadow operations, we turn each execution path in the original operation into a distinct shadow operation; an execution path that does not modify system state is explicitly encoded as a no-op shadow operation.When the shadow operations are in place, the generator operation is augmented to invoke the appropriate shadow operation at each path.

**Labeling shadow operations.** Table 2 reports the number of transactions in the TPC-W, RUBiS, and Quoddy, the number of blue and red shadow operations we identified using the labeling rules in §5.3, and the application changes measured in lines of code. Note that read-only transactions always map to blue no-op shadow operations. In the rest of this section we expand on the lessons learned from making applicaations RedBlue consistent.

## 7.1 TPC-W

TPC-W [7] models an online bookstore. The application server handles 14 different user requests such as browsing, searching, adding products to a shopping cart, or placing an order. Each user request generates between one and four transactions that access state stored across eight different tables. We extend an open source implementation of the benchmark [29] to allow a shopping cart to be shared by multiple users across multiple sessions.

**Writing TPC-W generator and shadow operations.** Of the twenty TPC-W transactions, thirteen are read-only and admit no-op shadow operations. The remaining seven update transactions translate to one or more shadow operations according to the number of distinct execution paths in the original operation .

We now give an example transaction, `doBuyConfirm`, which completes a user purchase. The pseudocode for the original transaction is shown in Figure 7(a).

The `doBuyConfirm` transaction removes all items from a shopping cart, computes the total cost of the purchase, and updates the stock value for the purchased items. If the stock would drop below a minimum threshold, then the transaction also replenishes the stock. The key challenge in implementing shadow operations for `doBuyConfirm` is that the original transaction does not commute with itself or any transaction that modifies the contents of a shopping cart. Naively treating the original transaction as a shadow operation would force every shadow operation to be red.

Figure 7(b) shows the generator operation of `doBuyConfirm`, and Figures 7(c) and 7(d) depict the corresponding pair of shadow operations: `doBuyConfirmIncre'` and `doBuyConfirmDecre'`. The former shadow operation is generated when the stock falls below the minimum threshold and must be replenished; the latter is generated when the purchase does not drive the stock below the minimum threshold and consequently does not trigger the replenishment path. In both cases, the generator operation is used to determine the number of items purchased and total cost as well the shadow operation that corresponds to the initial execution. At the end of the execution of the generator operation these parameters and the chosen shadow operation are then propagated to other replicas.

**Labeling TPC-W shadow operations.** For 29 shadow operations in TPC-W, we find that 27 can be blue and only two must be red. To label shadow operations, we

| Application | Original | | | | | RedBlue consistent extension | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | user requests | transactions | | | LOC | shadow operations | | | | LOC changed |
| | | total | read-only | update | | blue no-op | blue update | red | LOC | |
| TPC-W | 14 | 20 | 13 | 7 | 9k | 13 | 14 | 2 | 2.8k | 429 |
| RUBiS | 26 | 16 | 11 | 5 | 9.4k | 11 | 7 | 2 | 1k | 180 |
| Quoddy | 13 | 15 | 11 | 4 | 15.5k | 11 | 4 | 0 | 495 | 251 |

Table 2: Original applications and the changes needed to make them RedBlue consistent.

```
1  doBuyConfirm(cartId){
2   beginTxn();
3   cart = exec(SELECT * FROM cartTb WHERE cId=cartId);
4   cost = computeCost(cart);
5   orderId = getUniqueId();
6   exec(INSERT INTO orderTb VALUES(orderId,cart.item.id,cart.item.qty
          ,cost));
7   item =exec(SELECT * FROM itemTb WHERE id=cart.item.id);
8   if item.stock−cart.item.qty < 10 then:
9      delta = item.stock−cart.item.qty+21;
10     if delta > 0 then:
11        exec(UPDATE itemTb SET item.stock+ = delta);
12     else rollback();
13  else exec(UPDATE itemTb SET item.stock− = cart.item.qty);
14  exec(DELETE FROM cartContentTb WHERE cId=cartId AND id=
          cart.item.id);
15  commit();}
```

(a) Original transaction that commits changes to database.

```
1  doBuyConfirmGenerator(cartId){
2   sp = getScratchpad();
3   sp.beginTxn();
4   cart = sp.exec(SELECT * FROM cartTb WHERE cId=cartId);
5   cost = computeCost(cart);
6   orderId = getUniqueId();
7   sp.exec(INSERT INTO orderTb VALUES (orderId,cart.item.id,
          cart.item.qty,cost));
8   item = sp.exec(SELECT * FROM itemTb WHERE id=cart.item.id);
9   if item.stock−cart.item.qty < 10 then:
10     delta = item.stock−cart.item.qty+21;
11     if delta > 0 sp.exec(UPDATE itemTb SET item.stock+ = delta);
12     else sp.discard(); return;
13  else sp.exec(UPDATE itemTb SET item.stock− = cart.item.qty);
14  sp.exec(DELETE FROM cartTb WHERE cId=cartId AND id=cart.item.id);
15  L_TS = getCommitOrder();
16  sp.discard();
17  if replenished return (doBuyConfirmIncre′(orderId,cartId,
          cart.item.id,cart.item.qty,cost,delta,L_TS));
18  else return (doBuyConfirmDecre′(orderId,cartId,cart.item.Id,
          cart.item.qty,cost,L_TS));}
```

(b) Generator operation that manipulates data via a private *scratchpad*.

```
1  doBuyConfirmIncre′(orderId,cartId,itId,qty,cost,delta,L_TS){
2   exec(INSERT INTO orderTb VALUES(orderId,itId,qty,cost,L_TS));
3   exec(UPDATE itemTb SET item.stock+ = delta);
4   exec(UPDATE itemTb SET item.l_ts = L_TS WHERE item.l_ts < L_TS);
5   exec(UPDATE cartContentTb SET flag = TRUE WHERE id = itId AND
          cid = cartId AND l_ts <= L_TS);}
```

(c) Shadow doBuyConfirmIncre' (Blue) that replenishes the stock value.

```
1  doBuyConfirmDecre′(orderId,cartId,itId,qty,cost,L_TS){
2   exec(INSERT INTO orderTb VALUES(orderId,itId,qty,cost,L_TS));
3   exec(UPDATE itemTb SET item.stock− = qty);
4   exec(UPDATE itemTb SET item.l_ts = L_TS WHERE item.l_ts < L_TS);
5   exec(UPDATE cartContentTb SET flag = TRUE WHERE id = itId AND
          cid = cartId AND l_ts <= L_TS);}
```

(d) Shadow doBuyConfirmDecre' (Red) that decrements the stock value.

Figure 7: Pseudocode for the product purchase transaction in TPC-W. For simplicity the pseudocode assumes that the corresponding shopping cart only contains a single type of item.

identified two key invariants that the system must maintain. First, the number of in-stock items can never fall below zero. Second, the identifiers generated by the system (e.g., for items or shopping carts) must be unique.

The first invariant is easy to maintain by labeling `doBuyConfirmDecre′` (Figure 7(d)) and its close variant `doBuyConfirmAddrDecre′` red. We observe that they are the only shadow operations in the system that decrease the stock value, and as such are the only shadow operations that can possibly invalidate the first invariant. Note that the companion shadow operation `doBuyConfirmIncre′` (Figure 7(c)) *increases* the stock level, and can never drive the stock count below zero, so it can be blue.

The second invariant is more subtle. TPC-W generates IDs for objects (e.g., shopping carts, items, etc.) as they are created by the system. These IDs are used as keys for item lookups and consequently must themselves be unique. To preserve this invariant, we have to label many shadow operations red. This problem is well-known in database replication [6] and was circumvented by modifying the ID generation code, so that IDs become a pair $\langle appproxy\_id,\ seqnumber \rangle$, which makes these operations trivially blue.

## 7.2 RUBiS

RUBiS [11] emulates an online auction website modeled after eBay [1]. RUBiS defines a set of 26 requests that users can issue ranging from selling, browsing for, bidding on, or buying items directly, to consulting a personal profile that lists outstanding auctions and bids. These 26 user requests are backed by a set of 16 transactions that access the storage backend.

Of these 16 transactions, 11 are read-only, and therefore trivially commutative. For the remaining 5 update transactions, we construct shadow operations to make them commute, similarly to TPC-W. Each of these transactions leads to between 1 and 3 shadow operations.

Through an analysis of the application logic, we determined three invariants. First, that identifiers assigned by the system are unique. Second, that nicknames chosen by users are unique. Third, that item stock cannot fall below zero. Again, we preserve the first invariant using the global id generation strategy described in §7.1. The second and third invariants require both `RegisterUser′`, checking if a name submitted by a user was already chosen, and `storeBuyNow′`, which decreases stock, to be

labeled as red.

## 7.3 Quoddy

Quoddy [14] is an open source Facebook-like social networking site. Despite being under development, Quoddy already implements the most important features of a social networking site, such as searching for a user, browsing user profiles, adding friends, posting a message, etc. These main features define 13 user requests corresponding to 15 different transactions. Of these 15 transactions, 11 are read-only transactions, thus requiring trivial no-op shadow operations.

Writing and labeling shadow operations for the 4 remaining transactions in Quoddy was straightforward. Besides reusing the recipe for unique identifiers, we only had to handle an automatic conversion of dates to the local timezone (performed by default by the database) by storing dates in UTC in all sites. In the social network we did not find system invariants to speak of; we found that all shadow operations could be labeled blue.

## 7.4 Experience and discussion

Our experience showed that writing shadow operations is easy; it took us about one week to understand the code, and implement and label shadow operations for all applications. We also found that the strategy of generating a different shadow operation for each distinct execution path is beneficial for two reasons. First, it leads to a simple logic for shadow operations that can be based on operations that are intrinsically commutative, e.g., *increment/decrement*, *insertion/removal*. Second, it leads to a fine-grained classification of operations, with more execution paths leading to blue operations. Finally, we found that it was useful in more than one application to make use of a standard last-writer-wins strategy to make operations that overwrite part of the state commute.

# 8 Evaluation

We evaluate Gemini and RedBlue consistency using microbenchmarks and our three case study applications. The primary goal of our evaluation is to determine if RedBlue consistency can improve latency and throughput in geo-replicated systems.

## 8.1 Experimental setup

We run experiments on Amazon EC2 using extra large virtual machine instances located in five sites: US east (UE), US west (UW), Ireland (IE), Brazil (BR), and Singapore (SG). Table 3 shows the average round trip latency and observed bandwidth between every pair of sites. For experiments with fewer than 5 sites, new sites are added in the following order: UE, UW, IE, BR, SG. Unless otherwise noted, users are evenly distributed across all sites. Each VM has 8 virtual cores and 15GB of RAM. VMs run Debian 6 (Squeeze) 64 bit, MySQL 5.5.18, Tomcat 6.0.35, and Sun Java SDK 1.6. Each experimental run lasts for 10 minutes.

| | UE | UW | IE | BR | SG |
|---|---|---|---|---|---|
| UE | 0.4 ms<br>994 Mbps | 85 ms<br>164 Mbps | 92 ms<br>242 Mbps | 150 ms<br>53 Mbps | 252 ms<br>86 Mbps |
| UW | | 0.3 ms<br>975 Mbps | 155 ms<br>84 Mbps | 207 ms<br>35 Mbps | 181 ms<br>126 Mbps |
| IE | | | 0.4 ms<br>996 Mbps | 235 ms<br>54 Mbps | 350 ms<br>52 Mbps |
| BR | | | | 0.3 ms<br>993 Mbps | 380 ms<br>65 Mbps |
| SG | | | | | 0.3 ms<br>993 Mbps |

Table 3: Average round trip latency and bandwidth between Amazon sites.

## 8.2 Microbenchmark

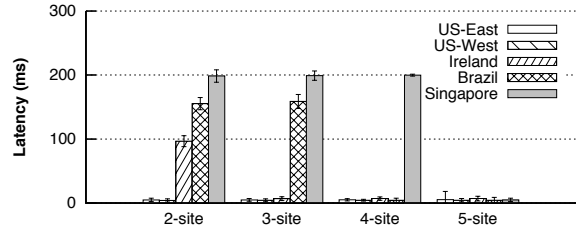
We begin the evaluation with a simple microbenchmark designed to stress the costs and benefits of partitioning operations into red and blue sets. Each user issues requests accessing a random record from a MySQL database. Each request maps to a single shadow operation; we say a request is blue if it maps to a blue shadow operation and red otherwise. The offered workload is varied by adjusting the number of outstanding requests per user and the ratio of red and blue requests.

We run the microbenchmark experiments with a dataset consisting of 10 tables, each initialized with 1,000,000 records; each record has 1 text and 4 integer attributes. The total size of the dataset is 1.0 GB.

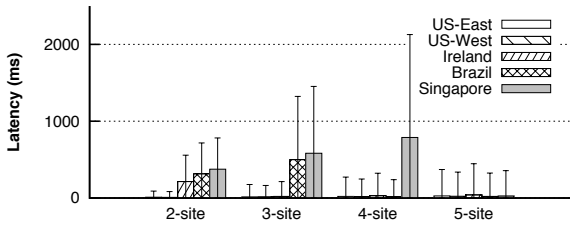
### 8.2.1 User observed latency

The primary benefit of using Gemini across multiple sites is the decrease in latency from avoiding the intercontinental round-trips as much as possible. As a result, we first explore the impact of RedBlue consistency on user experienced latency. In the following experiments each user issues a single outstanding request at a time.

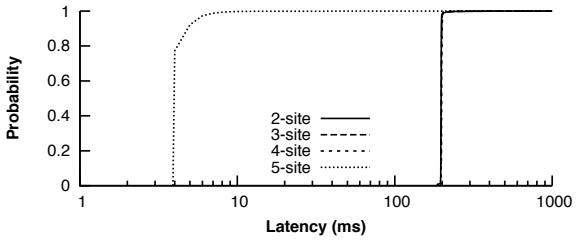
Figure 8(a) shows that the average latency for blue requests is dominated by the latency between the user and the closest site; as expected, average latency decreases as additional sites appear close to the user. Figure 8(b) shows that this trend also holds for red requests. The average latency and standard deviation, however, are higher for red requests than for blue requests. To understand this effect, we plot in Figures 8(c) and 8(d) the CDFs of observed latencies for blue and red requests, respectively, from the perspective of users located in Singapore. The observed latency for blue requests tracks closely with the round-trip latency to the closest site. For red requests, in the $k = 2$ through $k = 4$ site configurations, four requests from a user in Singapore are processed at the closest site during the one second in which the closest site holds the red token; every fifth request must wait $k - 1$ seconds for the token to return. In the 5 site configuration, the local site also becomes a replica of the service and therefore a much larger number of requests (more than 300) can be processed while the local site holds the red token. This changes the format of the curve, since there is now



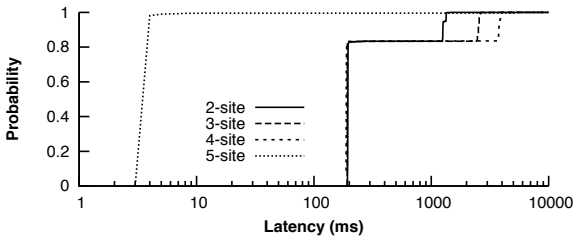

(a) Blue request latency for all users as number of sites increases




(b) Red request latency for all users as number of sites increases




(c) Blue latency CDF for Singapore users as number of sites increases




(d) Red latency CDF for Singapore users as number of sites increases

Figure 8: (a) and (b) show the average latency and standard deviation for blue and red requests issued by users in different locales as the number of sites is increased, respectively. (c) and (d) show the CDF of latencies for blue and red requests issued by users in Singapore as the number of sites is increased, respectively.

a much smaller fraction of requests that need to wait four seconds for the token to return.

### 8.2.2 Peak throughput

We now shift our attention to the throughput implications of RedBlue consistency. Figure 9 shows a throughput-latency graph for a 2 site configuration and three workloads: 100% blue, 100% red, and a 70% blue/30% red mix. The different points in each curve are obtained by increasing the offered workload, which is achieved by increasing the number of outstanding requests per user. For the mixed workload, users are partitioned into blue and red sets responsible for issuing requests of the specified color and the ratio is a result of this configuration.

The results in Figure 9 show that increasing the ratio of red requests degrades both latency and throughput. In particular, the two-fold increase in throughput for the all blue workload in comparison to the all red workload is a direct consequence of the coordination (not) required to process red (blue) requests: while red requests can only be executed by the site holding the red token to process, every site may independently process blue requests. The peak throughput of the mixed workload is proportionally situated between the two pure workloads.

## 8.3 Case studies: TPC-W and RUBiS

Our microbenchmark experiments indicate that RedBlue consistency instantiated with Gemini offers latency and throughput benefits in geo-replicated systems with sufficient blue shadow operations. Next, we evaluate Gemini using TPC-W and RUBiS.

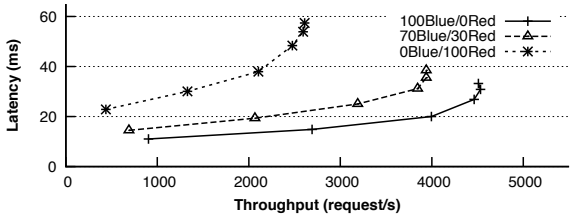


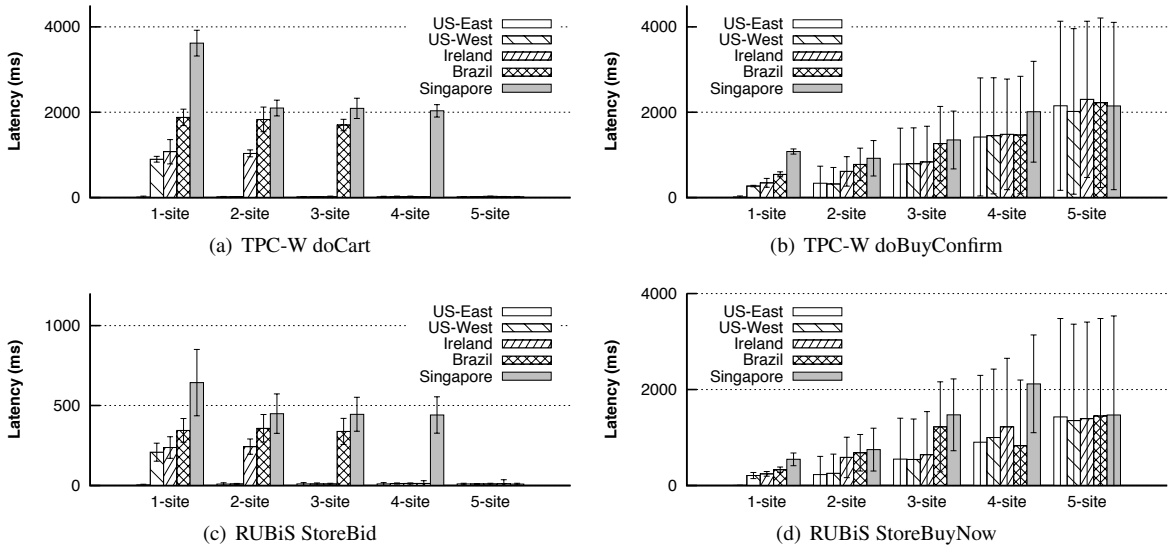

Figure 9: Throughput versus latency graph for a 2 site configuration with varying red-blue workload mixes.

### 8.3.1 Configuration and workloads

In all case studies experiments a single site configuration corresponds to the original unmodified code with users distributed amongst all five sites. Two through five site configurations correspond to the modified RedBlue consistent systems running on top of Gemini. When necessary, we modified the provided user emulators so that each user maintains $k$ outstanding requests and issues the next request as soon as a response is received.

**TPC-W.** TPC-W [7] defines three workload mixes differentiated by the percentage of client requests related to making purchases: browsing (5%), shopping (20%), ordering (50%). The dataset is generated with the following TPC-W parameters: 50 EBS and 10,000 items.

**RUBiS.** RUBiS defines two workload mixes: browsing, exclusively comprised of read-only interactions, and bidding, where 15% of user interactions are updates. We evaluate only the bidding mix. The RUBiS database contains 33,000 items for sale, 1 million users, 500,000 old items and is 2.1 GB in total.




(a) TPC-W doCart (b) TPC-W doBuyConfirm

(c) RUBiS StoreBid (d) RUBiS StoreBuyNow

Figure 10: Average latency for selected TPC-W and RUBiS user interactions. Shadow operations for doCart and StoreBid are always blue; for doBuyConfirm and StoreBuyNow they are red 98% and 99% of the time respectively.

| | Blue | Red | read-only | update |
|---|---|---|---|---|
| TPC-W shop | 99.2 | 0.8 | 85 | 15 |
| TPC-W browse | 99.5 | 0.5 | 96 | 4 |
| TPC-W order | 93.6 | 6.4 | 63 | 37 |
| RUBiS bid | 97.4 | 2.6 | 85 | 15 |

Table 4: Proportion of blue and red shadow operations and read-only and update requests in TPC-W and RUBiS workloads at runtime.

#### 8.3.2 Prevalence of blue and red shadow operations

Table 4 shows the distribution of blue and red shadow operations during execution of the TPC-W and RUBiS workloads. The results show that TPC-W and RUBiS exhibit sufficient blue shadow operations for it to be likely that we can exploit the potential of RedBlue consistency.

#### 8.3.3 User observed latency

We first explore the per request latency for a set of exemplar red and blue requests from TPC-W and RUBiS. For this round of experiments, each site hosts a single user issuing one outstanding request to the closest site.

From TPC-W we select doBuyConfirm (discussed in detail in §7.1) as an exemplar for red requests and doCart (responsible for adding/removing items to/from a shopping cart) as an exemplar for blue requests; from RUBiS we identify StoreBuyNow (responsible for purchasing an item at the buyout price) as an exemplar for red requests and StoreBid (responsible for placing a bid on an item) as an exemplar for blue requests. Note that doBuyConfirm and StoreBid can produce either red or blue shadow operations; in our experience they produce red shadow operations 98% and 99% of the time respectively.

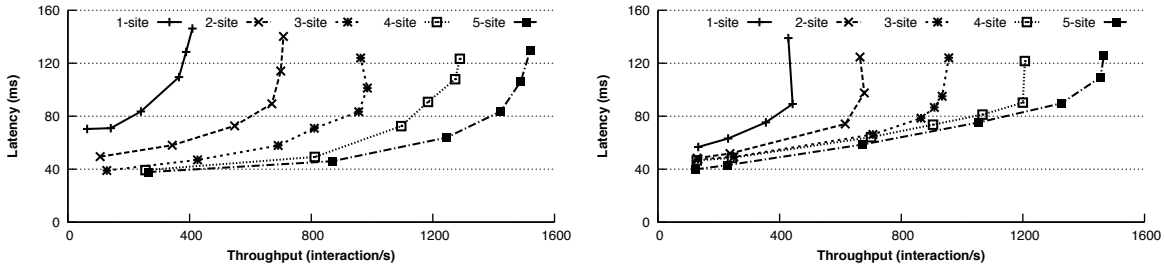
Figures 10(a) and 10(c) show that the latency trends for blue shadow operations are consistent with the results from the microbenchmark—observed latency is directly proportional to the latency to the closest site. The raw latency values are higher than the round-trip time from the user to the nearest site because processing each request involves sending one or more images to the user.

For red requests, Figures 10(b) and 10(d) show that llatency and standard deviation both increase with the number of sites. The increase in standard deviation is an expected side effect of the simple scheme that Gemini uses to exchange the red token and is consistent with the microbenchmark results. Similarly, the increase in average latency is due to the fact that the time for a token rotation increases, together with the fact that red requests are not frequent enough that several cannot be slipped in during the same token holding interval. We note that the token passing scheme used by Gemini is simple and additional work is needed to identify an optimal strategy for regulating red shadow operations.

#### 8.3.4 Peak throughput

We now shift our attention to the throughput afforded by our RedBlue consistent versions of TPC-W and RUBiS, and how it scales with the number of sites. For these experiments we vary the workload by increasing the number of outstanding requests maintained by each user. Throughput is measured according to interactions per second, a metric defined by TPC-W to correspond to user requests per second.

Figure 11 shows throughput and latency for the TPC-W shopping mix and RUBiS bidding mix as we vary the number of sites. In both systems, increasing the number of sites increases peak throughput and decreases average latency. The decreased latency results from situating users closer to the site processing their requests. The increase in throughput is due to processing blue and read-only operations at multiple sites, given that processing




(a) TPC-W shopping mix

(b) RUBiS bidding mix

Figure 11: Throughput versus latency for the TPC-W shopping mix and RUBiS bidding mix. The 1-site line corresponds to the original code; the 2/3/4/5-site lines correspond to the RedBlue consistent system variants.

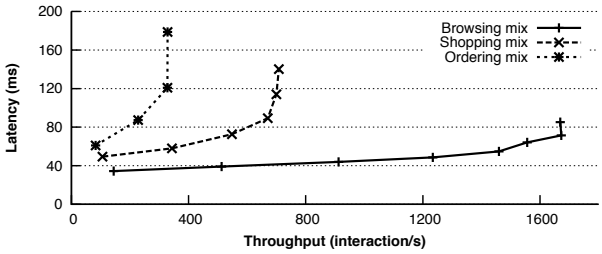



Figure 12: TPC-W: Throughput vs. latency graph for TPC-W with Gemini spanning two sites when running the three workload mixes.

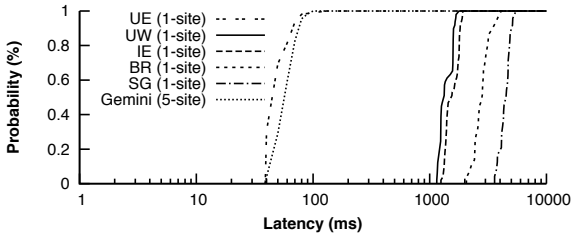



Figure 13: User latencies CDF for the addFriend request in single site Quoddy and 5-site Gemini deployments.

| | TPC-W shopping | | RUBiS biding | |
|---|---|---|---|---|
| | Original | Gemini | Original | Gemini |
| Thput. (inter/s) | 409 | 386 | 450 | 370 |
| Avg. latency | 14 ms | 15 ms | 6 ms | 7 ms |

Table 5: Performance comparison between the original code and the Gemini version for both TPC-W and RUBiS within a single site.

their side effects is relatively inexpensive. The speedup for a 5 site Gemini deployment of TPC-W is 3.7x against the original code for the shopping mix; the 5 site Gemini deployment of RUBiS shows a speedup of 2.3x.

Figure 12 shows the throughput and latency graph for a two site configuration running the TPC-W browsing, shopping, and ordering mixes. As expected, the browsing mix, which has the highest percentage of blue and read-only requests, exhibits the highest peak throughput, and the ordering mix, with the lowest percentage of blue and read-only requests, exhibits the lowest peak throughput.

## 8.4 Case study: Quoddy

Quoddy differs from TPC-W and RUBiS in one crucial way: it has no red shadow operations. We use Quoddy to show the full power of RedBlue geo-replication.

Quoddy does not define a benchmark workload for testing purposes. Thus we design a social networking workload generator based on the measurement study of Benevenuto et al. [4]. In this workload, 85% of the interactions are read-only page loads and 15% of the interactions include updates, e.g., request friendship, confirm friendship, or update status. Our test database contains 200,000 users and is 2.6 GB in total size.

In a departure from previous experiments, we run only two configurations. The first is the original Quoddy code in a single site. The second is our Gemini based RedBlue consistent version replicated across 5 sites. In both configurations, users are distributed in all 5 regions.

Figure 13 shows the CDF of user experienced latencies for the addFriend operation. All Gemini users experience latency comparable to the local users in the original Quoddy deployment; a dramatic improvement for users not based in the US East region. The significantly higher latencies for remote regions are associated with the images and javascripts that Quoddy distributes as part of processing the addFriend request.

## 8.5 Gemini overheads

Gemini is a middleware layer that interposes between the applications that leverage RedBlue consistency and a set of database systems where data is stored. We evaluate the performance overhead imposed by our prototype by comparing the performance of a single site Gemini deployment with the unmodified TPC-W and RUBiS systems directly accessing a database. For this experiment we locate all users in the same site as the service.

Table 5 presents the peak throughput and average latency for the TPC-W shopping and RUBiS bidding mixes. The peak throughput of a single site Gemini deployment is between 82% and 94% of the original and Gemini increases latency by 1ms per request.

# 9 Conclusion

In this paper, we presented a principled approach to building geo-replicated systems that are fast as possible

and consistent when needed. Our approach is based on our novel notion of RedBlue consistency allowing both strongly consistent (red) operations and eventually consistent (blue) operations to coexist, a concept of shadow operation enabling the maximum usage of blue operations, and a labeling methodology for precisely determining which operations to be assigned which consistency level. Experimental results from running benchmarks with our system Gemini show that RedBlue consistency significantly improves the performance of geo-replicated systems.

## Acknowledgments

We sincerely thank Edmund Wong, Rose Hoberman, Lorenzo Alvisi, our shepherd Jinyang Li, and the anonymous reviewers for their insightful comments and suggestions. The research of R. Rodrigues has received funding from the European Research Council under an ERC starting grant. J. Gehrke was supported by the National Science Foundation under Grant IIS-1012593, the iAd Project funded by the Research Council of Norway, a gift from amazon.com, and a Humboldt Research Award from the Alexander von Humboldt Foundation. N. Preguiça is supported by FCT/MCT projects PEst-OE/EEI/UI0527/2011 and PTDC/EIA-EIA/108963/2008.

## References

[1] Ebay website. http://www.ebay.com/, 2012.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. Technical report, Georgia Institute of Technology, 1994.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[4] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *IMC*, 2009.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. 1987.

[6] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*, 2008.

[7] T. consortium. Tpc benchmark-w specification v. 1.8. http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf, 2002.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[10] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.

[11] C. Emmanuel and M. Julie. Rubis: Rice university bidding system. http://rubis.ow2.org/, 2009.

[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[13] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 1997.

[14] Fogbeam Labs. Quoddy code repository, 2012. http://code.google.com/p/quoddy/.

[15] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[16] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 1990.

[18] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. In *VLDB*, 2009.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 1992.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[22] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. Technical report, MPI-SWS. http://www.mpi-sws.org/chengli/rbTR.pdf, 2012.

[23] H. Li. Practical consistency tradeoffs. In *PODC*, 2012.

[24] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, 2004.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *SOSP*, 2011.

[26] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[27] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC*, 2002.

[28] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[29] A. Rito da Silva et al. Project fenix applications and information systems of instituto superior tcnico. https://fenix-cvs.ist.utl.pt, 2012.

[30] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 2005.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 1990.

[32] E. Schurman and J. Brutlag. Performance relatedchanges and their user impact. Presented at velocity web performance and operations conference, 2009.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[34] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI*, 2009.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[36] D. Stocker. Delta transactions. http://collectiveweb.wordpress.com/2010/03/01/delta-transactions/, 2010.

[37] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[38] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[39] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 1996.

[40] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 1988.

[41] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.

# SymDrive: Testing Drivers without Devices

Matthew J. Renzelmann, Asim Kadav and Michael M. Swift
Computer Sciences Department, University of Wisconsin–Madison
{mjr,kadav,swift}@cs.wisc.edu

## Abstract

Device-driver development and testing is a complex and error-prone undertaking. For example, testing error-handling code requires simulating faulty inputs from the device. A single driver may support dozens of devices, and a developer may not have access to any of them. Consequently, many Linux driver patches include the comment "compile tested only."

SymDrive is a system for testing Linux and FreeBSD drivers without their devices present. The system uses symbolic execution to remove the need for hardware, and extends past tools with three new features. First, SymDrive uses static-analysis and source-to-source transformation to greatly reduce the effort of testing a new driver. Second, SymDrive checkers are ordinary C code and execute in the kernel, where they have full access to kernel and driver state. Finally, SymDrive provides an execution-tracing tool to identify how a patch changes I/O to the device and to compare device-driver implementations. In applying SymDrive to 21 Linux drivers and 5 FreeBSD drivers, we found 39 bugs.

## 1 Introduction

Device drivers are critical to operating-system reliability, yet are difficult to test and debug. They run in kernel mode, which prohibits the use of many runtime program-analysis tools available for user-mode code, such as Valgrind [34]. Their need for hardware can prevent testing altogether: over two dozen driver Linux and FreeBSD patches include the comment "compile tested only," indicating that the developer was unable or unwilling to run the driver. Even with hardware, it is difficult to test error-handling code that runs in response to a device error or malfunction. Thorough testing of failure-handling code is time consuming and requires exhaustive fault-injection tests with a range of faulty inputs.

Complicating matters, a single driver may support dozens of devices with different code paths. For example, one of the 18 supported medium access controllers in the E1000 network driver requires an additional EEPROM read operation while configuring flow-control and link settings. Testing error handling in this driver requires the specific device, and consideration of its specific failure modes.

Static analysis tools such as Coverity [17] and Microsoft's Static Driver Verifier [31] can find many bugs quickly. However, these tools are tuned for fast, relatively shallow analysis of large amounts of code and therefore only approximate the behavior of some code features, such as pointers. Furthermore, they have difficulty with bugs that span multiple invocations of the driver. Hence, static analysis misses large aspects of driver behavior.

We address these challenges using *symbolic execution* to test device drivers. This approach executes driver code on all possible device inputs, allows driver code to execute without the device present, and provides more thorough coverage of driver code, including error handling code. DDT [26] and S$^2$E [14, 15] previously applied symbolic execution to driver testing, but these systems require substantial developer effort to test new classes of drivers and, in many cases, even specific new drivers.

This paper presents *SymDrive*, a system to test Linux and FreeBSD drivers without devices. SymDrive uses static analysis to identify key features of the driver code, such as entry-point functions and loops. With this analysis, SymDrive produces an instrumented driver with call-outs to test code that allows many drivers to be tested with no modifications. The remaining drivers require a few annotations to assist symbolic execution at locations that SymDrive identifies.

We designed SymDrive for three purposes. First, a driver developer can use SymDrive to test driver patches by thoroughly executing all branches affected by the code changes. Second, a developer can use SymDrive as a debugging tool to compare the behavior of a functioning driver against a non-functioning driver. Third, SymDrive can serve as a general-purpose bug-finding tool and perform broad testing of many drivers with little developer input.

SymDrive is built with the S$^2$E system by Chipounov et al. [14, 15] , which can make any data within a virtual machine symbolic and explore its effect. SymDrive makes device inputs to the driver symbolic, thereby eliminating the need for the device and allowing execution on the complete range of device inputs. In addition, S$^2$E enables SymDrive to further enhance code coverage by mak-

ing other inputs to the driver symbolic, such as data from the applications and the kernel. When it detects a failure, either through an invalid operation or an explicit check, SymDrive reports the failure location and inputs that trigger the failure.

SymDrive extends S$^2$E with three major components. First, SymDrive uses *SymGen*, a static-analysis and code transformation tool, to analyze and instrument driver code before testing. SymGen automatically performs nearly all the tasks previous systems left for developers, such as identifying the driver/kernel interface, and also provides hints to S$^2$E to speed testing. Consequently, little effort is needed to apply SymDrive to additional drivers, driver classes, or buses. As evidence, we have applied SymDrive to eleven classes of drivers on five buses in two operating systems.

Second, SymDrive provides a *test framework* that allows *checkers* that validate driver behavior to be written as ordinary C code and execute in the kernel. These checkers have access to kernel state and the parameters and results of calls between the driver and the kernel. A checker can make pre- and post-condition assertions over driver behavior, and raise an error if the driver misbehaves. Using bugs and kernel programming requirements culled from code, documentation, and mailing lists, we wrote 49 checkers comprising 564 lines of code to enforce rules that maintainers commonly check during code reviews, such as matched allocation/free calls across entry points, no memory leaks, and proper use of kernel APIs.

Finally, SymDrive provides an *execution-tracing* mechanism for logging the path of driver execution, including the instruction pointer and stack trace of every I/O operation. These traces can be used to compare execution across different driver revisions and implementations. For example, a developer can debug where a buggy driver diverges in behavior from a previous working one. We have also used this facility to compare driver implementations across operating systems.

We demonstrate SymDrive's value by applying it to 26 drivers, and find 39 bugs, including two security vulnerabilities. We also find two driver/device interface violations when comparing Linux and FreeBSD drivers. To the best of our knowledge, no symbolic execution tool has examined as many drivers. In addition, SymDrive achieved over 80% code coverage in most drivers, and is largely limited by the ability of user-mode tests to invoke driver entry points. When we use SymDrive to execute code changed by driver patches, SymDrive achieves over 95% coverage on 12 patches in 3 drivers.

## 2 Motivation

The goal of our work is to improve driver quality through thorough testing and validation. To be successful, SymDrive must demonstrate (i) usefulness, (ii) simplicity, and (iii) efficiency. First, SymDrive must be able to find bugs that are hard to find using other mechanisms, such as normal testing or static analysis tools. Second, SymDrive must require low developer effort to test a new driver and therefore support many device classes, buses, and operating systems. Finally, SymDrive must be fast enough to apply to every patch.

### 2.1 Symbolic Execution

SymDrive uses symbolic execution to execute device-driver code without the device being present. Symbolic execution allows a program's input to be replaced with a *symbolic value*, which represents all possible values the data may have. A *symbolic-execution engine* runs the code and tracks which values are symbolic and which have *concrete* (*i.e.*, fully defined) values, such as initialized variables. When the program compares a symbolic value, the engine forks execution into multiple *paths*, one for each outcome of the comparison. It then executes each path with the symbolic value constrained by the chosen outcome of the comparison. For example, the predicate $x > 5$ forks execution by copying the running program. In one copy, the code executes the path where $x \leq 5$ and the other executes the path where $x > 5$. Subsequent comparisons can further constrain a value. In places where specific values are needed, such as printing a value, the engine can concretize data by producing a single value that satisfies all constraints over the data.

Symbolic execution detects bugs either through illegal operations, such as dereferencing a null pointer, or through explicit assertions over behavior, and can show the state of the executing path at the failure site.

**Symbolic execution with S$^2$E.** SymDrive is built on a modified version of the S$^2$E symbolic execution framework. S$^2$E executes a complete virtual machine as the program under test. Thus, symbolic data can be used anywhere in the operating system, including drivers and applications. S$^2$E is a virtual machine monitor (VMM) that tracks the use of symbolic data within an executing virtual machine. The VMM tracks each executing path within the VM, and schedules CPU time between paths. Each path is treated like a thread, and the scheduler selects which path to execute and when to switch execution to a different path.

S$^2$E supports *plug-ins*, which are modules loaded into the VMM that can be invoked to record information or to modify execution. SymDrive uses plugins to implement symbolic hardware, path scheduling, and code-coverage monitoring.

### 2.2 Why Symbolic Execution?

Symbolic execution is often used to achieve high coverage of code by testing on all possible inputs. For device drivers, symbolic execution provides an additional bene-

fit: executing without the device. Unlike most code, driver code can not be loaded and executed without its device present. Furthermore, it is difficult to force the device to generate specific inputs, which makes it difficult to thoroughly test error handling.

Symbolic execution eliminates the hardware requirement, because it can use symbolic data for all device input. An alternate approach is to code a software model of the device [33], which allows more accurate testing but greatly increases the effort required. In contrast, symbolic execution uses the driver itself as a model of device behavior: any device behavior used by the driver will be exposed as symbolic data.

Symbolic execution may provide inputs that correctly functioning devices may not. However, because hardware can provide unexpected or faulty driver input [25], this unconstrained device behavior is reasonable: drivers should not crash simply because the device provided an errant value.

In comparison to static analysis tools, symbolic execution provides several benefits. First, it uses existing kernel code as a model of kernel behavior rather than requiring a programmer-written model. Second, because driver and kernel code actually execute, it can reuse kernel debugging facilities, such as deadlock detection, and existing test suites. Thus, many bugs can be found without any explicit description of correct driver behavior. Third, symbolic execution can invoke a sequence of driver entry points, which allows it to find bugs that span invocations, such as resource leaks. In contrast, most static analysis tools concentrate on bugs occurring within a single entry point.

### 2.3 Why not Symbolic Execution?

While symbolic execution has previously been applied to drivers with DDT and S$^2$E, there remain open problems that preclude its widespread use:

**Efficiency.** The engine creates a new path for every comparison, and branchy code may create hundreds or thousands of paths, called *path explosion*. This explosion can be reduced by distinguishing and prioritizing paths that complete successfully. This approach enables executing deeper into the driver: if driver initialization fails, the operating system could not otherwise invoke most driver entry points. S$^2$E and DDT require complex, manually written annotations to provide this information. These annotations depend on kernel function names and behavioral details, which are difficult for programmers to provide. For example, the annotations often examine kernel function parameters, and modify the memory of the current path on the basis of the parameters. The path-scheduling strategies in DDT and S$^2$E favor exploring new code, but may not execute far enough down a path to test all functionality.

**Simplicity.** Existing symbolic testing tools require extensive developer effort to test a single class of drivers, plus additional effort to test each individual driver. For example, supporting Windows NDIS drivers in S$^2$E requires over 1,000 lines of code specific to this driver class. For example, the S$^2$E wrapper for the `NdisReadConfiguration` Windows function consists of code to (a) read all of the call's parameters, which is not trivial because the code is running outside the kernel, (b) fork additional paths for different possible symbolic return codes, (c) bypass the call to the function along these additional paths, and (d) register a separate wrapper function, of comparable complexity, to execute when this call returns. Since developers need to implement similarly complex code for many other functions in the driver/kernel interface, testing many drivers becomes impractical in these systems. Thus, these tools have only been applied to a few driver classes and drivers. Expanding testing to many more drivers requires new techniques to automate the testing effort.

**Specification.** Finally, symbolic execution by itself does not provide any specification of correct behavior: a "hello world" driver does nothing wrong, nor does it do anything right, such as registering a device with the kernel. In existing tools, tests must be coded like debugger extensions, with calls to read and write remote addresses, rather than as normal test code. Allowing developers to write tests in the familiar kernel environment simplifies the specification of correct behavior.

Thus, our work focuses on improving the state of the art to greatly simplify the use of symbolic execution for testing, and to broaden its applicability to almost any driver in any class on any bus.

## 3 Design

The SymDrive architecture focuses on thorough testing of drivers to ensure the code does not incorrectly use the kernel/driver interface, crash, or hang. We target test situations where the driver code is available, and use that code to simplify testing with a combination of symbolic execution, static code analysis and transformation, and an extensible test frameworkexecuting in the kernel.

The design of SymDrive is shown in Figure 1. The OS kernel and driver under test, as well as user-mode test programs, execute in a virtual machine. The symbolic execution engine provides symbolic devices for the driver. SymDrive provides stubs that invoke checkers on every call into or out of the driver. A test framework tracks execution state and passes information to plugins running in the engine to speed testing and improve test coverage.

During the development of SymDrive, we considered a more limited design in which symbolic execution was limited to driver code. In this model, exploring multiple paths through the kernel was not possible; callbacks to

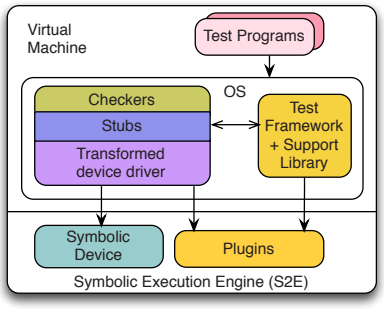



Figure 1: **The SymDrive architecture. A developer produces the transformed driver with SymGen and can write checkers and test programs to verify correctness.**

| Component | LoC |
|---|---|
| Changes to S$^2$E | 1,954 |
| SymGen | 2,681 |
| Test framework | 3,002 |
| Checkers | 564 |
| Support Library | 1,579 |
| Linux kernel changes | 153 |
| FreeBSD kernel changes | 81 |

Table 1: **Implementation size of SymDrive.**

the kernel instead required a model of kernel behavior to allow them to execute on multiple branches. After implementing a prototype of this design, we concluded that full-system symbolic execution is preferable because it greatly reduces the effort to test drivers by using real kernel code rather than a kernel model.

We implemented SymDrive for Linux and FreeBSD, as these kernels provide a large number of drivers to test. Only the test framework code running in the kernel is specialized to the OS. We made small, conditionally compiled changes to both kernels to print failures and stack traces to the S$^2$E log and to register the module under test with S$^2$E. The breakdown of SymDrive code is shown in Table 1.

SymDrive consists of five components: (i) a modified version of the S$^2$E symbolic-execution engine, which consists of a SymDrive-specific plugin plus changes to S$^2$E; (ii) symbolic devices to provide symbolic hardware input to the driver; (iii) a *test framework* executing within the kernel that guides symbolic execution; (iv) the *SymGen* static-analysis and code transformation tool to analyze and prepare drivers for testing; and (v) a set of OS-specific *checkers* that interpose on the driver/kernel interface for verifying and validating driver behavior.

### 3.1 Virtual Machine

SymDrive uses S$^2$E [15] version 1.1-10.09.2011, itself based on QEMU [7] and KLEE [10], for symbolic execution. S$^2$E provides the execution environment, path forking, and constraint solving capability necessary for symbolic execution. All driver and kernel code, including the test framework, executes within an S$^2$E VM. Changes to S$^2$E fall into two categories: (i) improved support for symbolic hardware, and (ii) the SymDrive path-selection mechanism, which is an S$^2$E plugin.

SymDrive uses invalid x86 opcodes for communication with the VMM and S$^2$E plugins to provide additional control over the executing code. We augment S$^2$E with new opcodes for the test framework that pass information into our extensions. These opcodes are inserted into driver code by SymGen and also invoked directly by the test framework.

The purpose of the opcodes is to communicate source-level information to the SymDrive plugins, which uses the information to guide the driver's execution. The opcodes (i) control whether memory regions are symbolic, as when mapping data for DMA; (ii) influence path scheduling by adjusting priority, search strategy, or killing other paths; and (iii) support tracing by turning it on/off and providing stack information.

### 3.2 Symbolic Devices

Drivers interact with devices according to well-defined, narrow interfaces. For PCI device drivers, this interface is comprised of I/O memory accessed via normal loads and stores, port I/O instructions, bus operations, DMA memory, and interrupts. Drivers using other buses, such as SPI and I$^2$C, use functions provided by the bus for similar operations.

SymDrive provides a symbolic device for the driver under test, while at the same time emulating the other devices in the system. A symbolic device provides three key behaviors. First, it can be *discovered*, so the kernel loads the appropriate driver. Second, it provides methods to read from and write to the device and return symbolic data from reads. Third, it supports interrupts and DMA, if needed. SymDrive currently supports 5 buses on Linux: PCI (and its variants), I$^2$C (including SMBus), Serial Peripheral Interface (SPI), General Purpose I/O (GPIO), and Platform;[1] and the PCI bus on FreeBSD.

**Device Discovery.** When possible, SymDrive creates symbolic devices in the S$^2$E virtual machine and lets existing bus code discover the new device and load the appropriate driver. For some buses, such as I$^2$C, the kernel or another driver normally creates a statically configured device object during initialization. For such devices, we created a small kernel module, consisting of 715 lines of code, that creates the desired symbolic device.

SymDrive can make the device configuration space symbolic after loading the driver by returning symbolic data from PCI bus functions with the test frame-

[1] The "platform bus" is a Linux-specific term that encompasses many embedded devices. Linux's ARM implementation, for example, supports a variety of SoCs, each with its own set of integrated devices. The drivers for these devices are often of the "platform" type.

work(similar to S$^2$E's NDIS driver support). PCI devices use this region of I/O memory for plug-and-play information, such as the vendor and device identifiers. If this data is symbolic, the device ID will be symbolic and cause the driver to execute different paths for each of its supported devices. Other buses have similar configuration data, such as "platform data" on the SPI bus. A developer can copy this data from the kernel source and provide it when creating the device object, or make it symbolic for additional code coverage.

**Symbolic I/O.** Most Linux and FreeBSD drivers do a mix of programmed I/O and DMA. SymDrive supports two forms of programmed I/O. For drivers that perform I/O through hardware instructions, such as `inb`, or through memory-mapped I/O, SymDrive directs S$^2$E to ignore write operations, because they do not return values that influence driver execution, and to return symbolic data from reads. The test framework overrides bus I/O functions, such as those used in I$^2$C drivers, to function analogously.

**Symbolic Interrupts.** After a driver registers an interrupt handler, the test framework invokes the interrupt handler on every transition from the driver into the kernel. This model represents a trade-off between realism and simplicity: it ensures the interrupt handler is called often enough to keep the driver executing successfully, but may generate spurious interrupts when the driver does not expect them.

**Symbolic DMA.** When a driver invokes a DMA mapping function, such as `dma_alloc_coherent`, the test framework uses a new S$^2$E opcode to make the memory act like a memory-mapped I/O region: each read returns a new symbolic value and writes have no effect. Discarding writes to DMA memory reflects the ability of the device to write the data via DMA at any time. The driver should not assume that data written here will be available subsequently. When the driver unmaps the memory, the test framework directs S$^2$E to revert the region to normal symbolic data, so writes are seen by subsequent reads.

### 3.3 Test Framework

The test framework is a kernel module executing with the virtual machine that assists symbolic execution and executes checkers. SymDrive relies on the test framework to guide and monitor symbolic execution in three ways. First, the test framework implements policy regarding which paths to prioritize or deprioritize. Second, the test framework may inject additional symbolic data to increase code coverage. As mentioned above, it implements symbolic I/O interfaces for some device classes. Finally, it provides the VMM with a stack trace for *execution tracing*, which produces a trace of the driver's I/O operations.

The test framework supports several load-time parameters for controlling its behavior. When loading the test framework with `insmod` or FreeBSD's `kldload`, developers can direct the test framework to enable high-coverage mode (described in Section 3.3.1), tracing, or a specific symbolic device. To configure the device, developers pass the device's I/O capabilities and name as parameters. Thus, developers can script the creation of symbolic devices to automate testing.

SymDrive has to address two conflicting goals in testing drivers: (i) executing as far as possible along a path to complete initialization and expose the rest of the driver's functionality; and (ii) executing as much code as possible within each function for thoroughness.

#### 3.3.1 Reaching Deeply

A key challenge in fully testing drivers is symbolically executing branch-heavy code, such as loops and initialization code that probes hardware. SymDrive relies on two techniques to limit path explosion in these cases: *favor-success scheduling* and *loop elision*. These techniques allow developers to execute further into a driver, and test functionality that is only available after initialization.

**Favor-success scheduling.** Executing past driver initialization is difficult because the code often has many conditionals to support multiple chips and configurations. Initializing a sound driver, for example, may execute more than 1,000 branches on hardware-specific details. Each branch creates additional paths to explore.

SymDrive mitigates this problem with a *favor-success* path-selection algorithm that prioritizes successfully executing paths, making it a form of best-first search. Notifications from the test framework increase the priority of the current path at every successful function return, both within the driver and at the driver/kernel interface. Higher priority causes the current path to be explored further before switching to another path. This strategy works best for small functions, where a successful path through the function is short.

At every function exit, the test framework notifies S$^2$E of whether the function completed successfully, which enables the VMM to prioritize successful paths to facilitate deeper exploration of code. The test framework determines success based on the function's return value. For functions returning integers, the test framework detects success when the function returns a value other than an `errno`, which are standard Linux and FreeBSD error values. On success, the test framework will notify the VMM to prioritize the current path. If the developer wishes to prioritize paths using another heuristic, he/she can add an annotation prioritizing any piece of code. We use this approach in some network drivers to select paths where the carrier is on, which enables execution of the driver's packet transmission code.

In order to focus symbolic execution on the driver, the

test framework prunes paths when control returns to the kernel successfully. It kills all other paths still executing in the driver and uses an opcode to concretize all data in the virtual machine, so the kernel executes on real values and will not fork new paths. This ensures that a single path runs in the kernel and allows developers to interact with the system and run user-mode tests.

**Loop elision.** Loops are challenging for symbolic execution because each iteration may fork new paths. S$^2$E provides an "EdgeKiller" plugin that a developer may use to terminate complex loops early, but requires developers to identify each loop's offset in the driver binary [15] and hence incur substantial developer effort.

SymDrive addresses loops explicitly by prioritizing paths that exit the loop quickly. Suppose an execution path *A* enters a loop, executes it once, and during this iteration more paths are created. If path *A* does not exit the loop after one iteration, SymDrive executes it for a second iteration and, unless it breaks out early, deprioritizes the second iteration because it appears stuck in the loop. SymDrive then selects some other path *B* that path *A* forked, and executes it. SymDrive repeats this process until it finds a path that exits the loop. If no paths exit the loop promptly, SymDrive selects some path arbitrarily and prioritizes it on each subsequent iteration, in the hope that it will exit the loop eventually. If this path still does not exit the loop within 20 iterations, SymDrive prints a warning about excessive path forking as there is no evident way to execute the loop efficiently without manual annotation.

This approach executes hardware polling loops efficiently and automatically, and warns developers when loops cause performance problems. However, this approach may fail if a loop is present in uninstrumented kernel code. It can also result in worse coverage of code that executes only if a polling loop times out. Moreover, loops that produce a value, such as a checksum calculation, cannot exit early without stopping the driver's progress. However, we have not found these problems to be significant.

SymDrive's approach extends the EdgeKiller plugin in two directions. First, it allows developers to annotate driver source rather than having to parse compiled code. Second, source annotations persist across driver revisions, whereas the binary offsets used in the EdgeKiller plugin need updating each time the driver changes.

Annotating code manually to improve its execution performance does reduce SymDrive's ability to find bugs in that code. Wherever annotations were needed in the drivers we examined, we strove to write them in such a way as to execute the problematic loop at least once before terminating early. For example, after a checksum loop, we would add a line to return a symbolic checksum value, which could then be compared against a correct one.

### 3.3.2 Increasing Coverage

SymDrive provides a *high-coverage* mode for testing specific functions, for example those modified by a patch. This mode changes the path-prioritization policy and the behavior of kernel functions. When the developer loads the test framework module, he/she can specify any driver function to execute in this mode.

When execution enters the specified function, the test framework notifies S$^2$E to favor unexecuted code (the default S$^2$E policy) rather than favoring successful paths. The test framework terminates all paths that return to the kernel in order to focus execution within the driver. In addition, when the driver invokes a kernel function, the test framework makes the return value symbolic. This mode is similar to the local consistency mode in S$^2$E [15], but requires no developer-provided annotations or plugins, and supports all kernel functions that return standard error values. For example, `kmalloc` returns a symbolic value constrained to be either NULL or a valid address, which tests error handling in the driver.

For the small number of kernel functions that return non-standard values, SymGen has a list of exceptions and how to treat their return values. The full list of exceptions for Linux currently contains 100 functions across all supported drivers. Of these, 64 are hardware-access functions, such as `inb` and `readl`, that always return symbolic data. A further 14 are arithmetic operations, such as `div32`. The remaining 22 functions return negative numbers in successful cases, or are used by the compiler to trigger a compilation failure when used incorrectly, such as `__bad_percpu_size`.

SymDrive also improves code coverage by introducing additional symbolic data in order to execute code that requires specific inputs from the kernel or applications. SymDrive can automatically make a Linux driver's module parameters symbolic, executes the driver with all possible parameters. Checkers can also make parameters to the driver symbolic, such as `ioctl` command values. This allows all `ioctl` code to be tested with a single invocation of the driver, because each comparison of the command will fork execution. In addition, S$^2$E allows using symbolic data anywhere in the virtual machine, so a user-mode test can pass symbolic data to the driver.

### 3.3.3 Execution Tracing

The test framework can generate execution traces, which are helpful to compare the execution of two versions of a driver. For example, when a driver patch introduces new bugs, the traces can be used to compare its behavior against previous versions. In addition, developers can use other implementations of the driver, even from another operating system, to find discrepancies that may signify incorrect interaction with the hardware.

A developer can enable tracing via a command-line tool

that uses a custom opcode to notify SymDrive to begin recording. In this mode, an S$^2$E plugin records every driver I/O operation, including reads and writes to port, MMIO, and DMA memory, and the driver stack at the operation. The test framework passes the current stack to S$^2$E on every function call.

The traces are stored as a trie (prefix tree) to represent multiple paths through the code compactly, and can be compared using the `diff` utility. SymDrive annotates each trace entry with the driver call stack at the I/O operation. This facilitates analysis of specific functions and comparing drivers function-by-function, which is useful since traces are subject to timing variations and different thread interleavings.

### 3.4 SymGen

All features of the test framework that interact with code, such as favor-success scheduling, loop prioritization, and making kernel return values symbolic are handled automatically via static analysis and code generation. The SymGen tool analyzes driver code to identify code relevant to testing, such as function boundaries and loops, and instruments code with calls to the test framework and checkers. SymGen is built using CIL [32].

**Stubs.** SymDrive interposes on all calls into and out of the driver with stubs that call the test framework and checkers. For each function in the driver, SymGen generates two stubs: a preamble, invoked at the top of the function, and a postscript, invoked at the end. The generated code passes the function's parameters and return value to these stubs to be used by checkers. For each kernel function the driver imports, SymGen generates a stub function with the same signature that wraps the function.

To support pre- and post-condition assertions, stubs invoke checkers when the kernel calls into the driver or the driver calls into the kernel. Checkers associated with a specific function `function_x` are named `function_x_check`. On the first execution of a stub, the test framework looks for a corresponding checker in the kernel symbol table. If such a function exists, the stub records its address for future invocations. While targeted at functions in the kernel interface, this mechanism can invoke checkers for any driver function.

Stubs employ a second lookup to find checkers associated with a function pointer passed from the driver to the kernel, such as a PCI probe function. Kernel stubs, when passed a function pointer, record the function pointer and its purpose in a table. For example, the Linux `pci_register_driver` function associates the address of each function in the `pci_driver` parameter with the name of the structure and the field containing the function. The stub for the `probe` method of a `pci_driver` structure is thus named `pci_driver_probe_check`. FreeBSD drivers use a similar technique.

```
s2e_loop_before(__LINE__, loop_id);
while(work--) {
  tmp___17 = readb(cp->regs + 55);
  if(!(tmp___17 & 16)) goto return_label;
  stub_schedule_timeout_uninterruptible(10L);
  s2e_loop_body(__LINE__, loop_id);
}
s2e_loop_after(__LINE__, loop_id);
```

Figure 2: **SymGen instruments the start, end, and body of loops automatically. This code, from the `8139cp` driver, was modified slightly since SymGen produces preprocessed output.**

Stubs detect that execution enters the driver by tracking the depth of the call stack. The first function in the driver notifies the test framework at its entry that driver execution is starting, and at its exit notifies the test framework that control is returning to the kernel. Stubs also communicate this information to the VMM so that it can make path-scheduling decisions based on function return values.

**Instrumentation.** The underlying principle behind SymGen's instrumentation is to inform the VMM of source level information as it executes the driver so it can make better decisions about which paths to execute. SymGen instruments the start and end of each driver function with a call into the stubs. As part of the rewriting, it converts functions to have a single exit point. It generates the same instrumentation for inline functions, which are commonly used in the Linux and FreeBSD kernel/driver interfaces.

SymGen also instruments the start, end, and body of each loop with calls to short functions that execute SymDrive-specific opcodes. These opcodes direct the VMM to prioritize and deprioritize paths depending on whether they exit the loop quickly. This instrumentation replaces most of the per-driver effort required by S$^2$E to identify loops, as well as the per-class effort of writing a consistency model for every function in the driver/kernel interface. SymGen also inserts loop opcodes into the driver, as shown in Figure 2, to tell S$^2$E which paths exit the loop, and should receive a priority boost.[2]

For complex code that slows testing, SymGen supports programmer-supplied annotations to simplify or disable the code temporarily. Short loops and those that do not generate states require no manual developer effort. Only loops that must execute for many iterations and generate new paths on each iteration need manual annotation, which we implement through C `#ifdef` statements. For example, the E1000 network driver verifies a checksum over EEPROM, and we modified it to accept any checksum value. We have found these cases to be rare.

[2] One interesting alternative is to prioritize paths that execute loops in their entirety. The problem with this approach is that it may generate many states in the process, and slow testing.

### 3.5 Limitations

SymDrive is neither sound nor complete. The false positives we have experienced fall into two major categories. First, symbolic execution is slow, which may cause the kernel to print timing warnings and cause driver timers to fire at the wrong time. Second, our initial checkers were imprecise and disallowed (bizarre) behavior the kernel considers legal. We have since fixed the checkers, and have not seen them generate false positives.

Although we have observed no false negatives among the checkers we wrote, SymDrive cannot check for all kinds of bugs. Of 11 common security vulnerabilities [12], SymDrive cannot detect integer overflows and data races between threads, though support for overflow detection is possible in principle because the underlying VMM interprets code rather than executing it directly. In addition, SymDrive cannot achieve full path coverage for all drivers because SymDrive's aggressive path pruning may terminate paths that lead to bugs. SymDrive may also miss race conditions, such as those requiring the interrupt handler to interleave with another thread in a specific way.

## 4 Checkers

SymDrive detects driver/kernel interface violations with checkers, which are functions interposing on control transfer between the driver and kernel that verify and validate driver behavior. Each function in the driver/kernel interface can, but need not, have its own checker. Drivers invoke the checkers from stubs, described above, which call separate checkers at every function in the driver/kernel interface. Since checkers run in the VM alongside the symbolically executing driver, they can verify runtime properties along each tested path.

The checkers use a *support library* that simplifies their development by providing much of their functionality. The library provides state variables to track the state of the driver and current thread, such as whether it has registered itself successfully and whether it can be rescheduled. The library also provides an object tracker to record kernel objects currently in use in the driver. This object tracker provides an easy mechanism to track whether locks have been initialized and to discover memory leaks. Finally, the library provides generic checkers for common classes of kernel objects, such as locks and allocators. The generic checkers encode the semantics of these objects, and thus do much of the work. For example, checkers for a mutex lock and a spin lock use the same generic checker, as they share semantics.

Writing a checker requires implementing checks within a call-out function. We have implemented 49 checkers comprising 564 lines of code for a variety of common device-driver bugs using the library API. Test #1 in Figure 3 shows an example call-out for `pci_register_driver`. The driver-function stub invokes the checker function with the parameters and return value of the kernel function and sets a `precondition` flag to indicate whether the checker was called before or after the function. In addition, the library provides the global `state` variable that a checker can use to record information about the driver's activity. As shown in this example, a checker can verify that the state is correct as a precondition, and update the state based on the result of the call. Checkers have access to the runtime state of the driver and can store arbitrary data, so they can find interprocedural, pointer-specific bugs that span multiple driver invocations.

```
/* Test #1 */ void __pci_register_driver_check(...) {
  if (precondition) {
    assert (state.registered == NOT_CALLED);
    set_state (&state.registered, IN_PROGRESS);
    set_driver_bus (DRIVER_PCI);
  } else /* postcondition */ {
    if (retval == 0) set_state (&state.registered, OK);
    else set_state (&state.registered, FAILED);
  }
}

/* Test #2 */ void __kmalloc_check
  (..., void *retval, size_t size, gfp_t flags) {
  if (precondition)
    mem_flags_test(GFP_ATOMIC, GFP_KERNEL, flags);
  else /* postcondition */
    generic_allocator(retval, size, ORIGIN_KMALLOC);
}

/* Test #3 */ void _spin_lock_irqsave_check
  (..., void *lock) {
  // generic_lock_state supports pre/post-conditions
  generic_lock_state(lock,
    ORIGIN_SPIN_LOCK, SPIN_LOCK_IRQSAVE, 1);
}
```

Figure 3: **Example checkers. The first checker ensures that PCI drivers are registered exactly once. The second verifies that a driver allocates memory with the appropriate `mem_flags` parameter. The third ensures lock/unlock functions are properly matched. Unlike Static Driver Verifier checkers [31], these checkers can track any path-specific run-time state expressible in C.**

Not every behavior requirement needs a checker. Symbolic execution leverages the extensive checks already included as kernel debug options, such as for memory corruption and locking. Most of these checks execute within functions called *from* the driver, and thus will be invoked on multiple paths. In addition, any bug that causes a kernel crash or panic will be detected by the operating system and therefore requires no checker.

We next describe a few of the 49 checkers we have implemented with SymDrive.

**Execution Context.** Linux prohibits the calling of functions that block when executing in an interrupt handler or while holding a spinlock. The execution-context checker verifies that flags passed to memory-allocation functions such as `kmalloc` are valid in the context of the currently

executing code. The support library provides a state machine to track the driver's current context using a stack. When entering the driver, the library updates the context based on the entry point. The library also supports locks and interrupt management. When the driver acquires or releases a spinlock, for example, the library pushes or pops the necessary context.

**Kernel API Misuse.** The kernel requires that drivers follow the proper protocol for kernel APIs, and errors can lead to a non-functioning driver or a resource leak. The support library state variables provide context for these tests. For example, a checker can track the success and failure of significant driver entry points, such as the `init_module` and PCI `probe` functions, and ensure that if the driver is registered on initialization, it is properly unregistered on shutdown. Test #1 in Figure 3 shows a use of these states to ensure that a driver only invokes `pci_register_driver` once.

**Collateral Evolutions.** Collateral evolutions occur when a small change to a kernel interface necessitates changes in many drivers simultaneously. A developer can use SymDrive to verify that collateral evolutions [35] are correctly applied by ensuring that patched drivers do not regress on any tests.

SymDrive can also ensure that the desired *effect* of a patch is reflected in the driver's execution. For example, recent kernels no longer require that network drivers update the `net_device->trans_start` variable in their `start_xmit` functions. We wrote a checker to verify that `trans_start` is constant across `start_xmit` calls.

**Memory Leaks.** The leak checker uses the support library's object tracker to store an allocation's address and length. We implemented checkers to verify allocation and free requests from 19 pairs of functions, and ensure that an object's allocation and freeing use paired routines.

The API library simplifies writing checkers for additional allocators down to a few lines of code. Test #2 in Figure 3 shows the `generic_allocator` call to the library used when checking `kmalloc`, which records that `kmalloc` allocated the returned memory. A corresponding checker for `kfree` verifies that `kmalloc` allocated the supplied address.

# 5 Evaluation

The purpose of the evaluation is to verify that SymDrive achieves its goals: (i) usefulness, (ii) simplicity, and (iii) efficiency.

## 5.1 Methodology

As shown in Table 2, we tested SymDrive on 26 drivers in 11 classes from several Linux kernel revisions (13 drivers from 2.6.29, 4 from 3.1.1, and 4 that normally run only on Android-based phones) and from FreeBSD 9 (5 drivers). Of the 26 drivers, we chose 19 as examples of a specific bus or class and the remaining 7 because we found frequent patches to them and thus expected to find bugs.

All tests took place on a machine running Ubuntu 10.10 x64 equipped with a quad-core Intel 2.50GHz Intel Q9300 CPU and 8GB of memory. All results are obtained while running SymDrive in a single-threaded mode, as SymDrive does not presently work with S$^2$E's multicore support.[3]

To test each driver, we carry out the following operations:

1. Run SymGen over the driver and compile the output.
2. Define a virtual hardware device with the desired parameters and boot the SymDrive virtual machine.
3. Load the driver with `insmod` and wait for initialization to complete successfully. Completing this step entails executing at least one successful path and returning success, though it is likely that other failed paths also run and are subsequently discarded.
4. Execute a workload (optional). We ensure all network drivers attempt to transmit and that sound drivers attempt to play a sound.
5. Unload the driver.

If SymDrive reports warnings about too many paths from complex loops, we annotate the driver code and repeat the operations. For most drivers, we run SymGen over only the driver code. For drivers that have fine-grained interactions with a library, such as sound drivers and the `pluto2` media driver, we run SymGen over both the library and the driver code. We annotated each driver at locations SymDrive specified, and tested each Linux driver with 49 checkers for a variety of common bugs. For FreeBSD drivers, we only used the operating system's built-in test functionality.

## 5.2 Bug Finding

Across the 26 drivers listed in Table 2, we found the 39 distinct bugs described in Table 3. Of these bugs, S$^2$E detected 17 via a kernel warning or crash, and the checkers caught the remaining 22. Although these bugs do not necessarily result in driver crashes, they all represent issues that need addressing and are difficult to find without visibility into driver/kernel interactions.

These results show the value of symbolic execution. Of the 39 bugs, 56% spanned multiple driver invocations. For example, the `akm8975` compass driver calls `request_irq` before it is ready to service interrupts. If an interrupt occurs immediately after this call, the driver will crash, since the interrupt handler dereferences a pointer

[3]This limitation is an engineering issue and prevents SymDrive from exploring multiple paths simultaneously. However, because SymDrive's favor-success scheduling often explores a single path deeply rather than many paths at once, S$^2$E's multi-threaded mode would have little performance benefit.

| Driver | Class | Bugs | LoC | Ann | Load | Unld. |
|---|---|---|---|---|---|---|
| *akm8975** | Compass | 4 | 629 | 0 | 0:22 | 0:08 |
| *mmc31xx** | Compass | 3 | 398 | 0 | 0:10 | 0:04 |
| tle62x0* | Control | 2 | 260 | 0 | 0:06 | 0:05 |
| me4000 | Data Ac. | 1 | 5,394 | 2 | 1:17 | 1:04 |
| phantom | Haptic | 0 | 436 | 0 | 0:16 | 0:13 |
| lp5523* | LED Ctl. | 2 | 828 | 0 | 2:26 | 0:19 |
| apds9802* | Light | 0 | 256 | 1 | 0:31 | 0:21 |
| 8139cp | Net | 0 | 1,610 | 1 | 1:51 | 0:37 |
| 8139too | Net | 2 | 1,904 | 3 | 3:28 | 0:35 |
| be2net | Net | 7 | 3,352 | 2 | 4:49 | 1:39 |
| dl2k | Net | 1 | 1,985 | 5 | 2:52 | 0:35 |
| e1000 | Net | 3 | 13,971 | 2 | 4:29 | 2:01 |
| et131x | Net | 2 | 8,122 | 7 | 6:14 | 0:47 |
| forcedeth | Net | 1 | 5,064 | 2 | 4:28 | 0:51 |
| ks8851* | Net | 3 | 1,229 | 0 | 2:05 | 0:13 |
| pcnet32 | Net | 1 | 2,342 | 1 | 2:34 | 0:27 |
| *smc91x** | Net | 0 | 2,256 | 0 | 10:41 | 0:22 |
| pluto2 | Media | 2 | 591 | 3 | 1:45 | 1:01 |
| econet | Proto. | 2 | 818 | 0 | 0:11 | 0:11 |
| ens1371 | Sound | 0 | 2,112 | 5 | 27:07 | 4:48 |
| *a1026** | Voice | 1 | 1,116 | 1 | 0:34 | 0:03 |
| ed | Net | 0 | 5,014 | 0 | 0:49 | 0:13 |
| re | Net | 0 | 3,440 | 3 | 16:11 | 0:21 |
| rl | Net | 0 | 2,152 | 1 | 2:00 | 0:08 |
| es137x | Sound | 1 | 1,688 | 2 | 57:30 | 0:09 |
| maestro | Sound | 1 | 1,789 | 2 | 17:51 | 0:27 |

Table 2: **Drivers tested. Those in *italics* run on Android-based phones, those followed by an asterisk are for embedded systems and do not use the PCI bus. Drivers above the line are for Linux and below the line are for FreeBSD. Line counts come from CLOC [1]. Times are in minute:second format, and are an average of three runs.**

| Bug Type | Bugs | Kernel / Checker | Cross EntPt | Paths | Ptrs |
|---|---|---|---|---|---|
| Hardware Dep. | 7 | 6 / 1 | 4 | 6 | 6 |
| API Misuse | 15 | 7 / 8 | 6 | 5 | 1 |
| Race | 3 | 3 / 0 | 3 | 2 | 3 |
| Alloc. Mismatch | 3 | 0 / 3 | 3 | 0 | 3 |
| Leak | 7 | 0 / 7 | 6 | 1 | 7 |
| Driver Interface | 3 | 0 / 3 | 0 | 2 | 0 |
| Bad pointer | 1 | 1 / 0 | 0 | 0 | 1 |
| **Totals** | 39 | 17 / 22 | 22 | 16 | 21 |

Table 3: **Summary of bugs found. For each category, we present the number of bugs found by kernel crash/warning or a checker and the number that crossed driver entry points ("Cross EntPt"), occurred only on specific paths, or required tracking pointer usage.**

that is not yet initialized. In addition, 41% of the bugs occurred on a unique path through a driver other than one that returns success, and 54% involved pointers and pointer properties that may be difficult to detect statically.

**Bug Validation.** Of the 39 bugs found, at least 17 were fixed between the 2.6.29 and 3.1.1 kernels, which indicates they were significant enough to be addressed. We were unable to establish the current status of 7 others because of significant driver changes. We have submitted bug reports for 5 unfixed bugs in mainline Linux drivers, all of which have been confirmed as genuine by kernel developers. The remaining bugs are from drivers outside the main Linux kernel that we have not yet reported.

### 5.3 Developer Effort

One of the goals of SymDrive is to minimize the effort to test a driver. The effort of testing comes from three sources: (i) annotations to prepare the driver for testing, (ii) testing time, and (iii) updating code as kernel interfaces change.

To measure the effort of applying SymDrive to a new driver, we tested the `phantom` haptic driver from scratch. The total time to complete testing was 1h:45m, despite having no prior experience with the driver and not having the hardware. In this time, we configured the symbolic hardware, wrote a user-mode test program that passes symbolic data to the driver's entry points, and executed the driver four times in different configurations. Of this time, the overhead of SymDrive compared to testing with a real device was an additional pass during compilation to run SymGen, which takes less than a minute, and 38 minutes to execute. Although not a large driver, this test demonstrates SymDrive's usability from the developer's perspective.

**Annotations.** The only per-driver coding SymDrive requires is annotations on loops that slow testing and annotations that prioritize specific paths. Table 2 lists the number of annotation sites for each driver. Of the 26 drivers, only 6 required more than two annotations, and 9 required no annotations. In all cases, SymDrive printed a warning indicating where an annotation would benefit testing.

**Testing time.** Symbolic execution can be much slower than normal execution. Hence, we expect it to be used near the end of development, before submitting a patch, or on periodic scans through driver code. We report the time to load, initialize, and unload a driver (needed for detecting resource leaks) in Table 2. Initialization time is the minimum time for testing, and thus presents a lower bound.

Overall, the time to initialize a driver is roughly proportional to the size of the driver. Most drivers initialize in 5 minutes or less, although the `ens1371` sound driver required 27 minutes, and the corresponding FreeBSD `es137x` driver required 58 minutes. These two results stem from the large amount of device interaction these drivers perform during initialization. Excluding these results, execution is fast enough to be performed for every patch, and with a cluster could be performed on every driver affected by a collateral evolution [35].

**Kernel evolution.** Near the end of development, we upgraded SymDrive from Linux 2.6.29 to Linux 3.1.1. If much of the code in SymDrive was specific to the kernel interface, porting SymDrive would be a large effort. However, SymDrive's use of static analysis and code generation minimized the effort to maintain tests as the kernel evolves: the only changes needed were to update a

| Driver | Touched Funcs. | Coverage | Time CPU | Time Latency |
|---|---|---|---|---|
| 8139too | 93% | 83% | 2h36m | 1h00m |
| a1026 | 95% | 80% | 15m | 13m |
| apds9802 | 85% | 90% | 14m | 7m |
| econet | 51% | 61% | 42m | 26m |
| ens1371 | 74% | 60% | *8h23m | *2h16m |
| lp5523 | 95% | 83% | 21m | 5m |
| me4000 | 82% | 68% | *26h57m | *10h25m |
| mmc31xx | 100% | 83% | 14m | 26m |
| phantom | 86% | 84% | 38m | 32m |
| pluto2 | 78% | 90% | 19m | 6m |
| tle62x0 | 100% | 85% | 16m | 12m |
| es137x | 97% | 70% | 1h22m | 58m |
| rl | 84% | 71% | 13m | 10m |

Table 4: **Code coverage.**

few checkers whose corresponding kernel functions had changed. The remainder of the system, including SymGen and the test framework, were unchanged. The number of lines of code changed was less than 100.

Furthermore, porting SymDrive to a new operating system is not difficult. We also ported the SymDrive infrastructure, checkers excluded, to FreeBSD 9. The entire process took three person-weeks. The FreeBSD implementation largely shares the same code base as the Linux version, with just a few OS-specific sections. This result confirms that the techniques SymDrive uses are compatible across operating systems.

### 5.4 Coverage

While SymDrive primarily uses symbolic execution to simulate the device, a second benefit is higher code coverage than standard testing. Table 4 shows coverage results for one driver of each class, and gives the fraction of functions executed ("Touched Funcs.") and the fraction of basic blocks *within* those functions ("Coverage").[4] In addition, the table gives the total CPU time to run the tests on a single machine (CPU) and the latency of the longest run if multiple machines are used (Latency). In all cases, we ran drivers multiple times and merged the coverage results. We terminated each run once it reached a steady state and stopped testing the driver once coverage did not meaningfully improve between runs.

Overall, SymDrive executed a large majority (80%) of driver functions in most drivers, and had high coverage (80% of basic blocks) in those functions. These results are below 100% for two reasons. First, we could not invoke all entry points in some drivers. For example, `econet` requires user-mode software to trigger additional driver entry points that SymDrive is unable to call on its own. In other cases, we simply did not spend enough time understanding how to invoke all of a driver's code, as some functionality requires the driver to be in a specific state that is difficult to realize, even with symbolic execution.

[4] * Drivers with an asterisk ran unattended, and their total execution time is not representative of the minimum.

| Driver | Touched Funcs. | Coverage | Time Serial | Time Parallel |
|---|---|---|---|---|
| 8139too | 100% | 96% | 9m | 5m |
| ks8851 | 100% | 100% | 16m | 8m |
| lp5523 | 100% | 97% | 12m | 12m |

Table 5: **Patched code coverage.**

Second, of the functions SymDrive did execute, additional inputs or symbolic data from the kernel were needed to test all paths. Following $S^2E$'s relaxed consistency model by making more of the kernel API symbolic could help improve coverage.

As a comparison, we tested the `8139too` driver on a real network card using `gcov` to measure coverage with the same set of tests. We loaded and unloaded the driver, and ensured that transmit, receive, and all ethtool functions executed. Overall, these tests executed 77% of driver functions, and covered 75% of the lines in the functions that were touched, as compared to 93% of functions and 83% of code for SymDrive. Although not directly comparable to the other coverage results due to differing methodologies, this result shows that SymDrive can provide coverage better than running the driver on real hardware.

### 5.5 Patch Testing

The second major use of SymDrive is to verify driver patches similar to a code reviewer. For this use, we seek high coverage in every function modified by the patch in addition to the testing described previously. We evaluate SymDrive's support for patch testing by applying all the patches between the 3.1.1 and 3.4-rc6 kernel releases that applied to the `8139too` (net), `ks8851` (net) and `lp5523` (LED controller) drivers, of which there were 4, 2, and 6, respectively. The other drivers lacked recent patches, had only trivial patches, or required upgrading the kernel, so we did not consider them.

In order to test the functions affected by a patch, we used favor-success scheduling to fast-forward execution to a patched function and then enabled high coverage mode. The results, shown in Table 5, demonstrate that SymDrive is able to quickly test patches as they are applied to the kernel, by allowing developers to test nearly all the changed code without any device hardware. SymDrive was able to execute 100% of the functions touched by all 12 patches across the 3 drivers, and an average of 98% of the code in each function touched by the patch. In addition, tests took an average of only 12 minutes to complete.

**Execution tracing.** Execution tracing provides an alternate means to verify patches by comparing the behavior of a driver before and after applying the patch. We used tracing to verify that SymDrive can distinguish between patches that change the driver/device interactions and those that do not, such as a collateral evolution. We tested five patches to the `8139too` network driver that

refactor the code, add a feature, or change the driver's interaction with the hardware. We executed the original and patched drivers and record the hardware interactions. Comparing the traces of the before and after-patch drivers, differing I/O operations clearly identify the patches that added a feature or changed driver/device interactions, including which functions changed. As expected, there were no differences in the refactoring patches.

We also apply tracing to compare the behavior of drivers for the same device across operating systems. Traces of the Linux `8139too` driver and the FreeBSD `rl` driver show differences in how these devices interact with the same hardware that could lead to incorrect behavior. In one case, the Linux `8139too` driver incorrectly treats one register as 4 bytes instead of 1 byte, while in the other, the `rl` FreeBSD driver uses incorrect register offsets for a particular supported chipset. Developers fixed the Linux bug independently after we discovered it, and we validated the FreeBSD bug with a FreeBSD kernel developer. We do not include these bugs in the previous results as they were not identified automatically by SymDrive.

These bugs demonstrate a new capability to find hardware-specific bugs by comparing independent driver implementations. While we manually compared the traces, it may be possible to automate this process.

### 5.6 Comparison to other tools.

We compare SymDrive against other driver testing/bug-finding tools to demonstrate its usefulness, simplicity, and efficiency.

**S$^2$E.** In order to demonstrate the value of SymDrive's additions to S$^2$E, we executed the `8139too` driver with only annotations to the driver source guiding path exploration but without the test framework or SymGen to prioritize relevant paths. In this configuration, S$^2$E executes using *strict consistency*, wherein the only source of symbolic data is the hardware, and maximizes coverage with the MaxTbSearcher plugin. This mode is the default when a developer does not write API-specific plugins; results improve greatly when these plugins are available [15]. We ran S$^2$E until it ran out of memory to store paths and started thrashing after 23 minutes.

During this test, only 33% of the functions in the driver were executed, with an average coverage of 69%. In comparison, SymDrive executed 93% of functions with an average coverage of 83% in 2½ hours. With S$^2$E alone, the driver did not complete initialization and did not attempt to transmit packets. In addition, S$^2$E's annotations could not be made on the driver source, but must be made on the binary instead. Thus, annotations must be regenerated every time a driver is compiled.

Adding more RAM and running the driver longer would likely have allowed the driver to finish executing the initialization routine. However, many uninteresting paths would remain, as S$^2$E has no automatic way to prune them. Thus, the developer would still have considerable difficulty invoking other driver entry points, since S$^2$E would continue to execute failing execution paths.

In order for S$^2$E to achieve higher coverage in this driver, we would need a plugin to implement a relaxed consistency model. However, the `8139too` driver (v3.1.1) calls 73 distinct kernel functions, which would require developer effort to code corresponding functions in the plugin.

**Static-analysis tools.** Static analysis tools are able to find many driver bugs, but require a large effort to implement a model of operating system behavior. For example, Microsoft's Static Driver Verifier (SDV) requires 39,170 lines of C code to implement an operating system model [31]. SymDrive instead relies on models only for the I/O bus implementations, which together account for 715 lines of code for 5 buses. SymDrive supports FreeBSD with only 491 lines of OS-specific code, primarily for the test framework, and can check drivers with the debugging facilities already included in the OS.

In addition, SDV achieves much of its speed through simplifying its analysis, and consequently its checkers are unable to represent arbitrary state. Thus, it is difficult to check complex properties such as whether a variable has matched allocation/free calls across different entry points.

**Kernel debug support.** Most kernels provide debugging to aid kernel developers, such as tools to detect deadlock, track memory leaks, or uncover memory corruption. Some of the test framework checkers are similar to debug functionality built into Linux. Compared to the Linux leak checker, kmemleak, the test framework allows testing a single driver for leaks, which can be drowned out when looking at a list of leaks across the entire kernel. Furthermore, writing checkers for SymDrive is much simpler: the Linux 3.1.1 kmemleak module is 1,113 lines, while, the test framework object tracker, including a complete hash table implementation, is only 722 lines yet provides more precise results.

## 6 Related Work

SymDrive draws on past work in a variety of areas, including symbolic execution, static and dynamic analysis, test frameworks, and formal specification.

**DDT and S$^2$E.** The DDT and S$^2$E systems have been used for finding bugs in binary drivers [14, 15, 26]. SymDrive is built upon S$^2$E but significantly extends its capabilities in three ways by leveraging driver source code. First and most important, SymDrive automatically detects the driver/kernel interface and generates code to interpose checkers at that interface. In contrast, S$^2$E requires programmers to identify the interface manually and write plugins that execute *outside the kernel*, where kernel symbols

are not available, though $S^2E$ and SymDrive both support re-using existing testing tools. Second, SymDrive automatically detects and annotates loops, which in $S^2E$ must be identified manually and specified as virtual addresses. As a result, the effort to test a driver is much reduced compared to $S^2E$. Third, checkers in SymDrive are implemented as standard C code executing in the kernel, making them easy to write, and are only necessary for kernel functions of interest. When the kernel interface changes, only the checkers affected by interface changes must be modified. In contrast, checkers in $S^2E$ are written as plugins outside the kernel, and the consistency model plugins must be updated for all changed functions in the driver interface, not just those relevant to checks.

**Symbolic testing.** There are numerous prior approaches to symbolic execution [9, 10, 13, 20, 26, 39, 40, 43, 45]. However, most apply to standalone programs with limited environmental interaction. Drivers, in contrast, execute as a library and make frequent calls into the kernel. BitBlaze supports environment interaction but not I/O or drivers [37].

To limit symbolic execution to a manageable amount of state, previous work limited the set of symbolically executed paths by applying smarter search heuristics and/or by limiting program inputs [11, 21, 26, 27, 28, 44], which is similar to SymDrive's path pruning and prioritization.

Other systems combine static analysis with symbolic execution [16, 18, 19, 36]. SymDrive uses static analysis to insert checkers and to dynamically guide the path selection policy from code features such as loops and return values. In contrast, these systems use the output of static analysis directly within the symbolic execution engine to select paths. Execution Synthesis [45] combines symbolic execution with static analysis, but is designed to reproduce existing bug reports with stack traces, and is thus complementary to SymDrive.

**Static analysis tools.** Static analysis tools can find specific kinds of bugs common to large classes of drivers, such as misuses of the driver/kernel [3, 4, 5, 31, 35] or driver/device interface [25] and ignored error codes [23, 41]. Static bug-finding tools are often faster and more scalable than symbolic execution [8].

We see three key advantages of testing drivers with symbolic execution. First, symbolic execution is better able to find bugs that arise from multiple invocations of the driver, such as when state is corrupted during one call and accessed during another. It also has a low false-positive rate because it makes few approximations. Second, symbolic execution has full access to driver and kernel state, which facilitates checking driver behavior. Furthermore, checkers that verify behavior can be written as ordinary C, which simplifies their development, and can track arbitrary runtime state such as pointers and driver data. Symbolic execution also supports the full functionality of C including pointer arithmetic, aliasing, inline assembly code, and casts. In contrast, most static analysis tools operate on a restricted subset of the language. Thus, symbolic execution often leads to fewer false positives. Finally, static tools require a model of kernel behavior, which in Linux changes regularly [22]. In contrast, SymDrive executes checkers written in C and has no need for an operating system model, since it executes kernel code symbolically. Instead, SymDrive relies only on models for each I/O bus, which are much simpler and shorter to write.

**Test frameworks.** Test frameworks such as the Linux Test Project (LTP) [24] and Microsoft's Driver Verifier (DV) [29, 30] can invoke drivers and verify their behavior, but require the device be present. In addition, LTP tests at the system-call level and thus cannot verify properties of individual driver entry points. SymDrive can use these frameworks, either as checkers, in the case of DV, or as a test program, in the case of LTP.

**Formal specifications for drivers.** Formal specifications express a device's or a driver's operational requirements. Once specified, other parts of the system can verify that a driver operates correctly [6, 38, 42]. However, specifications must be created for each driver or device. Amani et al. argue that the existing driver architecture is too complicated to be formally specified, and propose a new architecture to simplify verification [2]. Many of the challenges to static verification also complicate symbolic testing, and hence their architecture would address many of the issues solved by SymDrive.

## 7 Conclusions

SymDrive uses symbolic execution combined with a test framework and static analysis to test Linux and FreeBSD driver code without access to the corresponding device. Our results show that SymDrive can find bugs in mature driver code of a variety of types, and allow developers to test driver patches deeply. Hopefully, SymDrive will enable more developers to patch driver code by lowering the barriers to testing. In the future, we plan to implement an automated testing service for driver patches that supplements manual code reviews, and investigate applying SymDrive's techniques to other kernel subsystems.

## Acknowledgments

This work is supported by the National Science Foundation grants CNS-0745517 and CNS-0915363 and by a gift from Google. We would like to thank the many reviewers who provided detailed feedback on our work, and for early feedback from Gilles Muller and Julia Lawall. We would also like to thank the $S^2E$ developers, who provided us a great platform to build on. Swift has a significant fi-

nancial interest in Microsoft.

## References


[1] Al Danial. CLOC: Count lines of code. http://cloc.sourceforge.net/, 2010.

[2] S. Amani, L. Ryzhyk, A. Donaldson, G. Heiser, A. Legg, and Y. Zhu. Static analysis of device drivers: We can do better! In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, et al. Thorough static analysis of device drivers. In *EuroSys*, 2006.

[4] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. In *Commun. of the ACM*, volume 54, July 2011.

[5] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. In *Commun. of the ACM*, volume 54, June 2011.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, 2005.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[9] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. In *Intl. Conf. on Reliable Software*, 1975.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[11] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. *In ACM Transactions on Information and System Security*, 2008.

[12] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

[13] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *HotDep*, 2009.

[14] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[15] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[16] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.

[17] Coverity. Analysis of the Linux kernel, 2004. Available at http://www.coverity.com.

[18] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *EuroSys*, 2011.

[19] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[21] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, 2008.

[22] Greg Kroah-Hartman. The Linux kernel driver interface. http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt, 2011.

[23] H. Gunawi, C. Rubio-González, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *6th USENIX FAST*, 2008.

[24] IBM. Linux test project. http://ltp.sourceforge.net/, May 2010.

[25] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *SOSP*, 2009.

[26] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC*, 2010.

[27] E. Larson and T. Austin. High coverage detection of input-related security faculties. In *USENIX Security*, 2003.

[28] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[29] Microsoft. Windows device testing framework design guide. http://msdn.microsoft.com/en-us/library/windows/hardware/ff539645%28v=vs.85%29.aspx, 2011.

[30] Microsoft Corporation. How to use driver verifier to troubleshoot windows drivers. http://support.microsoft.com/kb/q244617/, Jan. 2005. Knowledge Base Article Q244617.

[31] Microsoft Corporation. Static Driver Verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, May 2010.

[32] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Constr.*, 2002.

[33] S. Nelson and P. Waskiewicz. Virtualization: Writing (and testing) device drivers without hardware. www.linuxplumbersconf.org/2011/ocw/sessions/243. In Linux Plumbers Conference, 2011.

[34] N. Nethercode and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[35] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, 2008.

[36] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI 2011*, 2011.

[37] C. S. Păsăreanu et al. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[38] L. Ryzhyk, I. Kuz, and G. Heiser. Formalising device driver interfaces. In *Workshop on Programming Languages and Systems*, Oct. 2007.

[39] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, 2005.

[40] D. Song et al. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.

[41] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *DSN*, 2006.

[42] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.

[43] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, 2005.

[44] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symp. on Security and Privacy*, 2006. IEEE Computer Society.

[45] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.

# Be Conservative: Enhancing Failure Diagnosis with Proactive Logging

Ding Yuan[†*], Soyeon Park[*], Peng Huang[*], Yang Liu[*], Michael M. Lee[*], Xiaoming Tang[*], Yuanyuan Zhou[*], Stefan Savage[*]

[*]*University of California, San Diego,* [†]*University of Illinois at Urbana-Champaign*

{diyuan,soyeon,ryanhuang,yal036,mmlee,x2tang,yyzhou,savage}@cs.ucsd.edu

## Abstract

When systems fail in the field, logged error or warning messages are frequently the only evidence available for assessing and diagnosing the underlying cause. Consequently, the efficacy of such logging—how often and how well error causes can be determined via postmortem log messages—is a matter of significant practical importance. However, there is little empirical data about how well existing logging practices work and how they can yet be improved. We describe a comprehensive study characterizing the efficacy of logging practices across five large and widely used software systems. Across 250 randomly sampled reported failures, we first identify that more than half of the failures could not be diagnosed well using existing log data. Surprisingly, we find that majority of these unreported failures are manifested via a common set of generic error patterns (e.g., system call return errors) that, if logged, can significantly ease the diagnosis of these unreported failure cases. We further mechanize this knowledge in a tool called *Errlog*, that proactively adds appropriate logging statements into source code while adding only 1.4% performance overhead. A controlled user study suggests that *Errlog* can reduce diagnosis time by 60.7%.

## 1 Introduction

Real systems inevitably experience failure—whether due to hardware faults, misconfigurations or software bugs. However, resolving *why* such a failure has occurred can be extremely time-consuming, a problem that is further exacerbated for failures in the field. Indeed, failures in production systems are the *bête noire* of debugging; they simultaneously require immediate resolution and yet provide the least instrumented and most complex operational environment for doing so. Even worse, when a system fails at a *customer site*, product support engineers may not be given access to the failed system or its data—a situation referred to colloquially as "debugging in the dark".

This paper addresses a simple, yet critical, question: why is it so difficult to debug production software systems? We examine 250 randomly sampled user-reported failures from five software systems (`Apache`, `squid`, `PostgreSQL`, `SVN`, and `Coreutils`) [1] and identify both the source of the failure and the particular information that would have been critical for its diagnosis. Surprisingly, we show that the majority (77%) of these failures manifest through a small number of concrete error patterns (e.g., error return codes, switch statement "fall-throughs", etc.). Unfortunately, more than half (57%) of the 250 examined failures did not log these detectable errors, and their empirical "time to debug" suffers dramatically as a result (taking 2.2X longer to resolve on average in our study).

Driven by this result, we further show that it is possible to fully automate the insertion of such proactive logging statements parsimoniously, yet capturing the key information needed for postmortem debugging. We describe the design and implementation of our tool, *Errlog*, and show that it automatically inserts messages that cover 84% of the error cases manually logged by programmers across 10 diverse software projects. Further, the error conditions automatically logged by *Errlog* capture 79% of failure conditions in the 250 real-world failures we studied. Finally, using a controlled *user study* with 20 programmers, we demonstrate that the error messages inserted by *Errlog* can cut failure diagnosis time by 60.7%.

## 2 Background

While there have been significant advances in postmortem debugging technology, the production environment imposes requirements—low overhead and privacy sensitivity—that are challenging to overcome in commercial settings.

For example, while in principal, deterministic replay—widely explored by the research community [3, 11, 29, 31]—allows a precise postmortem reproduction of the execution leading to a failure, in practice it faces a range of deployment hurdles including high overhead (such systems must log most non-deterministic events), privacy concerns (by definition, the replay trace should contain all input) and integration complexity (particularly in distributed environments with a range of vendors).

By contrast, the other major postmortem debugging advance, cooperative debugging, has broader commercial deployment, but is less useful for debugging individual failures. In this approach, exemplified by systems such as Windows Error Reporting [15] and the Mozilla Quality Feedback Agent [23], failure reports are collected (typically in the form of limited memory dumps due to privacy concerns) and statistically aggregated across large numbers of system installations, providing great utility in triag-

[1] The data we used can be found at: http://opera.ucsd.edu/errlog.htm

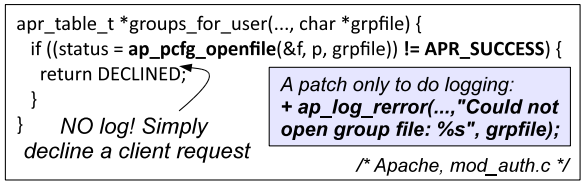



Figure 1: A real world example from `Apache` on the absence of error log message. After diagnosing this failure, the developer released a patch that only adds an error-logging statement.

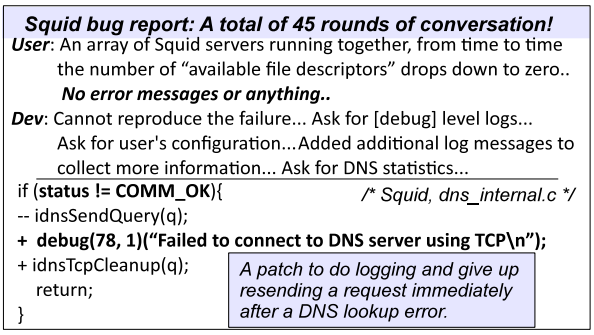



Figure 2: A real world example from `squid` to demonstrate the challenge of failure diagnosis in the absence of error messages, one that resulted in a long series of exchanges (45 rounds) between the user and developers.

ing which failures are most widely experienced (and thus should be more carefully debugged by the vendor). Unfortunately, since memory dumps do not capture dynamic execution state, they offer limited fidelity for exploring the root cause of any individual failure. Finally, sites with sensitive customer information can be reticent to share arbitrary memory contents with a vendor.

### The key role of logging

Consequently, software engineers continue to rely on traditional system logs (e.g., syslog) as a principal tool for troubleshooting failures in the field. What makes these logs so valuable is their ubiquity and commercial acceptance. It is an industry-standard practice to request logs when a customer reports a failure and, since their data typically focuses narrowly on issues of system health, logs are generally considered far less sensitive than other data sources. Moreover, since system logs are typically human-readable, they can be inspected by a customer to establish their acceptability. Indeed, large-scale system vendors such as Network Appliance, EMC, Cisco and Dell report that such logs are available from the majority of their customers and many even allow logs to be transmitted automatically and without review [10].

Even though log messages may not directly pinpoint the root cause (e.g. hardware errors, misconfigurations, software bugs) of a failure, they provide useful clues to narrow down the diagnosis search space. As this paper will show later, failures in the field *with* error messages have much shorter diagnosis time than those without.

### Remembering to log

However, the utility of logging is ultimately predicated on what gets logged; how well have developers anticipated the failure modes that occur in practice? As we will show in this paper, there is significant room for improvement.

Figure 1 shows one real world failure from the `Apache` web server. The root cause was a user's misconfiguration causing `Apache` to access an invalid file. While the error (a failed `open` in `ap_pcfg_openfile`) was explicitly checked by developers themselves, they neglected to log the event and thus there was no easy way to discern the cause postmortem. After many exchanges with the user, the developer added a new error message to record the error, finally allowing the problem to be quickly diagnosed.

Figure 2 shows another real world failure example from the `squid` web proxy. A user reported that the server randomly exhausted the set of available file descriptors without any error message. In order to discern the root cause, `squid` developers worked hard to gather diagnostic information (including 45 rounds of back-and-forth discussion with the user), but the information (e.g., debug messages, configuration setting, etc.) was not sufficient to resolve the issue. Finally, after adding a statement to log the checked error case in which `squid` was unable to connect to a DNS server (i.e., `status != COMM_OK`), they were able to quickly pinpoint the right root cause—the original code did not correctly cleanup state after such an error.

In both cases, the programs themselves already explicitly checked the error cases, but the programmer neglected to include a statement to log the error event, resulting in a long and painful diagnosis.

One of the main objectives of this paper is to provide empirical evidence concerning the value of error logging. However, while we hope our results will indeed motivate developers to improve this aspect of their coding, we also recognize that automated tools can play an important role in reducing this burden.

### Log automation vs log enhancement

Recently, Yuan et. al [37, 36] have studied how developers modify logging statements over time and proposed methods and tools to improve the quality of *existing* log messages by automatically collecting additional diagnostic information in each message. Unfortunately, while such approaches provide clear enhancements to the fidelity provided by a given log message, they cannot help with the all too common cases (such as seen above) when there are *no* log messages at all.

However, the problem of inserting entirely new log messages is significantly more challenging than mere log enhancement. In particular, there are two new challenges posed by this problem:

- *Shooting blind* : Prior to a software release, it is hard to predict what failures will occur in the field, mak-

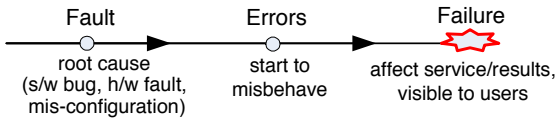



Figure 3: Classic Fault-Error-Failure model.

ing it difficult to know in advance where to insert log messages to best diagnose future failures.

- *Overhead concerns*: Blindly adding *new* log messages can add significant, unacceptable performance overhead to software's normal execution.

Fundamentally, any attempt to add new log messages needs to balance utility and overhead. To reach this goal, our work is heavily informed by practical experience. Just as system builders routinely design around the constraints of technology and cost, so too must they consider the role of cultural acceptance when engineering a given solution. Thus, rather than trying to create an entirely new logging technique that must then vie for industry acceptance, we focus instead on how to improve the quality and utility of the system logs that are already being used in practice. For similar reasons, we also choose to work "bottom-up"—trying to understand, and then improve, how existing logging practice interacts with found failures—rather than attempting to impose a "top-down" coding practice on software developers.

## 3 Where to Log?

Before we decide where to add log points, it is useful to understand how a failure happens. In his seminal work two decades ago, J.C. Laprie decomposed the structural elements of system failures—fault, error and failure—into a model that is widely used today [20]. As shown in Figure 3, a *fault* is a root cause, which can be a software bug, a hardware malfunction, or a misconfiguration. A fault can produce abnormal behaviors referred to as *errors*. However, some of these errors will have no user-perceivable side-effects or may be transparently handled by the system. It is only the subset of remaining errors which further propagate and become visible to users that are referred to as *failures*, such as crash, hang, incorrect result, incomplete functionality, etc.

To further inform our choice of where to place log statements, we divide errors into two categories:

*(i) Detected errors (i.e., exceptions)*: Some errors are checked and caught by a program itself. For example, it is a commonly accepted best practice to check library or system call return values for possible errors.

*(ii) Undetected errors* : Many errors, such as incorrect variable values, may be more challenging to detect mechanistically. Developers may not know in advance what should be a normal value for a variable. Therefore, some errors will always remain latent and undetected until they eventually produce a failure.

| Appl. | LOC | #Default log points* | |
|---|---|---|---|
| | | Total | Err+Warn |
| Apache | 249K | 1160 | 1102 (95%) |
| Squid | 121K | 1132 | 1052 (92%) |
| Postgres | 825K | 6234 | 6179 (99%) |
| SVN | 288K | 1836 | 1806 (98%) |
| Coreutils | 69K | 1086 | 1080 (99%) |

Table 1: Applications used in our study and the number of log points (i.e. logging statements). *: the number of log points under the default verbosity mode. "Err+Warn": number of log points with warning, error, or fatal verbosities.

| Appl. | #Failures | | |
|---|---|---|---|
| | population* | sampled | **with logs** |
| Apache | 838 | 65 | 24 (37%) |
| Squid | 680 | 50 | 20 (40%) |
| Postgres | 195 | 45 | 24 (53%) |
| SVN | 321 | 45 | 25 (56%) |
| Coreutils | 212 | 45 | 15 (33%) |
| **Total** | **2246** | **250** | **108 (43%)** |

Table 2: The number of sampled failures and the subset with failure-related log messages. A failure is classified as "with logs" if any log point exists on the execution path between the fault to the symptom. *: the total number of valid failures that have been fixed in the recent five years in the Bugzilla.

To dive in one step further, detected errors can be handled in three different ways: *(i) Early termination*: a program can simply exit when encountering an error. *(ii) Correct error handling*: a program recovers from an error appropriately, and continues execution. *(iii) Incorrect error handling*: a program does not handle the error correctly and results in an unexpected failure.

These distinctions provide a framework for considering the best program points for logging. In particular, detected errors are naturally "log-worthy" points. Obviously, if a program is about to terminate then there is a clear causal relation between the error and the eventual failure. Moreover, even when a program attempts to handle an error, its exception handlers are frequently buggy themselves since they are rarely well tested [30, 17, 16]. Consequently, logging is appropriate in most cases where a program detects an error explicitly—as long as such logging does not introduce undue overhead. Moreover, logging such errors has no runtime overhead in the common (no error) case.

## 4 Learning from Real World Failures

This section describes our empirical study of how effective existing logging practices are in diagnosis. To drive our study, we randomly sampled 250 real world failures reported in five popular systems, including four servers (`Apache httpd`, `squid`, `PostgreSQL`, and `SVN`) and a utility toolset (`GNU Coreutils`), as shown in Table 1.

The failure sample sets for each system are shown in Table 2. These samples were from the corresponding Bugzilla databases (or mailing lists if Bugzilla was not

available). The reporting of a distinct failure and its follow-up discussions between the users and developers are documented under the same ticket. If a failure is a duplicate of another, developers will close the ticket by marking it as a "duplicate". Once a failure got fixed, developers will often close the ticket as "fixed" and post the patch of the fix. We randomly sampled those non-duplicate, fixed failures that were reported within the recent five years. We carefully studied the reports, discussions, related source code and patches to understand the root cause and its propagation leading to each failure.

In our study, we focus primarily on the *presence* of a failure-related log message, and do not look more deeply into the content of the messages themselves. Indeed, the log message first needs to be present before we consider the quality of its content, and it is also not easy to objectively measure the usefulness of log content. Moreover, Yuan et. al.'s recent LogEnhancer work shows promise in automatically enhancing each existing log message by recording the values of causally-related variables [37].

**Threats to Validity:** As with all characterization studies, there is an inherent risk that our findings may be specific to the programs studied and may not apply to other software. While we cannot establish representativeness categorically, we took care to select diverse programs—written for both server and client environments, in both concurrent and sequential styles. At the very least these software are widely used; each ranks first or second in market share for its product's category. However, there are some commonalities to our programs as all are written in C/C++ and all are open source software. Should logging practice be significantly different in "closed source" development environments or in software written in other languages then our results may not apply.

Another potential source of bias is in the selection of failures. Quantity-wise we are on a firmer ground, as under standard assumptions, the Central Limit Theorem predicts a 6% margin of error at the 95% confidence level for our 250 random samples [28]. However, certain failures might not be reported to Bugzilla. Both `Apache` and `Postgres` have separate mailing lists for security issues; Configuration errors (including performance tunings) are usually reported to the user-discussion forums. Therefore our study might be biased towards software bugs. However, before a failure is resolved, it can be hard for users to determine the nature of the cause, therefore our study still cover many configuration errors and security bugs.

Another concern is that we might miss those very hard failures that never got fixed. However, as the studied applications are well maintained, *severity* is the determining factor of the likelyhood for a failure to be fixed. High severity failures, regardless of its diagnosis difficulty, are likely to be diagnosed and fixed. Therefore the failures that we miss are likely those not-so-severe ones.

Finally, there is the possibility of observer error in the qualitative aspects of our study. To minimize such effects, two inspectors separately investigated every failure and compared their understandings with each other. Our failure study took 4 inspectors 4 months of time.

### 4.1 Failure Characterization

Across each program we extract its embedded log messages and then analyze how these messages relate to the failures we identified manually. We decompose these results through a series of findings for particular aspects of logging behavior.

• **Finding 1:** *Under the default verbosity mode[2], almost all (97%) logging statements in our examined software are error and warning messages (including fatal ones)*. This result is shown in Table 1. Verbose or bookkeeping messages are usually not enabled under the default verbosity mode due to overhead concerns. This supports our expectation that error/warning messages are frequently the only evidence for diagnosing a system failure in the field.

• **Finding 2:** *Log messages produce a substantial benefit, reducing median diagnosis time between 1.4 and 3 times (on average 2.2X faster)*, as shown in Figure 4, supporting our motivating hypothesis about the importance of appropriate logging. This result is computed by measuring each failure's "duration" (i.e., the duration from the time the failure is reported to the time a correct patch is provided). We then divide the failure set into two groups: (1) those with failure-related log messages reported and (2) those without, and compare the median diagnosis time between the two groups. Obviously, some failures might be easier to diagnose than the others, but since our sample set is relatively large we believe our results will reflect any gross qualitative patterns (note, our results may be biased if the difficulty of logging is strongly correlated with the future difficulty of diagnosis, although we are unaware of any data or anecdotes supporting this hypothesis).

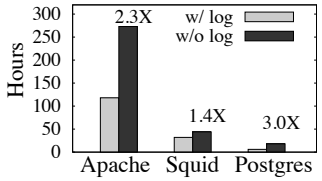


Figure 4: Benefit of logging on diagnosis time (median).

• **Finding 3:** *the majority (57%) of failures do not have failure-related log messages*, leaving support engineers and developers to search for root causes "in the dark". This result is shown in Table 2. Next, we further zoom in to understand why those cases did not have log messages and whether it is hard to log them in advance.

• **Finding 4:** *Surprisingly, the programs themselves have caught early error-manifestations in the majority (61%) of the cases*. The remaining 39% are undetected until the final failure point. This is documented in Figure 5, which

[2]Throughout the entire paper, we assume the default verbosity mode (i.e., no verbosity), which is the typical setting for production runs.

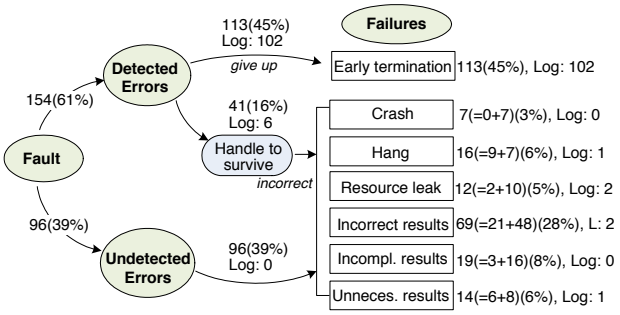



Figure 5: Fault manifestation for our sampled failures. (=x+y): x failures from detected errors and y failures from undetected errors. "Log: N": N cases have failure-related log messages.

| Appl. | Detected Error | | Undetected Error | |
|---|---|---|---|---|
| | Early terminat. | Handle incorrect. | Generic except. | Semantic except. |
| Apache | 23 | 18 | 9 | 15 |
| Squid | 23 | 9 | 10 | 8 |
| Postgres | 24 | 4 | 5 | 12 |
| SVN | 26 | 0 | 7 | 12 |
| Coreutils | 17 | 10 | 8 | 10 |
| **Total** | **113(73%)** | **41(27%)** | **39(41%)** | **57(59%)** |
| | **154** | | **96** | |

Table 3: Error manifestation characteristics of examined software. All detected errors were caught by generic exception checks such as those in Table 5. Some undetected errors could have been detected in the same way.

| Appl. | Early Termination | | Handle Incorrectly | |
|---|---|---|---|---|
| | no log | w/ log | no log | w/ log |
| Apache | 3 | 20 | 14 | 4 |
| Squid | 4 | 19 | 8 | 1 |
| Postgres | 0 | 24 | 4 | 0 |
| SVN | 1 | 25 | 0 | 0 |
| Coreutils | 3 | 14 | 9 | 1 |
| **Total Detected** | **11(10%)** | **102(90%)** | **35(85%)** | **6(15%)** |
| | **113** | | **41** | |

Table 4: Logging practices when general errors are detected.

shows how our sampled failures map to the error manifestation model presented in Section 3. Table 3 breaks them down by application, where the behavior is generally consistent. This indicates that programmers did reasonably well in anticipating many possible errors in advance.

However, as shown in Figure 5 programmers do not comprehensively *log* these detected errors. Fortunately, the result also indicates that log automation can be a rescue—at least 61% of failures manifest themselves through explicitly detected exceptions, which provide natural places to log the errors for postmortem diagnosis.

Further drilling down, we consider two categories of failures for which programmers themselves detected errors along the fault propagation path: early termination and incorrect handling. As shown in Table 4, the vast majority (90%) of the first category log the errors appropriately (10% miss this easy opportunity and impose unnecessary obstacles to debugging; Figure 1 documents one such omission in `Apache`). Logging overhead is not a big concern since the programs subsequently terminate.

| Generic Exception Conditions | Detected Errors | |
|---|---|---|
| | total | w/ logs |
| Function return errors | 69 (45%) | 50 (72%) |
| Exception signals(e.g., SIGSEGV) | 22 (14%) | 22 (100%) |
| Unexpected cases falling into default | 27 (18%) | 12 (44%) |
| Resource leak | 1 (1%) | 1 (100%) |
| Failed input validity check | 17 (11%) | 8 (47%) |
| Failed memory safety check | 7 (4%) | 7 (100%) |
| Abnormal exit/abort from execution | 11 (7%) | 8 (73%) |
| **Total** | **154** | **108 (70%)** |

Table 5: Logging practices for common exceptions.

For the second category (i.e., those failure cases where programs decided to tolerate the errors but unfortunately did so incorrectly), the majority of the cases did not log the detected errors.

Table 4 also shows that `Postgres` and `SVN` are much more conservative in surviving detected errors. Among their 54 detected errors, developers chose early termination in 93% (50/54) of the detected errors. In comparison, for the other three applications, only 63% of the detected errors terminate the executions. We surmise this is because data integrity is the first class requirement for `Postgres` and `SVN`—when errors occur, they seldom allow executions to continue at the risk of data damaging.

• **Finding 5:** *41 of the 250 randomly sampled failures are caused by incorrect or incomplete error handling. Unfortunately, most (85%) of them do not have logs.* This indicates that developers should be conservative in error handling code: at least log the detected errors since error handling code is often buggy. The `squid` example shown in Figure 2 documents such an example.

Adding together the two categories, there are a total of 46 cases that did not log detected errors. In addition, there are also 39 failures shown in Table 3 in which the programs could have detected the error via generic checks (e.g., system call error returns). Therefore we have:

• **Finding 6:** *Among the 142 failures without log messages, there were obvious logging opportunities for 60% (85) of them. In particular, 54% (46) of them already did such checks, but did not log the detected errors.*

**Logging Practice Recommendation:** Overall, these findings suggest that it is worthwhile to conservatively log detected errors, regardless of whether there is error-handling code to survive or tolerate the errors.

### 4.2 Logging Generic Exceptions

Table 5 documents these generic exception patterns, many of which are checked by the studied programs but are not logged. We explain some of them and highlight good practices that we encountered.

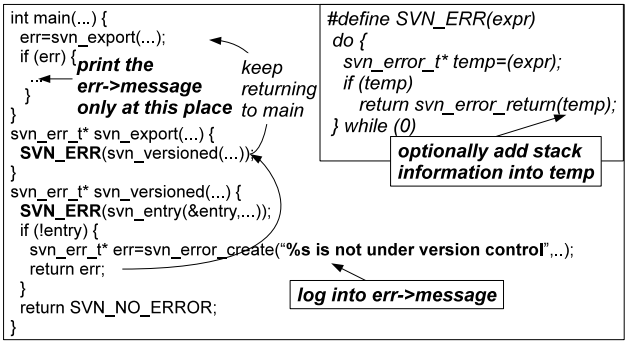



Figure 6: SVN's good logging practices for checking and logging function return errors.

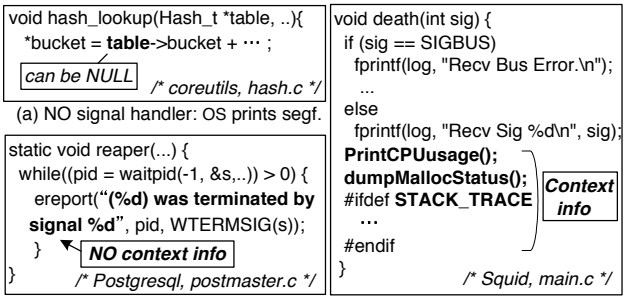



Figure 7: Logging practices for exception signals.

*(1) Function return errors:* It is a common practice to check for function (e.g., system call) return errors. In our study, 45% of detected errors were caught via function return values as shown on Table 5. However, a significant percentage (28%) of them did not log such errors.

**Good practice:** SVN uniformly logs function return errors. First, as shown in Figure 6, almost all SVN function calls are made through a special macro SVN_ERR, which checks for error return. Second, if a function returns an error to its caller, it prepares an error message in a buffer, err->message. Every error is eventually returned back to main through the call path via SVN_ERR and then main prints out the error message. Consequently, as shown in Table 4, almost all exceptions detected by SVN are logged before early termination.

*(2) Exception signals:* In general, many server programs register their own signal handlers to catch fatal signals (e.g., SIGSEGV, SIGTERM). In our study, about 14% of detected errors were caught by the programs' own signal handlers, and fortunately all were logged.

However, all examined software (except for squid) only logs signal names. Figure 7 compares the logging practices in three of them: (a) Coreutils does not have a signal handler. OS prints a generic "segmentation fault" message. (b) Postgres's log does not provide much better information than the default OS's signal handler. (c) **Good practice:** squid logs system status and context information such as CPU and memory usage, as well as the stack frames, when catching exception signals.

| | | | |
|---|---|---|---|
| Statement cov.* | 10 (18%) | Decision cov. | 12 (21%) |
| Condition cov. | 2 (4%) | Weak mutation | 4 (7%) |
| Mult. cond. cov. | 2 (4%) | Loop cov. | 1 (2%) |
| Concurr. cov. | 1 (2%) | Perf. profiling | 1 (2%) |
| Functional cov. | 34 (60%) | **Total failures** | 57 |

Table 6: The number of hard-to-check failures that could have been caught during testing, assuming 100% test coverage with each criteria. *: can also be detected by decision coverage test.

*(3) Unexpected cases falling through into default:* Sometimes when programs fail to enumerate all possible cases in a switch statement, the execution may unexpectedly fall through into the base "default" case, and lead to a failure. In our study, 18% of detected errors belong to this category, but only 44% of them are logged.

*(4) Other exceptions*: Programs also perform other types of generic exception checks such as bound-checks, input vadility checks, resource leak checks, etc., (Table 5) but they often forget to log detected errors, losing opportunities to gather evidences for postmortem diagnosis.

### 4.3 Logging for Hard-to-check Failures

As shown earlier in Table 3, 57 failures are hard to detect via generic exception checks. We refer them as *hard-to-check errors*. When a production failure occurs, it is usually due to an unusual input or environment triggering some code paths that are not covered during in-house testing. Table 6 shows that 21% of the 57 hard-to-check failure cases execute some branch edges that we surmise have never been executed during testing (otherwise, the bugs on those edges would definitely have been exposed)[3]. Therefore, if we log on those branch decisions that have not been covered during testing, i.e., cold paths, it would be useful for diagnosis. Of course, special care needs to be taken if some cold paths show up too frequently during runtime.

• **Finding 7:** *Logging for untested code paths would collect diagnostic information for some of them.*

## 5 *Errlog*: A Practical Logging Tool

Driven by the findings in our study, we further build an automatic logging tool called *Errlog*, which analyzes the source code to identify potential unlogged exceptions (abnormal or unusual conditions), and then inserts log statements. Therefore, *Errlog* can automatically enforce good logging practices. We implement our source code analysis algorithms using the Saturn [2] static analysis framework.

*Errlog* faces three major challenges: (1) Where are such potential exceptions? (2) Has the program itself checked for the exception? If so, has the program logged it after checking it? (3) Since not every potential exception may be terminal (either because the program has mechanisms to survive it or it is not a true exception at all), how do we

[3]Due to software's complexity, cost of testing, and time-to-market pressure, complex systems can rarely achieve 100% test coverage.

| | Exception Pattern | How to identify in source code |
|---|---|---|
| DE | Function return error | Mechanically search for libc/system calls. If a libc/system call's error return value is not checked by the program, *Errlog* injects new error checking code. Such a check won't incur too much overhead as it is masked by the overhead of a function call. |
| | Failed memory safety check | Search for checks for null pointer dereference and out-of-bound array index. If no such safety check exists, *Errlog* does *NOT* add any check due to false positive concerns. |
| | Abnormal exit/abort | Search for "abort, exit, _exit". The constraint *EC* for this pattern is "true". |
| | Exception signals | Intercept and log abnormal signals. Our logging code uses memory buffer and is re-entrant. |
| LE | Unexpected cases falling into default | Search for the "default" in a switch statement or a switch-like logic, such as `if.. else if.. else...`, where at least the same variable is tested in each `if` condition. |
| | Invalid input check | Search for text inputs, using a simple heuristic to look for string comparisons (e.g., strcmp). The exception is the condition that these functions return "not-matched" status. In our study, 47% of the "invalid input checks" are from these standard string matching functions. |
| AG | Resource leak | *Errlog* monitors resource (memory and file descriptor) usage and logs them with context information. *Errlog* uses exponential-based sampling to reduce the overhead (Section 5.3). |

Table 7: Generic exception patterns searched by *Errlog*. These patterns are directly from our findings in Table 5 in Section 4.

avoid significant performance overhead without missing important diagnostic information?

To address the first challenge, *Errlog* follows the observations from our characterization study. It identifies potential exceptions by mechanically searching in the source code for the seven generic exception patterns in Table 5. In addition, since many other exception conditions are program specific, *Errlog* further "learns" these exceptions by identifying the frequently logged conditions in the target program. Moreover, it also optionally identifies untested code area after in-house testing.

For the second challenge, *Errlog* checks if the exception check already exists, and if so, whether a log statement also exists. Based on the results, *Errlog* decides whether to insert appropriate code to log the exception.

To address the third challenge, *Errlog* provides three logging modes for developers to choose from, based on their preferences for balancing the amount of log messages versus performance overhead: *Errlog*-DE for logging definite exceptions, *Errlog*-LE for logging definite and likely exceptions, and *Errlog*-AG for aggressive logging. Moreover, *Errlog*'s runtime logging library uses dynamic sampling to further reduce the overhead of logging without losing too much logging information.

**Usage** Users of *Errlog* only need to provide the name of the default logging functions used in each software. For example, the following command is to use *Errlog* on the `CVS` version control system:

```
Errlog --logfunc="error" path-to-CVS-src
```

where `error` is the logging library used by `CVS`. *Errlog* then automatically analyzes the code and modifies it to insert new log statements. *Errlog* can also be used as a tool that recommends where to log (e.g., a plug-in to the IDE) to the developers, allowing them to insert logging code to make the message more meaningful.

### 5.1 Exception Identification

In this step, *Errlog* scans the code and generates the following predicate: *exception(program_point P, constraint EC)*, where *P* is the program location of an exception check, and *EC* is the constraint that causes the exception to happen. In the example shown in Figure 2, *P* is the source code location of "`if (status!=COMM_OK)`", and *EC* is `status!=COMM_OK`. *EC* is used later to determine under which condition we should log the exception and whether the developer has already logged the exception.

**Search for generic exceptions** Table 7 shows the generic exception patterns *Errlog* automatically identifies, which are directly from the findings in our characterization study.

#### 5.1.1 Learning Program-Specific Exceptions

*Errlog*-LE further attempts to automatically identify program-specific exceptions *without any program-specific knowledge*. If a certain condition is frequently logged by programmers in multiple code locations, it is likely to be "log-worthy". For example, the condition `status!=COMM_OK` in Figure 2 is a `squid`-specific exception that is frequently followed by an error message. Similar to previous work [12] that statically learns program invariants for bug detection, *Errlog*-LE automatically learns the conditions that programmers log on more than two occasions. To avoid false positives, *Errlog* also checks that the logged occasions outnumber the unlogged ones.

**The need for control and data flow analysis** It is non-trivial to correctly identify log-worthy conditions. For example, the exception condition in Figure 8 is that `pcre_malloc` returns `NULL`, not `tmp==NULL`. *Errlog* first analyzes the control-flow to identify the condition that immediately leads to an error message. It then analyzes the data-flow, in a backward manner, on each variable involved in this condition to identify its source. However, such data-flow analysis cannot be carried arbitrarily deep as doing so will likely miss the actual exception source. For each variable *a*, *Errlog*'s data-

```
tmp=pcre_malloc(...);
if (tmp == NULL)
  goto out_of_memory;
  ... ...
out_of_memory:
  error ("out of memory");
```

Figure 8: Example showing the need of control & data flow analysis.

flow analysis stops when it finds a *live-in* variable as its source, i.e., a function parameter, a global variable, a constant, or a function return value. In Figure 8, *Errlog* first identifies the condition that leads to the error message being `tmp==NULL`. By analyzing the data-flow of `tmp`, *Errlog* further finds its source being the return value of `pcre_malloc`. Finally, it replaces the `tmp` with `pcre_malloc()` and derives the correct error condition, `pcre_malloc()==NULL`. Similarly, the condition `status!=COMM_OK` in Figure 2 is learnt because `status` is a formal parameter of the function.

**Identifying helper logging functions** *Errlog* only requires developers to provide the name of the default logging function. However, in all the large software we studied, there are also many helper logging functions that simply wrap around the default ones. *Errlog* identifies them by recursively analyzing each function in the bottom-up order along the call graph. If a function $F$ prints a log message under the condition `true`, $F$ is added to the set of logging functions.

**Explicitly specified exceptions (optional)** *Errlog* also allows developers to explicitly specify domain-specific exception conditions in the form of code comments right before the exception condition check. Our experiments are conducted without this option.

#### 5.1.2 Identifying Untested Code Area (optional)

*Errlog*-AG further inserts log points for code regions not covered by in-house testing. We use the test coverage tool `GNU gcov` [14] and the branch decision coverage criteria. For each untested branch decision, *Errlog* instruments a log point. For multiple nested branches, *Errlog* only inserts a log point at the top level. This option is not enabled in our experiments unless otherwise specified.

### 5.2 Log Printing Statement Insertion

**Filter the exceptions already logged by a program** This is to avoid redundant logging, which can result in overhead and redundant messages. Determining if an exception $E$ has already been logged by a log point $L$ is challenging. First, $L$ may not be in the basic block immediately after $E$. For example, in Figure 8, the exception check and its corresponding log point are far apart. Therefore, simply searching for $L$ within the basic block following $E$ is not enough. Second, $E$ might be logged by the caller function via an error return code. Third, even if $L$ is executed when $E$ occurs, it might not indicate that $E$ is logged since $L$ may be printed regardless of whether $E$ occurs or not.

*Errlog* uses precise path sensitive analysis to determine whether an exception has been logged. For each identified $exception(P, EC)$, *Errlog* first checks whether there is a log point $L$ within the same function $F$ that: i) will execute if $EC$ occurs, and ii) there is a path reaching $P$ but not $L$ (which implies that $L$ is not always executed regardless of $EC$). If such an $L$ exists, then $EC$ has already been logged. To check for these two conditions, *Errlog* first captures the path-sensitive conditions to reach $P$ and $L$ as $C_P$ and $C_L$ respectively. It then turns the checking of the above two conditions into a satisfiability problem by checking the following using a SAT solver:

1. $C_P \wedge EC \wedge \neg C_L$ is *not* satisfiable.
2. $C_P \wedge \neg C_L$ is satisfiable.

The first condition is equivalent to i), while the second condition is equivalent to ii).

If no such log point exists, *Errlog* further checks if the exception is propagated to the caller via return code. It checks if there is a return statement associated with $EC$ in a similar way as it checks for a log point. It remembers the return value, and then analyzes the caller function to check if this return value is logged or further propagated. Such analysis is recursively repeated in every function.

**Log placement** If no logging statement is found for an exception $E$ from the analysis above, *Errlog* inserts its own logging library function, "`Elog(logID)`", into the basic block after the exception check. If no such check exists, *Errlog* also adds the check.

Each logging statement records (i) a log ID unique to each log point, (ii) the call stack, (iii) casually-related variable values identified using LogEnhancer [37] [4], (iv) a global counter that is incremented with each occurrence of any log point, to help postmortem reconstruction of the message order. For each system-call return error, the `errno` is also recorded. No static text string is printed at runtime. *Errlog* will compose a postmortem text message by mapping the log ID and `errno` to a text string describing the exception. For example, *Errlog* would print the following message for an `open` system-call error: *"open system call error: No such file or directory: ./filepath ..."*.

### 5.3 Run-time Logging Library

Due to the lack of run-time information and domain knowledge during our static analysis, *Errlog* may also log non-exception cases, especially with *Errlog*-LE and *Errlog*-AG. If these cases occur frequently at run time, the time/space overhead becomes a concern.

To address this issue, *Errlog*'s run-time logging library borrows the idea of adaptive sampling [19]. It exponentially decreases the logging rate when a log point $L$ is reached from the same calling context many times. The rationale is that frequently occurred conditions are less likely to be important exceptions; and even if they are, it is probably useful enough to only record its $2^n$th dynamic occurrences. To reduce the possibility of missing true exceptions, we also consider the whole context (i.e., the call stack) instead of just each individual log point. For each calling context reaching each $L$ we log its $2^n$th dynamic occurrences. We further differentiate system call return errors by the value of `errno`. For efficiency, *Errlog* logs into

[4] LogEnhancer [37] is a static analysis tool to identify useful variable values that should be logged with each *existing* log message.

| App. | *Errlog*-DE | | | | | *Errlog*-LE | | | | *Errlog*-AG | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | func. ret. | mem. safe. | abno. exit | sig-nals | **Total** | switch-default | input check | learned errors | **Total** | res. leak | **Total** |
| Apache | 30 | 41 | 9 | 22 | 102 (0.09X) | 117 | 389 | 360 | 968 (0.83X) | 24 | 992 (0.86X) |
| Squid | 393 | 112 | 29 | 3 | 537 (0.47X) | 116 | 147 | 17 | 817 (0.72X) | 26 | 843 (0.74X) |
| Postgres | 619 | 166 | 28 | 9 | 822 (0.13X) | 432 | 7 | 1442 | 2703 (0.43X) | 65 | 2768 (0.44X) |
| SVN | 33 | 6 | 1 | 3 | 43 (0.02X) | 53 | 1 | 8 | 105 (0.06X) | 31 | 136 (0.07X) |
| Coreutil cp | 34 | 4 | 9 | 2 | 49 (0.73X) | 13 | 5 | 0 | 67 (1.00X) | 4 | 71 (1.06X) |
| CVS | 1109 | 360 | 23 | 3 | 1495 (1.30X) | 52 | 49 | 645 | 2241 (1.95X) | 32 | 2273 (1.97X) |
| OpenSSH | 714 | 31 | 26 | 3 | 774 (0.32X) | 112 | 31 | 63 | 980 (0.40X) | 23 | 1003 (0.41X) |
| lighttpd | 171 | 16 | 30 | 3 | 220 (0.27X) | 67 | 27 | 6 | 320 (0.39X) | 37 | 357 (0.44X) |
| gzip | 45 | 3 | 32 | 3 | 83 (0.85X) | 40 | 3 | 16 | 142 (1.45X) | 14 | 156 (1.59X) |
| make | 339 | 6 | 16 | 3 | 364 (2.72X) | 29 | 12 | 10 | 415 (3.10X) | 6 | 421 (3.14X) |
| Total | 3487 | 745 | 203 | 54 | 4489 (0.30X) | 1031 | 671 | 2567 | 8758 (0.58X) | 262 | 9020 (0.60X) |

Table 8: Additional log points added by *Errlog*. The "total" of LE and AG include DE and DE+LE, respectively, and are compared to the number of existing log points (Table 1 and 9). Note that most of these log points are *not* executed during normal execution.

in-memory buffers and flushes them to disk when they become full, execution terminates, and when receiving user defined signals.

Note that comparing with other buffering mechanisms such as "log only the first/last N occurrences", adaptive sampling offers a unique advantage: the printed log points can be postmortem ranked in the reverse order of their occurrence frequencies, with the intuition that frequently logged ones are less likely true errors.

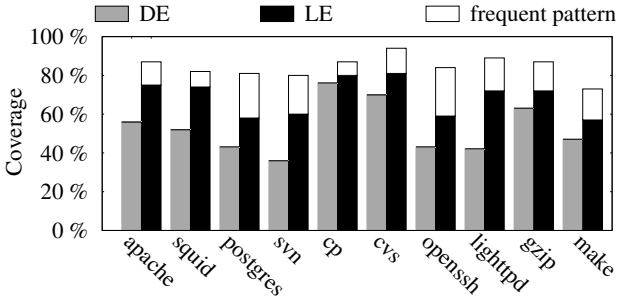



Figure 9: Coverage of existing log points by *Errlog*. For *Errlog*-LE, we break down the coverages into log points identified by generic exceptions and those learned by frequent logging patterns. AG has similar coverages as LE.

## 6 In-lab Experiment

We evaluate *Errlog* using both in-lab experiments and a controlled user study. This section presents the in-lab experiments. In addition to the applications we used in our characterization study, we also evaluate *Errlog* with 5 more applications as shown in Table 9.

### 6.1 Coverage of Existing Log Points

It is hard to objectively evaluate the usefulness of log messages added by *Errlog* without domain knowledge. However, one objective evaluation is to measure how many of the existing log points, added manually by developers, can be added by *Errlog* automatically. Such measurement could evaluate how much *Errlog* matches domain experts' logging practice.

Note that while our Section 4 suggests that the current logging practices miss many logging opportunities, we do not imply that existing log points are unnecessary. On the contrary, existing error messages are often quite helpful in failure diagnosis as they were added by domain experts, and many of them were added in the form of afterthoughts. This is confirmed by our Finding 2: existing log messages would reduce the diagnosis time by 2.2X. Therefore, comparing with existing log points provides an objective measurement on the effectiveness of *Errlog*.

Figure 9 shows that *Errlog*, especially with *Errlog*-LE, can automatically cover an average of 84% of existing log points across all evaluated software. In comparison, *Errlog*-DE logs only definite errors and achieves an average of 52% coverage, still quite reasonable since on average it adds less than 1% overhead.

| App. | description | LOC | #Default Log Points | |
|---|---|---|---|---|
| | | | Total | Err+Warn |
| CVS | version cont. sys. | 111K | 1151 | 1139 (99%) |
| OpenSSH | secure connection | 81K | 2421 | 2384 (98%) |
| lighttpd | web server | 54K | 813 | 792 (97%) |
| gzip | comp/decom. files | 22K | 98 | 95 (97%) |
| make | builds programs | 29K | 134 | 129 (96%) |

Table 9: The *new* software projects used to evaluate *Errlog*, in addition to the five examined in our characterization study.

### 6.2 Additional Log Points

In addition to the existing log points, *Errlog* also adds new log points, shown in Table 8. Even though *Errlog*-LE adds 0.06X–3.10X additional log points, they only cause an average of 1.4% overhead (Section 6.3) because most of them are not triggered when the execution is normal.

**Logging for untested branch decision** Table 10 shows *Errlog*-AG's optional logging for untested branch decisions, which is not included in the results above. For `Apache`, `Postgres`, `SVN` and `Coreutils`, we used the test cases released together with the software.

| App. | Uncovered decisions | # log points |
|---|---|---|
| Apache | 57.0% (2915) | 655 |
| Postgres | 51.7% (51396) | 11810 |
| SVN | 53.7% (14858) | 4051 |
| Coreutils | 62.3% (9238) | 2293 |

Table 10: Optional logging for untested branch decisions.

| Software | Adaptive sampling* | | | No sampling | |
|---|---|---|---|---|---|
| | DE | LE | AG | DE | LE |
| Apache | <1% | <1% | 2.7% | <1% | <1% |
| Squid | <1% | 1.8% | 2.1% | 4.3% | 9.6% |
| Postgres | 1.5% | 1.9% | 2.0% | 12.6% | 40.1% |
| SVN | <1% | <1% | <1% | <1% | <1% |
| cp | <1% | <1% | <1% | 6.3% | 6.3% |
| CVS | <1% | <1% | <1% | <1% | 2.3% |
| Openssh scp | 2.0% | 4.6% | 4.8% | 5.2% | 27.1% |
| lighttpd | <1% | <1% | 2.2% | <1% | <1% |
| gzip | <1% | <1% | <1% | <1% | <1% |
| make | 3.9% | 4.0% | 4.8% | 4.2% | 6.8% |
| Average | 1.1% | 1.4% | 2.1% | 3.5% | 9.4% |

Table 11: The performance overhead added by *Errlog*'s logging. *: By default, *Errlog* uses adaptive sampling. We also show the overhead without using sampling only to demonstrate the effect of adaptive sampling.

| | func. ret. | mem. safe. | switch default | input check | learned errors | Total |
|---|---|---|---|---|---|---|
| Log pts. | 5 | 8 | 5 | 7 | 10 | 35 |

Table 12: Noisy log points exercised during correct executions.

### 6.3 Performance Overhead

We evaluate *Errlog*'s logging overhead during the software's normal execution. Server performance is measured in peak-throughput. Web servers including `Apache httpd`, `squid`, and `lighttpd` are measured with `ab` [4]; `Postgres` is evaluated with `pgbench` [24] using the select-only workload; `SVN` and `CVS` with a combination of check-out, merge, copy, etc.; `OpenSSH`'s `scp` with repeatedly transferring files; `gzip` and `cp` with processing large files; `make` with compiling `PostgreSQL`.

Table 11 shows *Errlog*'s logging overhead during the normal execution. For all evaluated software, the default *Errlog*-LE imposes an average of 1.4% run-time overhead, with a maximum of 4.6% for `scp`. The most aggressive mode, *Errlog*-AG, introduces an average of 2.1% overhead and a maximum of 4.8%. The maximum runtime memory footprint imposed by *Errlog* is less than 1MB.

`scp` and `make` have larger overhead than others in Table 11. It is because `scp` is relatively CPU intensive (lots of encryptions) and also has a short execution time. Compared to I/O intensive workloads, the relative logging overhead added by *Errlog* becomes more significant in CPU intensive workloads. Moreover, short execution time may not allow *Errlog* to adapt the sampling rate effectively. `make` also has relatively short execution time.

**Noisy messages** More log messages are not always better. However, it is hard to evaluate whether each log point captures a true error since doing so requires domain expertise. Therefore we simply treat the log points that are executed during our performance testing as noisy messages as we are not aware of any failures in our performance testing. Among the five applications we used in our failure study, only a total of 35 log points (out of 405 error condition checks) are executed, between 3-12 for each application. Table 12 breaks down these 35 log points by different patterns. Examples of these include using the error return of `stat` system call to verify a file's non-existence in normal executions. Since we use adaptive sampling, the size of run-time log is small (less than 1MB).

| App. | Tot. fails | w/ existing logs | *Errlog*- | | |
|---|---|---|---|---|---|
| | | | DE | LE | AG |
| Apache | 58 | 18 (31%) | 28 (48%) | 43 (74%) | 48 (83%) |
| Squid | 45 | 15 (33%) | 23 (51%) | 37 (82%) | 37 (82%) |
| Postgres | 45 | 24 (53%) | 26 (58%) | 32 (71%) | 34 (76%) |
| SVN | 45 | 25 (56%) | 30 (67%) | 33 (73%) | 33 (73%) |
| Coreutils | 45 | 15 (33%) | 28 (62%) | 34 (76%) | 37 (82%) |
| **Total** | 238* | 97 (41%) | 135 (57%) | 179 (75%) | 189 (79%) |

Table 13: *Errlog*'s effect on the randomly sampled 238 real-world failure cases. *: 12 of our 250 examined failure cases cannot be evaluated since the associated code segments are for different platforms incompatible with our compiler.

**Sampling overhead comparison** We also evaluate the efficiency of adaptive sampling by comparing it with "no sampling" in Table 11. "No sampling" logging records *every* occurrence of executed log points into memory buffer and flushes it to disk when it becomes full. We do not evaluate "no sampling" on *Errlog*-AG as it is more reasonable to use sampling to monitor resource usage.

Adaptive sampling effectively reduces *Errlog*-LE's overhead from no-sampling's 9.4% to 1.4%. The majority of the overhead is caused by a few log points on an execution's critical paths. For example, in `Postgres`, the index-reading function, where a lock is held, contains a log point. By decreasing the logging rate, adaptive sampling successfully reduces no-sampling's 40.1% overhead to 1.9%. In comparison, the effect of sampling is less obvious for `make`, where its short execution time is not sufficient for adaptive sampling to adjust its sampling rate.

**Analysis time** Since *Errlog* is used off-line to add log statements prior to software release, the analysis time is less critical. *Errlog* takes less than 41 minutes to analyze each evaluated software except for `postgres`, which took 3.8 hours to analyze since it has 1 million LOC. Since *Errlog* scans the source code in one-pass, its analysis time roughly scales linearly with the increase of the code size.

### 6.4 Real World Failures

Table 13 shows *Errlog*'s effect to the real-world failures we studied in Section 4. In this experiment we turn on the logging for untested code region in *Errlog*-AG. Originally, 41% of the failures had log messages. With *Errlog*, 75% and 79% of the failures (with *Errlog*-LE and AG, respec-

| Name | Repro | Description |
|---|---|---|
| apache crash | ✓ | A configuration error triggered a NULL pointer dereference. |
| apache no-file | ✓ | The name of the group-file contains a typo in the configuration file. |
| chmod | × | fail silently on dangling symbolic link. |
| cp | ✓ | fail to copy the content of /proc/cpuinfo. |
| squid | × | when using Active Directory as authentication server, incorrectly denies user's authentication due to truncation on security token. |

Table 14: Real-world failures used in our user study.

tively) have failure-related log messages.

**Effectiveness of *Errlog* for Diagnosis** We evaluate the usefulness of the added log messages in diagnosis using SherLog [35], a log-inference engine. Given log messages related to a failure, SherLog reconstructs the execution paths must/may have taken to lead to the failure. Our evaluation shows that 80% of the new messages can help SherLog to successfully infer the root causes.

## 7 User Study

We conduct a controlled user study to measure the effectiveness of *Errlog*. Table 14 shows the five real-world production failures we used. Except for "`apache` crash", the other four failed silently. Failures are selected to cover diverse root causes (bugs and misconfigurations), symptoms, and reproducibilities. We test on 20 programmers (no co-author of this paper is included), who indicated that they have extensive and recent experience in C/C++.

Each participant is asked to fix the 5 failures as best as she/he could. They are provided a controlled Linux workstation and a full suite of debugging tools, including `GDB`. Each failure is given to a randomly chosen 50% of the programmers with *Errlog* inserted logs, and the other 50% without *Errlog* logs. All participants are given the explanation of the symptom, the source tree, and instructions on how to reproduce the three reproducible failures–this is actually biased against *Errlog* since it makes the no-*Errlog* cases easier (it took us hours to understand how to reproduce the two `Apache` failures). The criteria of a successful diagnosis is for the users to *fix* the failure. Further, there is a 40 minutes time limit per failure; failing to fix the failure is recorded as using the full limit. 40 minutes is a best estimation of the maximum time needed.

Note that this is a best-effort user study. The potential biases should be considered when interpreting our results. Below we discuss some of the potential biases and how we addressed them in our user study:

*Bias in case selection:* We did not select some very hard-to-diagnose failures and only chose two unreproducible ones, since diagnosis can easily take hours of time. This bias, however, is likely *against Errlog* since our result shows that *Errlog* is more effective on failures with a larger diagnosis time.

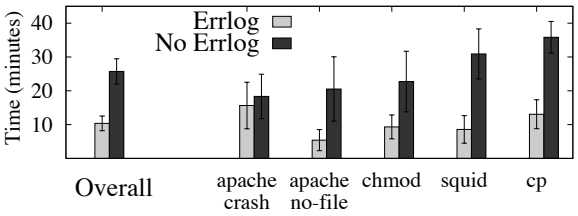



Figure 10: User study result, with error bars showing 95% confidence interval.

*Bias in user selection:* The participants might not represent the real programmers of these software. Only four users indicated familiarities with the code of these software. However, we do provide each user a brief tutorial of the relevant code. Moreover, studies [34] have shown that many programmers fixing real-world production failures are also not familiar with the code to be fixed because many companies rely on sustaining engineers to do the fix. Sustaining engineers are usually not the developers who wrote the code in the first place.

*Bias in methodology:* As our experiment is a single-blind trial (where we, the experimenters, know the ground truth), there is a risk that the subjects are influenced by the interaction. Therefore we give the users written instructions for each failure, with the only difference being the presence/absence of the log message; we also minimize our interactions with the user during the trial.

**Results** Figure 10 shows our study result. On average programmers took 60.7% less time diagnosing these failures when they were provided with the logs added by *Errlog* ($10.37 \pm 2.18$ minutes versus $25.72 \pm 3.75$ minutes, at 95% confidence interval). An unpaired T-test shows that the hypothesis "*Errlog* saves diagnosis time" is true with a probability of 99.9999999% ($p=5.47 \times 10^{-10}$), indicating the data strongly supports this hypothesis.

Overall, since factors such as individuals' capability are amortized among a number of participants, the only constant difference between the two control groups is the existence of the log messages provided by *Errlog*. Therefore we believe the results reflect *Errlog*'s effectiveness.

Less formally, all the participants reported that they found the additional error messages provided by *Errlog* significantly helped them diagnose the failures. In particular, many participants reported that "(*Errlog* added) logs are in particular helpful for debugging more complex systems or unfamiliar code where it required a great deal of time in isolating the buggy code path."

However, for one failure, "`apache` crash", the benefit of *Errlog* is not statistically significant. The crash is caused by a `NULL` pointer dereference. *Errlog*'s log message is printed simply because `SIGSEGV` is received. Since users could reproduce the crash and use `GDB`, they could relatively quickly diagnose it even without the log.

In comparison, *Errlog* achieves maximum diagnosis time reduction in two cases: "`squid`" (by 72.3%) and

"apache no-file" (by 73.7%). The squid bug is a tricky one: due to the complexity in setting up the environment and user privacy concerns, it is not reproducible by the participants. Without logs, most of the control group took time-consuming goose chases through the complicated code. In contrast, the error message from *Errlog*, caused by the abnormal return of snprintf, guided most of the users from the other group to quickly spot the unsafe use of snprintf that truncated a long security token.

In the "apache no-file" case (the one shown in Figure 1), apache cannot open a file due to a typo in the configuration file. Without any error message, some programmers did not even realize this was caused by a misconfiguration and started to debug the code. In contrast, the error message provided by *Errlog* clearly indicates the open system call cannot find the file, allowing most programmers in this group to quickly locate and fix the typo.

## 8 Limitations and Discussions

*(1) What failures cannot benefit from Errlog?* Not all the failures can be successfully diagnosed with *Errlog*. First, *Errlog* fails to insert log messages for 21% of the randomly sampled failures (Table 13). The error conditions of these failures are subtle, domain-specific, or are caused by underlying systems whose errors are not even properly propagated to the upper level applications [26]. *Errlog* could be further used with low-overhead run-time invariants checking [13] to log the violations to the invariants.

Second, while log messages provide clues to narrow down the search, they may not pinpoint the root cause. Section 6.4 shows that for 20% of the failures, the added log messages are not sufficient for the diagnosis. Such examples include (i) concurrency bugs where the thread-interleaving information is required and (ii) failures where key execution states are already lost at the log point. Note that a majority ($> 98\%$) of failures in the real world are caused by semantic bugs, misconfigurations, and hardware errors but not by concurrency bugs [27].

However, this does not mean *Errlog* can only help diagnosing easy failures. Log messages collect more diagnostic information, not to pinpoint the exact root cause. Evidences provided by logs along the fault propagation chain, despite how complicated this chain is, will likely help narrowing down the search space. Therefore even for concurrency bugs, an error message is still likely to be useful to reduce the diagnosis search space.

*(2) What is the trade-off of using adaptive sampling?* Adaptive sampling might limit the usage of log messages. If the program has already exercised a log point, it is possible that this log will not be recorded for a subsequent error. Long running programs such as servers are especially vulnerable to this limitation. To alleviate this limitation, we differentiate messages by runtime execution contexts including stack frames and errno. We can also periodically reset the sampling rate for long running programs.

In addition, adaptive sampling might preclude some useful forms of reasoning for a developer. For instance, the absence of a log message no longer guarantees that the program did not take the path containing the log point (assuming the log message has already appeared once). Moreover, even with the global order of each printed message, it would be harder to postmortem correlate them given the absence of some log occurrences.

To address this limitation, programmers can first use adaptive sampling on every log point during the testing and beta-release runs. Provided with the logs printed during normal executions, they can later switch to non-sampling logging for those not-exercised log points (which more likely capture true errors), while keep using sampling on those exercised ones for overhead concerns.

*(3) Can Errlog completely replace developers in logging?* The semantics of the auto-generated log messages are still not comparable to those written by developers. The message semantic is especially important for support engineers or end users who usually do not have access to the source code. *Errlog* can be integrated into the programming IDE, suggesting logging code as developers program and allowing them to improve inserted log messages and assign proper verbosity levels.

*(4) How about verbose log messages?* This paper only studies log messages under the default verbosity mode, which is the typical production setting due to overhead concerns [36]. Indeed, verbose logs can also help debugging production failures as developers might ask user to reproduce the failure with the verbose logging enabled. However, such repeated failure reproduction itself is undesirable for the users in the first place. How to effectively insert verbose messages remains as our future work.

*(5) What is the impact of the imprecisions of the static analysis?* Such imprecisions, mainly caused by pointer aliasing in C/C++, might result in redundant logging and insufficient logging. However, given that Saturn's intra-procedural analysis precisely tracks pointer aliases [2], such impact is limited only to the inter-procedural analysis (where the error is propagated via return code to the callers to log). In practice, however, we found programmers seldom use aliases on an error return code.

## 9 Related Work

**Log enhancement and analysis** Some recent proposals characterize, improve, and analyze *existing* log messages for failure diagnosis [36, 37, 35, 32]. LogEnhancer [37] adds variables into each existing log message to collect more diagnostic information; Our previous work [36] studied developers' modifications to existing logging code and found that they often cannot get the logging right at the first attempt. SherLog [35] is a postmortem debugger that combines runtime logs and source code to reconstruct

the partial execution path occurred in the failed execution.

However, all of these studies only deal with *existing* log messages, and do not address the challenge of where to add new logs as discussed in Section 2.

The different objective makes our techniques very different from the systems listed above. For example, both SherLog and LogEnhancer start from an existing log message to backtrack the execution paths. In comparison, *Errlog* scans the entire source code to identify different exception patterns. *Errlog* also learns the program-specific errors, identifies the untested code areas, checks whether exceptions are already logged, and logs with adaptive sampling at runtime. All these techniques are unique to *Errlog* for its objective.

**Detecting bugs in exception handling code** Many systems aim to expose bugs in the exception handling code [17, 26, 22, 16, 33], including two [17, 26] that statically detect the unchecked errors in file-system code. *Errlog* is different and complementary to these systems. *Errlog* has a different goal: easing the postmortem failure diagnosis, instead of detecting bugs. Therefore we need to empirically study the weakness in logging practices, and build a tool to automatically add *logging statements*. Only our exception identification part (Section 5.1) shares some similarities with [17, 26]. In addition, some exception patterns such as fall-through in switch statements, signal handling, and domain-specific errors are not checked by prior systems. These additional exceptions detected by *Errlog* might benefit the prior systems for detecting more bugs in the corresponding error handling code.

**Bug-type specific diagnostic information collection** Some studies [5, 19, 6, 38] proposed to collect runtime information for specific types of bugs. For example, DCop [38] collects runtime lock/unlock trace for diagnosing deadlocks. These systems are more powerful but are limited to debugging only specific types of fault, whereas *Errlog* applies to various fault types but may log only the erroneous manifestations (instead of root causes).

**Logging for deterministic replay** Other systems [31, 3, 11, 29] aim at deterministically replaying the failure execution, which generally requires recording all the inputs or impose high run-time logging overhead. Castro et al. propose replacing private information while preserving diagnosis power [8], but they require replaying the execution at the user's site. *Errlog* is complementary and targets the failures where reproduction is difficult due to privacy concerns, unavailability of execution environments, etc.

**Tracing for failure diagnosis** Execution trace monitoring [21, 18, 25, 1] has been used to collect diagnostic information, where both normal and abnormal executions are monitored. *Errlog* logs only exceptions so its runtime overhead is small during normal executions.

**Static analysis** *Errlog* uses the Saturn [2] symbolic execution framework. Similar static analysis [7, 9] was used for other purposes, such as bug detection. *Errlog* uses code analysis for a different objective: log insertion for future postmortem diagnosis. Therefore many design aspects are unique to *Errlog*, such as checking whether the exception is logged, learning domain-specific errors, etc.

## 10 Conclusions

This paper answers a critical question: where is the proper location to print a log message that will best help postmortem failure diagnosis, without undue logging overhead? We comprehensively investigated 250 randomly sampled failure reports, and found a number of exception patterns that, if logged, could help diagnosis. We further developed *Errlog*, a tool that adds proactive logging code with only 1.4% logging overhead. Our controlled user study shows that the logs added by *Errlog* can speed up the failure diagnosis by 60.7%.

## Acknowledgements

We greatly appreciate anonymous reviewers and our shepherd, Florentina I. Popovici, for their insightful feedback. We are also grateful to the Opera group, Alex Rasmussen, and the UCSD System and Networking group for the useful discussions and paper proof-reading. We also thank all of the volunteers in the user study for their time spent in debugging. This research is supported by NSF CNS-0720743 grant, NSF CSR Small 1017784 grant and NetApp Gift grant.

## References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, 2003.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, 2007.

[3] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proc. of the ACM 22nd Symp. on Operating Systems Principles*, pages 193–206, 2009.

[4] ab - Apache HTTP server benchmarking tool. `http://goo.gl/NLAqZ`.

[5] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *Proc. of the 12th International Conference on on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2006.

[6] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 405–422, 2007.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX*

*Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[8] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. *SIGARCH Comput. Archit. News*, 36(1):319–328, Mar. 2008.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[10] Dell. Streamlined Troubleshooting with the Dell system E-Support tool. *Dell Power Solutions*, 2008.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the Symp. on Operating Systems Design & Implementation*, 2002.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[13] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.

[14] gcov – a test coverage program. `http://goo.gl/R9PoN`.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proc. of the ACM 22nd Symposium on Operating Systems Principles*, pages 103–116, 2009.

[16] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In *Proc. of the Symp. on Networked Systems Design & Implementation*, pages 239–252, 2011.

[17] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: error handling is occasionally correct. In *Proc. of the USENIX Conference on File and Storage Technologies*, pages 207–222, 2008.

[18] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. $G^2$: a graph processing system for diagnosing distributed systems. In *Proc. of the USENIX Annual Technical Conference*, pages 299–312, 2011.

[19] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[20] J.-C. Laprie. Dependable computing: concepts, limits, challenges. In *Proceedings of the 25th International Conference on Fault-tolerant Computing*, pages 42–54, 1995.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[22] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, Dec. 2011.

[23] Mozilla quality feedback agent. `http://goo.gl/V9zl2`.

[24] pgbench - PostgreSQL wiki. `goo.gl/GKhms`.

[25] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *Proc. of the 3rd Symp. on Networked Systems Design & Implementation*, pages 115–128, 2006.

[26] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, 2009.

[27] S. K. Sahoo, J. Criswell, and V. S. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proc. of the 32nd Intl. Conf. on Software Engineering*, pages 485–494, 2010.

[28] C. Spatz. Basic statistics, 1981.

[29] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 109–120, 2011.

[30] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., 21st International Symposium*, pages 2–9, 1991.

[31] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, 2011.

[32] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 117–132, 2009.

[33] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006.

[34] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 26–36, 2011.

[35] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.

[36] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proc. of the Intl. Conf. on Software Engineering*, pages 102–112, 2012.

[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, Feb. 2012.

[38] C. Zamfir and G. Candea. Low-overhead bug fingerprinting for fast debugging. In *Proceedings of the 1st International Conference on Runtime Verification*, pages 460–468, 2010.

# X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software

*Mona Attariyan*[†*], *Michael Chow*[†], *and Jason Flinn*[†]
*University of Michigan*[†] *Google, Inc.*[*]

## Abstract

Troubleshooting the performance of production software is challenging. Most existing tools, such as profiling, tracing, and logging systems, reveal *what* events occurred during performance anomalies. However, users of such tools must infer *why* these events occurred; e.g., that their execution was due to a root cause such as a specific input request or configuration setting. Such inference often requires source code and detailed application knowledge that is beyond system administrators and end users.

This paper introduces *performance summarization*, a technique for automatically diagnosing the root causes of performance problems. Performance summarization instruments binaries as applications execute. It first attributes performance costs to each basic block. It then uses dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause. Finally, it summarizes the overall cost of each potential root cause by summing the per-block cost multiplied by the cause-specific likelihood over all basic blocks. Performance summarization can also be performed differentially to explain performance differences between two similar activities. X-ray is a tool that implements performance summarization. Our results show that X-ray accurately diagnoses 17 performance issues in Apache, lighttpd, Postfix, and PostgreSQL, while adding 2.3% average runtime overhead.

## 1 Introduction

Understanding and troubleshooting performance problems in complex software systems is notoriously challenging. When a system does not perform as expected, system administrators and end users have few options. Explicit error messages are often absent or misleading [58]. Profiling and monitoring tools may reveal symptoms such as heavy usage of a bottleneck resource, but they do not link symptoms to root causes. Interpretation of application logs often requires detailed knowledge of source code or application behavior that is beyond a casual user. Thus, it is unsurprising that up to 20% of misconfigurations submitted for developer support are those that result in severe performance degradation [58] (the authors of this study speculate that even this number is an underestimate).

[*]This work was done when Mona Attariyan attended Michigan.

Why is troubleshooting so challenging for users? The most important reason is that current tools only solve half the problem. Troubleshooting a performance anomaly requires determining *why* certain events, such as high latency or resource usage, happened in a system. Yet, most current tools, such as profilers and logging, only determine *what* events happened during a performance anomaly. Users must manually infer the root cause from observed events based upon their expertise and knowledge of the software. For instance, a logging tool may detect that a certain low-level routine is called often during periods of high request latency, but the user must then infer that the routine is called more often due to a specific configuration setting. For administrators and end users who do not have intimate knowledge of the source code, log entries may be meaningless and the inference to root causes may be infeasible.

In this paper, we introduce a new tool, called X-ray, that helps users troubleshoot software systems without relying on developer support. X-ray focuses on attributing performance issues to root causes under a user's control, specifically configuration settings and program inputs. Why these causes? Numerous studies have reported that configuration and similar human errors are the largest source of errors in deployed systems [10, 11, 24, 25, 30, 32, 34, 58], eclipsing both software bugs and hardware faults. Further, errors such as software bugs cannot be fixed by end users alone.

X-ray does not require source code, nor does it require specific application log messages or test workloads. Instead, it employs binary instrumentation to monitor applications as they execute. It uses one of several metrics (request latency, CPU utilization, file system activity, or network usage) to measure performance costs and outputs a list of root causes ordered by the likelihood that each cause has contributed to poor performance during the monitored execution. Our results show that X-ray often pinpoints the true root cause by ranking it first out of 10s or 100s of possibilities. This is ideal for casual users and system administrators, who can now focus their troubleshooting efforts on correcting the specific input and parameters identified by X-ray.

X-ray introduces the technique of *performance summarization*. This technique first attributes performance costs to very fine-grained events, namely user-level instructions and system calls executed by the application.

Then, it uses dynamic information flow analysis to associate each such events with a ranked list of probable root causes. Finally, it summarizes the cost of each root cause over all events by adding the products of the per-event cost and an estimate of the likelihood that the event was caused by the root cause in question. The result is a list of root causes ordered by performance costs.

As described so far, performance summarization reveals which root causes are most costly during an entire application execution. For severe performance issues, such root causes are likely the culprit. However, some performance issues are more nuanced: they may occur only during specific time periods or affect some application operations but not others. Hence, X-ray provides several scoping options. Users may analyze performance during specific time periods, or they may look at a causal path such as a server's processing of a request.

X-ray also supports nuanced analysis via two additional summarization modes. In *differential performance summarization*, X-ray compares the execution of two similar operations and explains why their performance differs. For example, one can understand why two requests to a Web server took different amounts of time to complete even though the requested operations were identical. Differential performance analysis identifies branches where the execution paths of the requests diverge and assigns the performance difference between the two branch outcomes to the root causes affecting the branch conditionals. *Multi-input differential summarization* compares a potentially large number of similar operations via differential analysis and outputs the result as either a ranked list or a graphical explanation.

X-ray is designed to run in production environments. It leverages prior work in deterministic replay to offload the heavyweight analysis from the production system and execute it later on another computer. X-ray splits its functionality by capturing timing data during recording so that the data are not perturbed by heavyweight analysis. X-ray's replay implementation is flexible: it allows insertion of dynamic analysis into the replayed execution via the popular Pin tool [28], but it also enables low-overhead recording by not requiring the use of Pin or instrumentation code during recording.

Thus, this paper contributes the following:

- A demonstration that one can understand why performance issues are occurring in production software without source code, error and log messages, controlled workloads, or developer support.
- The technique of performance summarization, which attributes performance costs to root causes.
- The technique of differential performance summarization for understanding why two or more similar events have different performance.
- A deterministic replay implementation that enables both low-overhead recording and use of Pin binary instrumentation during replay.

Our evaluation reproduces and analyzes performance issues in Apache, lighttpd, Postfix, and PostgreSQL. In 16 of 17 cases, X-ray identifies a true root cause as the largest contributor to the performance problem; in the remaining case, X-ray ranks one false positive higher than the true root causes. X-ray adds only an average overhead of 2.3% on the production system because the bulk of its analysis is performed offline on other computers.

## 2 Related work

Broadly speaking, troubleshooting has three steps: detecting the problem, identifying the root cause(s), and solving the problem. X-ray addresses the second step.

Profilers [8, 13, 29, 35, 42, 45, 53], help detect a performance problem (the first step) and identify symptoms associated with the problem (which assists with the second step). They reveal *what* events (e.g., functions) incur substantial performance costs, but their users must manually infer *why* those events executed. Unlike X-ray, they do not associate events with root causes.

Similarly, most tools that target the second step (identifying the root cause) identify events associated with performance anomalies but do not explain why those events occur. Many such tools observe events in multiple components or protocol layers and use the observed causal relationships to propagate and merge performance data. X-trace [22] observes network activities across protocols and layers. SNAP [59] profiles TCP statistics and socket-call logs and correlates data across a data center. Aguilera *et al*. [1] infer causal paths between application components and attribute delays to specific nodes. Pinpoint [15, 16] traces communication between middleware components to infer which components cause faults and the causal paths that link black-box components. These tools share X-ray's observation that causality is a powerful tool for explaining performance events. However, X-ray distinguishes itself by observing causality *within* application components using dynamic binary instrumentation. This lets X-ray observe the relationship between component inputs and outputs. In contrast, the above tools only observe causality external to application components unless developers annotate code.

Other tools build or use a model of application performance. Magpie [7] extracts the component control flow and resource consumption of each request to build a workload model for performance prediction. Magpie's per-request profiling can help diagnose potential performance problems. Even though Magpie provides detailed performance information to manually infer root causes, it still does not automatically diagnose why the observed performance anomalies occur. Magpie uses schemas to determine which requests are being executed by high-level components; X-ray uses data and control flow analysis to map requests to lower-level events (instructions

and system calls) without needing schemas from its user.

Cohen *et al*. [19] build models that correlate system-level metrics and threshold values with performance states. Their technique is similar to profiling in that it correlates symptoms and performance anomalies but does not tie anomalies to root causes.

Many systems [14, 20, 60, 61] tune performance by injecting artificial traffic and using machine learning to correlate observed performance with specific configuration options. Unlike X-ray, these tools limit the number of options analyzed to deal with an exponential state space. Spectroscope [46] diagnoses performance changes by comparing request flows between two executions of the same workload. Kasick *et al*. [26] compare similar requests to diagnose performance bugs in parallel file systems. All of the above systems do not monitor causality within a request, so they must hold all but a single variable constant to learn how that variable affects performance. In practice, this is difficult because minor perturbations in hardware, workload, etc. add too much noise. In contrast, X-ray can identify root causes even when requests are dissimilar because it observes how requests diverge at the basic-block level.

Several systems are holistic or address the third step (fixing the problem). PeerPressure [54] and Strider [55] compare Windows registry state on different machines. They rely on the most common configuration states being correct since they cannot infer why a particular configuration fails. Chronus [56] compares configuration states of the same computer across time. AutoBash [50] allows users to safely try many potential configuration fixes.

X-ray uses a taint tracking [33] implementation provided by ConfAid [6] to identify root causes. ConfAid was originally designed to debug program failures by attributing those failures to erroneous configuration options. X-ray re-purposes ConfAid to tackle performance analysis. X-ray might instead have used other methods for inferring causality such as symbolic execution [12]. For instance, S2E [17] presented a case study in which symbolic execution was used to learn the relationship between inputs and low-level events such as page faults and instruction counts. Our decision to use taint tracking was driven both by performance considerations and our desire to work on COTS (common-off-the-shelf) binaries.

X-ray uses deterministic record and replay. While many software systems provide this functionality [2, 18, 21, 23, 36, 49, 52], X-ray's implementation has the unique ability to cheaply record an uninstrumented execution and later replay the execution with Pin.

## 3 X-ray overview

X-ray pinpoints why a performance anomaly, such as high request latency or resource usage, occurred. X-ray targets system administrators and other end users, though its automated inference should also prove useful to developers. Most of our experience to date comes from troubleshooting network servers, but X-ray's design is not limited to such applications.

X-ray does not require source code because it uses Pin [28] to instrument x86 binaries. X-ray users specify which files should be treated as configuration or input sources for an application. X-ray also treats any data read from an external network socket as an input source. As data from such sources are processed, X-ray recognizes configuration tokens and other root causes through a limited form of binary symbolic execution.

An X-ray user first records an interval of software execution. Section 6.5 shows that X-ray has an average recording overhead of 2.3%. Thus, a user can leave X-ray running on production systems to capture rare and hard-to-reproduce performance issues. Alternatively, X-ray can be used only when performance issues exhibit. X-ray defers heavyweight analysis to later, deterministically equivalent re-executions. This also allows analysis to be offloaded from a production system. Because X-ray analysis is 2–3 orders of magnitude slower than logging, we envision that only the portions of logs during which performance anomalies were observed will be analyzed.

For each application, an X-ray user must specify configuration sources such as files and directories, as well as a filter that determines when a new request begins.

For each analysis, an X-ray user selects a cost metric. X-ray currently supports four metrics: execution latency, CPU utilization, file system usage, and network use. X-ray also has a flexible interface that allows the creation of new metrics that depends on either observed timings or the instructions and system calls executed.

A user also specifies which interval of execution X-ray should analyze. The simplest method is to specify the entire recorded execution. In this case, X-ray returns a list of root causes ordered by the magnitude of their effect on the chosen cost metric. In our experience with severe performance issues, examining the entire execution interval typically generates excellent results.

However, some performance issues are nuanced. An issue may only occur during specific portions of a program's execution, or the problem may affect the processing of some inputs but not others. Therefore, X-ray allows its users to target the analysis scope. For instance, a user can specify a specific time interval for analysis, such as a period of high disk usage.

Alternatively, X-ray can analyze an application as it handles one specific input, such as a network request. X-ray uses both causal propagation through IPC channels and flow analysis to understand which basic blocks in different threads and processes are processing the input. It performs its analysis on only those basic blocks.

A user may also choose to compare the processing of two different inputs. In this case, X-ray does a differential performance summarization in which it first identifies the branches where the processing of the inputs di-

verged and then calculates the difference in performance caused by each divergence. We expect users to typically select two similar inputs that differ substantially in performance. However, our results show that X-ray provides useful data even when selected inputs are very dissimilar.

Finally, a user may select multiple inputs or all inputs received in a time period and perform an n-way differential analysis. In this case, X-ray can either return a ranked list of the root causes of pairwise divergences over all such inputs, or it can display the cost of divergences as a flow graph. We have found this execution mode to be a useful aid for selecting two specific requests over which to perform a more focused differential analysis.

Executions recorded by X-ray can be replayed and analyzed multiple times. Thus, X-ray users do not need to know which cost metrics and analysis scopes they will use when they record an execution.

## 4 Building blocks

X-ray builds on two areas of prior work: dynamic information flow analysis and deterministic record and replay. For each building block, we first describe the system on which we built and then highlight the most substantial modifications we have made to support X-ray.

### 4.1 Dynamic information flow analysis

#### 4.1.1 Background

X-ray uses taint tracking [33], a form of dynamic information flow analysis, to determine potential root causes for specific events during program execution. It uses ConfAid [6] for this purpose.

ConfAid reports the potential root cause of a program failure such as a crash or incorrect output. It assigns a unique taint identifier to registers and memory addresses when data is read into the program from configuration files. It identifies specific configuration tokens through a rudimentary symbolic execution that only considers string data and common (glibc) functions that compare string values. For instance, if data read from a configuration file is compared to "FOO", then ConfAid associates that data with token `FOO`.

As the program executes, ConfAid propagates taint identifiers to other locations in the process's address space according to dependencies introduced via data and control flow. ConfAid analyzes both direct control flow (values modified by instructions on the taken path of a branch depend on the branch conditional) and implicit control flow (values that would have been modified by instructions on paths not taken also depend on the branch conditional). Rather than track taint as a binary value, ConfAid associates a weight with each taint identifier that represents the strength of the causal relationship between the tainted value and the root cause. When ConfAid observes the failure event (e.g., a bad output), it outputs all root causes on which the current program control flow depends, ordered by the weight of that dependence. ConfAid employs a number of heuristics to estimate and limit causality propagation. For instance, data flow propagation is stronger than direct control flow, and both are stronger than indirect control flow. Also, control flow taint is aged gradually (details are in [6]).

#### 4.1.2 Modifications for X-Ray

One of the most important insights that led to the design of X-ray is that the marginal effort of determining the root cause of all or many events in a program execution is not substantially greater than the effort of determining the root cause of a single event. Because a taint tracking system does not know a-priori which intermediate values will be needed to calculate the taint of an output, it must calculate taints for *all* intermediate values. Leveraging this insight, X-ray differs from ConfAid in that it calculates the control flow taint for the execution of every basic block. This taint is essentially a list of root causes that express which, if any, input and configuration values caused the block to be executed; each root cause has a weight, which is a measure of confidence.

We modified ConfAid to analyze multithreaded programs. To limit the scope of analysis, when X-ray evaluates implicit control flow, it only considers alternative paths within a single thread. This is consistent with ConfAid's prior approach of bounding the length of alternate paths to limit exponential growth in analysis time. We also modified ConfAid to taint data read from external sources such as network sockets in addition to data read from configuration files. Finally, we modified ConfAid to run on either a live or recorded execution.

X-ray uses the same weights and heuristics for taint propagation that are used by ConfAid. We performed a sensitivity analysis, described in Section 6.4, on the effect of varying the taint propagation weights—the results showed that the precise choice of weights has little effect on X-ray, but the default ConfAid weights led to slightly more accurate results than other weights we examined.

### 4.2 Deterministic record and replay

X-ray requires deterministic record and replay for two reasons. First, by executing time-consuming analysis on a recording rather than a live execution, the performance overhead on a production system can be reduced to a few percent. Second, analysis perturbs the timing of application events to such a large degree that performance measurements are essentially meaningless. With deterministic replay, X-ray monitors timing during recording when such measurements are not perturbed by analysis, but it can still use the timing measurements for analysis during replay because the record and the replay are guaranteed to execute the same instructions and system calls.

#### 4.2.1 Background

Deterministic replay is well-studied; many systems record the initial state of an execution and log all non-

deterministic events that occur [2, 9, 21, 36, 49, 52, 57]. They reproduce the same execution, possibly on another computer [18], by restoring the initial state and supplying logged values for all non-deterministic events.

X-ray implements deterministic record and replay by modifying the Linux kernel and glibc library. It can record and replay multiple processes running on one or more computers. For each process, X-ray logs the order of and values returned by system calls and synchronization operations. It also records the timing of signals.

To record and replay multithreaded programs, one must also reproduce the order of all data races [44]. X-ray uses profiling to detect and instrument racing instructions. We execute an offline data race detector [51] on recorded executions. This race detector follows the design of DJIT+ [41]; it reports all pairs of instructions that raced during a recorded execution without false positives or false negatives. X-ray logs the order of the racing instructions during later recordings of the application. If no new data races are encountered after profiling, deterministic replay of subsequent executions is guaranteed. In the rare case where a new race is encountered, we add the racing instructions to the set of instrumented instructions on subsequent runs. Since the vast majority of data races do not change the order or result of logged operations [51], X-ray can often analyze executions with previously unknown data races. X-ray could also potentially search for an execution that matches observed output [2, 36].

#### 4.2.2 Modifications for X-ray

X-ray has a custom replay implementation because of our desire to use Pin to insert binary instrumentation into replayed executions. The simplest strategy would have been to implement record and replay with Pin itself [37]. However, we found Pin overhead too high; even with zero instrumentation, just running applications under Pin added a 20% throughput overhead for our benchmarks.

To reduce overhead, X-ray implements record and replay in the Linux kernel and glibc. Thus, Pin is only used during offline replay. This implementation faces a substantial challenge: from the point of view of the replay system, the replayed execution is not the same as the recorded execution because it contains additional binary instrumentation not present during recording. While Pin is transparent to the application being instrumented, it is *not* transparent to lower layers such as the OS.

X-ray's replay system is *instrumentation-aware*; it compensates for the divergences in replayed execution caused by dynamic instrumentation. Pin makes many system calls, so X-ray allocates a memory area that allows analysis tools run by Pin to inform the replay kernel which system calls are initiated by the application (and should be replayed from the log) and which are initiated by Pin or the analysis tool (and should execute normally).

X-ray also compensates for interference between resources requested by the recorded application and resources requested by Pin or an analysis tool. For instance, Pin might request that the kernel `mmap` a free region of memory. If the kernel grants Pin an arbitrary region, it might later be unable to reproduce the effects of a recorded application `mmap` that returns the same region. X-ray avoids this trap by initially scanning the replay log to identify all regions that will be requested by the recorded application and pre-allocating them so that Pin does not ask for them and the kernel does not return them. X-ray also avoids conflicts for signal handlers, file handles, and System V shared memory identifiers.

Finally, the replay system must avoid deadlock. The replay system adds synchronization to reproduce the same order of system calls, synchronization events, and racing instructions seen during recording. Pin adds synchronization to ensure that application operations such as memory allocation are executed atomically with the Pin code that monitors those events. X-ray initially deadlocked because it was unaware of Pin locking. To compensate, X-ray now only blocks threads when it knows Pin is not holding a lock; e.g., rather than block threads executing a system call, it blocks them prior to the instruction that follows the system call.

## 5 Design and implementation

X-ray divides its execution into online and offline phases. The offline phase is composed of multiple replayed executions. This design simplifies development by making it easy to compose X-ray analysis tools out of modular parts.

### 5.1 Online phase

Since the online phase of X-ray analysis runs on a production system, X-ray uses deterministic record and replay to move any activity with substantial performance overhead to a subsequent, offline phase. The only online activities are gathering performance data and logging system calls, synchronization operations and known data races.

X-ray records timestamps at the entry and exit of system calls and synchronization operations. It minimizes overhead by using the x86 timestamp counter and writing timestamps to the same log used to store non-deterministic events. The number of bytes read or written during I/O is returned by system calls and hence captured as a result of recording sources of non-determinism.

### 5.2 First offline pass: Scoping

The first offline pass maps the scope of the analysis selected by an X-ray user to a set of application events. While X-ray monitors events at the granularity of user-level instructions and system calls, it is sufficient to identify only the basic blocks that contain those events since the execution of a basic block implies the execution of all events within that block.

A user may scope analysis to a time interval or to the processing of one of more inputs. If the user specifies a time interval, X-ray includes all basic blocks executed by any thread or process within that interval. If the user scopes the analysis to one or more inputs, X-ray identifies the set of basic blocks that correspond to the processing of each input via *request extraction*.

### 5.2.1 Request extraction

Request extraction identifies the basic block during which each request (program input) was processed. For the applications we have examined to date, inputs are requests received over network sockets. However, the principles described below apply to other inputs, such as those received via UI events.

Since the notion of a request is application-dependent, X-ray requires a filter that specifies the boundaries of incoming requests. The filter is a regular expression that X-ray applies to all data read from *external sockets*, which we define to be those sockets for which the other end point is not a process monitored by X-ray. For instance, the Postfix filter looks for the string `HELO` to identify incoming mail. Only one filter must be created for each protocol (e.g., for SMTP or HTTP).

Request extraction identifies the causal path of each request from the point in application execution when the request is received to the point when the request ends (e.g., when a response is sent). X-ray supports two methods to determine the causal path of a request within process execution. These methods offer a tradeoff between generality and performance. Both are implemented as Pin tools that are dynamically inserted into binaries.

The first method is designed for simple applications and multi-process servers. It assumes that a process handles a single request at a time, but it allows multiple processes to concurrently handle different requests (e.g., different workers might simultaneously process different requests). When a new request is received from an external socket, X-ray taints the receiving process with a unique identifier that corresponds to the request. X-ray assumes that the process is handling that request until the process receives a message that corresponds to a different request or until the request ends (e.g., when the application closes the communcation socket). A new taint may come either from an external source (in which case, it is detected by the input data matching the request filter), or it may come from an internal source (another process monitored by X-ray), in which case the request taint is propagated via the IPC mechanism described below.

The second method directly tracks data and control flow taint. When a request is received from an external socket, X-ray taints the return codes and data modified by the receiving system call with the request identifier. X-ray propagates taint within an address space as a process executes. It assigns each basic block executed by the process to at most one request based on the control flow taint of the thread at the time the basic block is executed. Untainted blocks are not assigned to a request. A block tainted by a single identifier is assigned to request corresponding to that identifier. A block tainted by multiple identifiers is assigned to the request whose taint identifier has the highest weight; if multiple identifiers have the same weight, the block is assigned to the request that was received most recently.

Comparing the two methods, the first method has good performance and works well for multi-process servers such as Postfix and PostgreSQL. However, it is incapable of correctly inferring causal paths for multithreaded applications and event-based applications in which a single process handles multiple requests concurrently. The second method handles all application types well but runs slower than the first method. X-ray uses the second method by default, but users may select the first method for applications known to be appropriate.

Request extraction outputs a list of the basic blocks executed by each request. Each block is denoted by a <id, address, count> tuple. The *id* is the Linux identifier of the thread/process during recording, *address* is the first instruction of the block in the executable, and *count* is the number of instructions executed by the process prior to the first instruction of the basic block. Thus, *count* differentiates among multiple dynamic executions of a static basic block. Since deterministic replay executes exactly the same sequence of application instructions, the *count* of each block matches precisely across multiple replays and, thus, serves as a unique identifier for the block during subsequent passes.

### 5.2.2 Inter-process communication

Replayed processes read recorded data from logs rather than from actual IPC channels. X-ray establishes separate mechanisms, called *side channels*, to communicate taint between processes and enforce the same causal ordering on replayed processes that was observed during the original recording. For instance, a process that blocked to receive a message on a socket during recording will block on the side channel during replay to receive the taint associated with the message.

Side channels propagate taint from one address space to another. X-ray supports several IPC mechanisms including network and local sockets, files, pipes, signals, fork, exit, and System V semaphores. During replay, when a recorded system call writes bytes to one of these mechanisms, X-ray writes the data flow taint of those bytes to the side channel. X-ray merges that taint with the control flow taint of the writing thread. Even mechanisms that do not transfer data (e.g., signals) still transfer control flow taint (e.g., the control flow of the signal handler is tainted with the control flow taint of the signaler).

When replay is distributed, one computer acts as the replay master. Processes running on other computers register with the master; this allows each replay process to

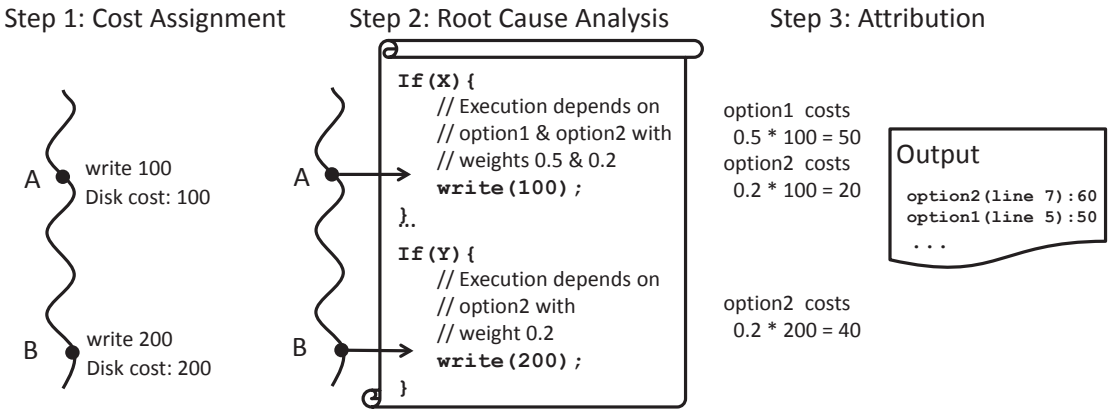



**Figure 1**. Example of performance summarization

determine which sockets are external and which connect to other replay processes. For simplicity, all side channels pass through the master, so replay processes on other computers read to and write from side channels by making RPCs to a server running on the master.

#### 5.2.3 Attributing performance costs

During the first pass, X-ray also attributes performance costs to all *events* (application instructions and system calls executed) within the chosen scope. As a performance optimization, all events within a single basic block are attributed en masse because all are executed if the block is executed.

Recall that X-ray users may currently choose latency, CPU utilization, file system use, or network throughput as cost metrics. The latency of each system call and synchronization operation is recorded during online execution. X-ray attributes the remaining latency to user-level instructions. From the recorded timestamps in the log, it determines the time elapsed between each pair of system calls and/or synchronization events. X-ray dynamic instrumentation counts the number of user-level instructions executed in each time period. It divides the two values to estimate the latency of each instruction.

To calculate CPU utilization, X-ray counts the instructions executed by each basic block. To calculate file system and network usage, it observes replayed execution to identify file descriptors associated with the resource being analyzed. When a system call accesses such descriptors, X-ray attributes the cost of the I/O operation to the basic block that made the system call.

### 5.3 Second pass: performance summarization

Performance summarization, in which costs are attributed to root causes, is performed during the second execution pass. X-ray currently supports three modes: (1) basic summarization, which computes the dominant root causes over some set of input basic blocks, (2) differential summarization, which determines why the processing of one input had a different cost than the processing of another input, and (3) multi-request differential summarization, which computes the differential cost across three or more inputs.

#### 5.3.1 Basic performance summarization

Basic performance summarization individually analyzes the per-cause performance cost and root cause of all events. It then sums the per-event costs to calculate how much each root cause has affected application performance.

Figure 1 shows how basic performance summarization works. In the first pass, X-ray determines which basic blocks are within the analysis scope and assigns a performance cost to the events in each block. In the second pass, X-ray uses taint tracking to calculate a set of possible root causes for the execution of each such block. Essentially, this step answers the question: "how likely is it that changing a configuration option or receiving a different input would have prevented this block from executing?" X-ray uses ConfAid to track taints as weights that show how strongly a particular root cause affects why a byte has its current value (data flow) or why a thread is executing the current instruction (control flow).

X-ray next attributes a per-block cost to each root cause. This attribution is complicated by the fact that ConfAid returns only an ordered list of potential root causes. Weights associated with causes are relative metrics and do not reflect the actual probability that each cause led to the execution of a block. We considered several strategies for attribution:

- **Absolute weight.** The simplest strategy multiplies each per-cause weight by the per-block performance cost. This is an intuitive strategy since X-ray, like ConfAid, aims only to achieve a relative ranking of causes.
- **Normalized weight.** The weights for a block are normalized to sum to one before they are multiplied by the performance cost. This strategy tries to calculate an absolute performance cost for each cause. However, it may strongly attribute a block to

a root cause in cases where the block's execution is likely not due to *any* root cause.

- **Winner-take-all.** The entire per-block cost is attributed to the highest ranking cause (or equally shared in the case of ties).
- **Learned weights.** Based on earlier ConfAid results [6], we calculate the probability that a cause ranked 1st, 2nd, 3rd, etc. is the correct root cause. We use these probabilities to weight the causes for each block according to their relative rankings. Note that the benchmarks used for deciding these weights are completely disjoint from the benchmarks used in this paper.

Based on the sensitivity study reported in Section 6.4, we concluded that X-ray results are robust across all attribution strategies that consider more than just the top-ranked root cause. X-ray uses the absolute weight strategy by default because it is simple and it had slightly better results in the study.

X-ray calculates the total cost for each root cause by summing the per-block costs for that cause over all basic blocks within the analysis scope; e.g., in Figure 1, the per-block costs of `option2` are 20 and 40, so its total cost is 60). X-ray outputs potential root causes as a ranked list ordered by cost; each list item shows the token string, the config file or input source, the line number within the file/source, and the total cost.

### 5.3.2 Differential performance summarization

Differential performance summarization compares any two executions of an application activity, such as the processing of two Web requests. Such activities have a common starting point (e.g., the instruction receiving the request), but their execution paths may later diverge.

Figure 2 shows an example of differential performance summarization. X-ray compares two activities by first identifying all points where the paths of the two executions diverge and merge using longest common sub-sequence matching [31]. In the figure, the execution paths of the activities are shown by the solid and dashed lines, and the conditional branches where the paths diverge are denoted as C1 and C2.

X-ray represents the basic blocks that processed each request as a string where each static basic block is a unique symbol. Recorded system calls and synchronization operations give a partial order over blocks executed by multiple threads and processes. Any blocks not so ordered executed concurrently; they do not contain racing instructions. X-ray uses a fixed thread ordering to generate a total order over all blocks (the string) that obeys the recorded partial order. The matching algorithm then identifies divergence and merge points.

X-ray uses taint tracking to evaluate why each divergence occurred. It calculates the taint of the branch conditional at each divergence point. Note that since X-ray uses dynamic analysis, loops are simply treated as a series of branch conditionals (that happen to be the same instruction). The cost of a divergence is the difference between the performance cost of all basic blocks on the divergent path taken by the first execution and the cost of all blocks on the path taken by the second execution. As in the previous section, X-ray uses the absolute weight method by default. Finally, X-ray sums the per-cause costs over all divergences and outputs a list of root causes ordered by differential cost. In Figure 2, `option2` is output before `option1` because its total cost is greater.

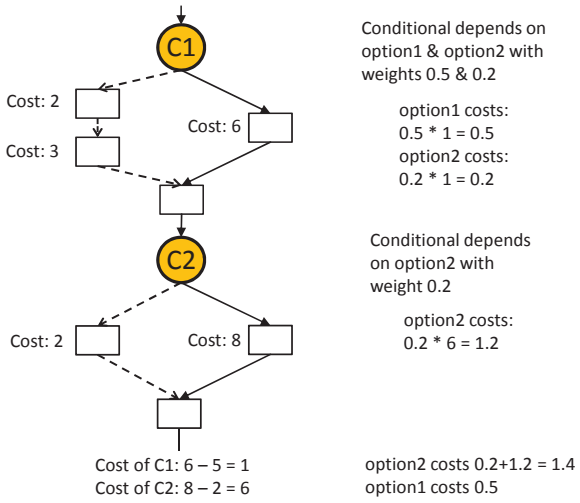



**Figure 2**. Differential performance summarization

### 5.3.3 Multi-input differential summarization

Pairwise differential summarization is a powerful tool, but it is most useful if an X-ray user can identify two similar inputs that have markedly different performance. To aid the user in this task, X-ray has a third performance summarization mode that can graphically or numerically compare a large number of inputs.

Multi-input summarization compares inputs that match the same filter. The processing path of these inputs begins at the same basic block (containing the system call that receives the matching data). The subsequent processing paths of the various inputs split and merge. Typically, the paths terminate at the same basic block (e.g., the one closing a connection). If they do not, X-ray inserts an artificial termination block that follows the last block of each input. This allows the collection of input paths to be viewed as a lattice, as shown in Figure 3 for an example with three unique paths.

X-ray discovers this lattice by first representing each input path as a string (as it does for pairwise differential analysis). It then executes a greedy longest common sub-sequence matching algorithm [31] in which it first merges the two strings with the smallest edit distance to form a common path representation, then merges the closest remaining input string with the common representation, etc. The common representation is a graph

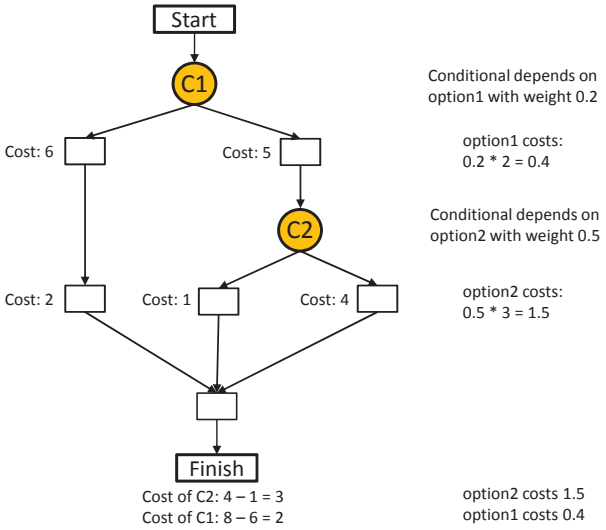



**Figure 3**. Multi-input differential summarization

where each vertex is a branch at which input paths diverged or a merge point at which input paths converged. Each edge is a sub-path consisting of the basic blocks executed for all inputs that took a common sub-path between two vertexes.

Next, X-ray determines the cost of all divergences. Intuitively, the cost (or benefit) of taking a particular branch should be the difference between the lowest-cost execution that can be achieved by the path taken subtracted from the lowest-cost execution that can be achieved by the path not taken. The weight of a graph edge is the sum of the costs for each block along the edge averaged over all requests that executed the blocks in that edge (e.g., this might be calculated by summing the average latency for each block). X-ray calculates the shortest path on the reverse graph from the termination node to every divergence node for each possible branch. At each branch where paths diverged, X-ray calculates the cost of taking a particular branch to be the difference between the shortest path if that branch is taken and the shortest path from any branch available at that node. For instance, in Figure 3, the cost of conditional branch C2 is 3 (subtracting the cost of the right branch from the left). For C1, the cost is 2 because the shortest path taking the left branch is 8 and the shortest path taking the right branch is 6. The per-divergence cost is then merged with the per-root-cause taints of the branch conditional.

X-ray offers two modes for displaying this data. The first is a numerical summarization that integrates the per-cause costs over all divergences in the graph and displays all root causes in order of total cost. The second method shows the lattice graph, with each divergence node displaying the cost and reasons for the divergence, and the width of each edge representing the number of inputs that traversed the particular sub-path. An X-ray user can use this graph to identify the most surprising or costly divergences, then select two representative inputs that took opposite branches at that point for a more detailed pairwise comparison. The simpler ordered list is appropriate for casual users. The richer graphical output may be best for power users (or possibly developers).

Multi-path analysis sometimes produced erroneous results due to infeasible shortest paths. These paths arise because X-ray uses taint tracking rather than symbolic analysis. Consider two divergences points that test the same condition. If the `true` outcome has the shortest path after the first test and the `false` outcome has the shortest path after the second test, the shortest path is infeasible because the same condition cannot evaluate to two different values. X-ray uses a statistical analysis to infer infeasible paths. Given a sufficient set of input path samples that pass through two divergence vertexes, if the partition generated by the branch each path took at the first vertex is isomorphic to the partition generated by the branch each took at the second vertex, X-ray infers that the two divergences depend on the same condition and does not allow a shortest path that takes a conflicting path through the two vertexes.

### 5.4 Fast forward

For long-running applications, replaying days of execution to reach a period of problematic performance may be infeasible. Periodic checkpointing of application state is insufficient because most applications read configuration files at the start of their execution. Thus, the execution after a checkpoint is missing data needed to trace problems back to configuration root causes.

X-ray uses a *fast forward* heuristic to solve this problem. After configuration data is read, X-ray associates dirty bits with each taint to monitor the amount of taint changing during request handling. When less than `n`% of taint values have been changed by the first `n` requests after reading a taint source, X-ray considers configuration processing to have quiesced. It saves the current taint values and fast forwards execution to a point that is at least `n` requests prior to the period of execution being investigated (or to the next opening of a configuration file). It restores saved taints into the address space(s) of the application and resumes instrumented execution.

Use of the fast forward heuristic is optional because it may lead to incorrect results when configuration values are modified as a result of processing requests unmonitored by X-ray. However, we have not seen this behavior in any application to date.

## 6 Evaluation

Our evaluation answers the following questions:

- How accurately does X-ray identify root causes?
- How fast can X-ray troubleshoot problems?
- How much overhead does X-ray add?

| Application | Test | Description of performance test cases |
|---|---|---|
| Apache | 1 | The number of requests that can be handled in one TCP connection is set too low. Reestablishing connections delays some requests [5]. |
| | 2 | Directory access permissions are based on the domain name of the client sending the request, leading to extra DNS lookups [4]. |
| | 3 | Logging the host names of clients sending requests to specific directories causes extra DNS requests for files in those directories [4]. |
| | 4 | Authentication for some directories causes high CPU usage peaks when files in those directories are accessed [3]. |
| | 5 | Apache can be configured to generate content-MD5 headers calculated using the message body. This header provides an end-to-end message integrity with high confidence. However, for larger files, the calculation of the digests causes high CPU usage [27]. |
| | 6 | By default, Apache sends eTags in the header of HTTP responses that can be used by clients to avoid resending data in the future if file contents have not changed. However, many mobile phone libraries do not correctly support this option [43]. |
| Postfix | 1 | Logging more information for a list of specific hosts causes excessive disk activity when one host is the computer running Postfix [38]. |
| | 2 | Postfix can be configured to examine the body of the messages against a list of regular expressions known to be from spammers or viruses. This setting can significantly increase the CPU usage for handling a received message if there are many expression patterns [40]. |
| | 3 | Postfix can be configured to reject requests from blacklisted domains. Based on the operators specified, Postfix performs extra DNS calls, which significantly increases message handling latency [39]. |
| PostgreSQL | 1 | PostgreSQL tries to identify the correct time zone of the system for displaying and interpreting time stamps if the time zone is not specified in the configuration file. This increases the startup time of PostgreSQL by a factor of five. |
| | 2 | PostgreSQL can be configured to synchronously commit the write-ahead logs to disk before sending the end of the transaction message to the client. This setting can cause extra delays in processing transactions if the system is under a large load [48]. |
| | 3 | The frequency of taking checkpoints from the write-ahead log can be configured in the PostgreSQL configuration file. More frequent checkpoints decrease crash recovery time but significantly increase disk activity for busy databases [47]. |
| | 4 | Setting the delay between activity rounds for the write-ahead log writer process causes excessive CPU usage [47]. |
| | 5 | A background process aggressively collects database statistics, causing inordinate CPU usage [47]. |
| lighttpd | 1 | Equivalent to Apache bug 1. |
| | 2 | Equivalent to Apache bug 4. |
| | 3 | Equivalent to Apache bug 6. |

**Table 1**. Description of the performance test cases used for evaluation

## 6.1 Experimental Setup

We used X-ray to diagnose performance problems in four servers with diverse concurrency models: the Apache Web server version 2.2.14, the Postfix mail server version 2.7, the PostgreSQL database version 9.0.4, and the lighttpd Web server version 1.4.30. Apache is multithreaded; new connections are received by a listener thread and processed by worker threads. In Postfix, multiple utility processes handle each part of a request; on average, a request is handled by 5 processes. In PostgreSQL, each request is handled by one main process, but work is offloaded in batch to utility processes such as a write-ahead log writer. The lighttpd Web server is event-driven; one thread multiplexes handling of multiple concurrent requests using asynchronous I/O. We ran all experiments on a Dell OptiPlex 980 with a 3.47 GHz Intel Core i5 Dual Core processor and 4 GB of memory, running a Linux 2.6.26 kernel modified to support deterministic replay.

## 6.2 Root cause identification

We evaluated X-ray by reproducing 16 performance issues (described in Table 1) culled from the cited performance tuning and troubleshooting books, Web documentation, forums, and blog posts. To recreate each issue, we either modified configuration settings or sent a problematic sequence of requests to the server while we recorded server execution. We also used X-ray to troubleshoot an unreported performance issue (described below) that was hampering our evaluation.

For each test, Table 2 shows the scope and metric used for X-ray analysis. The metric was either suggested by the problem description or a bottleneck resource identified by tools such as `top` and `iostat`. The next column shows where true root cause(s) of the problem were ranked by X-ray. X-ray considered on average 120 possible root causes for the Apache tests, 54 for Postfix, 54 for PostgreSQL, and 48 for lighttpd (these are the average number of tokens parsed from input and configuration files). The last column shows how long X-ray offline analysis took. The reported results do not use the fast-forward heuristic—however, X-ray achieves the same results when fast-forward is enabled.

Our overall results were very positive. X-ray ranked the true root cause(s) first in 16 out of 17 tests. In several cases, multiple root causes contribute to the problem, and X-ray ranked all of them before other causes. In two of the above cases, the true root cause is tied with one or two false positives. In the remaining test, X-ray ranked one false positive higher than the true root causes. Further, the analysis time is quite reasonable when compared to the time and effort of manual analysis: X-ray took 2 minutes and 4 seconds on average to identify the root cause, and no test required more than 9 minutes of analysis time. We next describe a few tests in more detail.

### 6.2.1 Apache

Apache test 1 shows the power of differential analysis. The threshold for the number of requests that can reuse the same TCP connection is set too low, and reestablishing connections causes a few requests to exhibit higher latency. To investigate, we sent 100 various requests to the Apache server using the *ab* benchmarking tool. The requests used different HTTP methods (GET

| Application | Test | Analysis scope | Analysis metric | Correct root cause(s) (rank) | Analysis time |
|---|---|---|---|---|---|
| Apache | 1 | Differential | Latency | **MaxKeepAliveRequests** (1st) | 0m 44s |
| | 2 | Differential | Latency | **Allow** (t-1st), **domain** (t-1st) | 0m 40s |
| | 3 | Differential | Latency | **On** (1st), **HostNameLookups** (2nd) | 0m 43s |
| | 4 | Request | CPU | **AuthUserFile** (1st) | 0m 43s |
| | 5 | Differential | CPU | **On** (1st), **ContentDigest** (2nd) | 0m 44s |
| | 6 | Differential | Network | **Input(eTag)** (t-1st) | 0m 42s |
| Postfix | 1 | Request | File system | **debug_peer_list** (t-1st), **domain** (t-1st) | 8m 10s |
| | 2 | Request | CPU | **body_checks** (1st) | 2m 38s |
| | 3 | Request | Latency | **reject_rbl_client** (1st) | 2m 18s |
| PostgreSQL | 1 | Time interval | CPU | **timezone** (1st) | 6m 59s |
| | 2 | Request | Latency | **wal_sync_method** (2nd), **synchronous_commit** (3rd) | 2m 04s |
| | 3 | Time interval | File system | **checkpoint_timeout** (1st) | 3m 06s |
| | 4 | Time interval | CPU | **wal_writer_delay** (1st) | 2m 33s |
| | 5 | Time interval | CPU | **track_counts** (1st) | 1m 51s |
| lighttpd | 1 | Differential | Latency | **auth.backend.htpasswd.userfile** (1st), | 0m 34s |
| | 2 | Request | CPU | **Input(eTag)** (t-1st), | 0m 24s |
| | 3 | Differential | Network | **server.max-keep-alive-requests** (1st), | 0m 24s |

This table shows the type of X-ray analysis performed, the ranking of all true root causes in the ordered list returned by X-ray and X-ray's execution time. The notation, t-1st, shows that the cause was tied for first.

**Table 2**. X-ray results

and POST) and asked for files of different sizes.

We used X-ray to perform a differential summarization of two similar requests (HTTP GETs of different small files), one of which had a small latency and one of which had a high latency. X-ray identified the `MaxKeepAliveRequests` configuration token as the highest-ranked contributor out of 120 tokens. Based on this information, an end user would increase the threshold specified by that parameter; we verified that this indeed eliminates the observed latency spikes. In the next section, we vary the requests compared for this test to examine how the accuracy of differential analysis depends on the similarity of inputs.

In Apache test 6, the root cause of high network usage is the client's failure to use the HTTP conditional eTag header. A recent study [43] found that many smartphone HTTP libraries do not support this option, causing redundant network traffic. X-ray identifies this problem via differential analysis, showing that it can sometimes identify bad client behavior via analysis of servers. We verified that correct eTag support substantially reduces network load.

#### 6.2.2 Postfix

The first Postfix test reproduces a problem reported in a user's blog [38]—emails with attachments sent from his account transferred very slowly, while everything else, including mail received by IMAP services, had no performance issues. Using `iotop`, the user observed that one child process was generating a lot of disk activity. He poured through the server logs and saw that the child process was logging large amounts of data. Finally, he scanned his configuration file and eventually realized that the `debug_peer_list`, which specifies a list of hosts that trigger logging, included his own IP address. Like many configuration problems, the issue is obvious once explained, yet even an experienced user still spent hours identifying the problem. Further, this level of analysis is beyond inexperienced users.

In contrast, X-ray quickly and accurately pinpoints the root cause. We simply analyzed requests during a period of high disk usage. X-ray identified the `debug_peer_list` parameter and a token corresponding to our network domain as the top root causes. Since changing either parameter fixes the problem, the user described above could have saved much time with this important clue. Also, no manual analysis such as reading log files was required, so even an inexperienced user could benefit from these results.

#### 6.2.3 PostgreSQL

The first PostgreSQL test is from our own experience. Our evaluation started and stopped PostgreSQL many times. We noticed that our scripts ran slowly due to application start-up delay, so we used X-ray to improve performance. Since `top` showed 100% CPU usage, we performed an X-ray CPU analysis for the interval prior to PostgreSQL receiving the first request.

Unexpectedly, X-ray identified the `timezone` configuration token as the top root cause. In the configuration file, we had set the `timezone` option to `unknown`, causing PostgreSQL to expend a surprising amount of effort to identify the correct time zone. Based on this clue, we specified our correct time zone; we were pleased to see PostgreSQL startup time decrease by over 80%. Admittedly, this problem was esoteric (most users do not start and stop PostgreSQL repeatedly), but we were happy that X-ray helped solve an unexpected performance issue.

In PostgreSQL test 2, X-ray ranked the `shared_buffers` configuration token higher than both true root causes. Manual analysis showed that this token controls the number of database buffers and hence is tested repeatedly by the loop that initializes those buffers. This adds a control flow taint to all database buffers that does not age rapidly due to the large number of such buffers. Such taint could be eliminated by

specifically identifying initialization logic, but we have yet to determine a sound method for doing so.

#### 6.2.4 lighttpd

We chose to evaluate lighttpd to stress X-ray's flow analysis with an event based server in which one thread handles many concurrent requests. Three of the bugs that we examined for Apache have no clear analog in lighttpd. For the remaining three bugs, we introduced similar problems in lighttpd by modifying its configuration file and/or sending erroneous input. X-ray ranked the true root cause first in two tests; in the remaining test, the true root cause was tied for first with two other parameters. From this, we conclude that the flow-based request identification works well with both multithreaded (Apache) and event-based (lighttpd) programs.

### 6.3 Differential analysis

Experimental methods for analyzing differential performance [14, 26, 46] often require that inputs be identical except for the variable being examined. Unlike these prior methods, X-ray's differential analysis analyzes application control flow and determines the root cause for each divergence between processing paths. Our hypothesis is that this will enable X-ray to generate meaningful results even for inputs that have substantial differences.

To validate this hypothesis, we repeated the first Apache test. Instead of selecting similar requests, we selected the pair of requests that were most different: a small HTTP POST that failed and a very large HTTP GET that succeeded. Somewhat surprisingly, X-ray still reported `MaxKeepAliveRequests` as the top root cause. The reason was a bit fortuitous: in our benchmark, the `MaxKeepAliveRequests` option happened to increase the latency of the small request, so the latency due to the misconfiguration exhibited as a performance degradation, while the difference in request input exhibited as a performance improvement.

We verified this by reversing the order of the two requests so that the large request was slowed by connection re-establishment rather than the small request. In this case, X-ray reported differences in the input request data as the largest reason why the processing of the large request is slower. It incorrectly ranked the `DocumentRoot` parameter second because the root is appended to the input file name before data is read from disk. `MaxKeepAliveRequests` ranked third.

We conclude that differential analysis does not always require that two requests be substantially similar in order to identify root causes of performance anomalies. Differences in input will of course rise to the top of the ranked list. However, after filtering these causes out, the true root cause was still ranked very near the top in our experiment, so a user would not have had to scan very far.

Finally, we applied multi-request differential analysis to this test by sending 100 requests of varying types (GET and POST), sizes, and success/failure outcomes. When we compared all 100 requests and filtered out input-related causes, the true root cause was ranked second (behind the `ServerRoot` token). For an end user, this mode is especially convenient because the user need not identify specific requests to compare. For power users and developers, the graphical version of the multi-path output shows the specific requests for which `MaxKeepAliveRequests` causes path divergences.

| Strategy | False positives | | | | True cause |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3+ | unranked |
| Absolute | 21 | 2 | 0 | 0 | 0 |
| Normalized | 20 | 0 | 3 | 0 | 0 |
| Winner-take-all | 15 | 3 | 1 | 2 | 2 |
| Learning | 20 | 2 | 1 | 0 | 0 |

For each strategy, this shows the number of false positives ranked above each of the 23 true root causes from Table 2. The winner-take-all strategy failed to identify 2 true root causes.

**Table 3**. Accuracy of attribution strategies

### 6.4 Sensitivity analysis

Section 5.3.1 described four strategies for attributing performance to root causes. Table 3 summarizes the results of running all tests in Section 6.2 with each strategy. There are 23 true root causes in the 17 tests. The second column shows the number of these for which no false positive is higher in the final X-ray rankings. The next column shows the number for which 1 false positive is ranked higher, etc. The final column shows true causes that did not appear at all in X-ray's rankings.

The winner-take-all strategy is substantially worse than the other strategies because the true root cause ranks second or third for many basic blocks, and so its impact is underestimated. All other strategies are roughly equivalent, with the absolute strategy being slightly more accurate than the other two. We conclude that the X-ray algorithm is not very sensitive to the particular attribution algorithm as long as that algorithm considers more than just the top cause for each basic block.

As described in Section 4.1, X-ray uses ConfAid's taint aging heuristics: control flow taint is multiplied by a weight of 0.5 when it is merged with data flow taint or when a new tainted conditional branch is executed. We performed a sensitivity analysis, shown in Table 4, that examined the effect of changing this weight. Note that a weight of 0 is equivalent to the winner-take-all strategy, and a weight of 1 does not age taint at all. While the default weight of 0.5 produced slightly better results than other weights, all values within the range 0.125–0.875 had roughly equivalent results in our experiments.

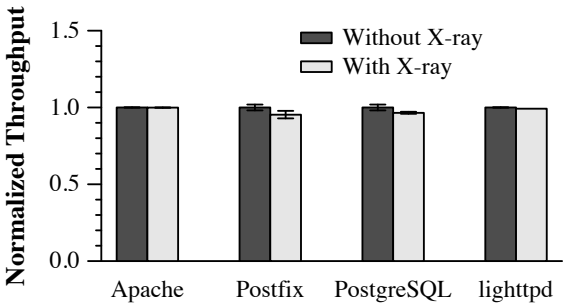
### 6.5 X-ray online overhead

We measured online overhead by comparing the throughput and latency of applications when they are

| Weight | False positives | | | | True cause |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3+ | unranked |
| 0 | 15 | 3 | 1 | 2 | 2 |
| 0.125 | 19 | 2 | 2 | 0 | 0 |
| 0.25 | 20 | 3 | 0 | 0 | 0 |
| 0.5 | 21 | 2 | 0 | 0 | 0 |
| 0.75 | 20 | 0 | 1 | 2 | 0 |
| 0.875 | 20 | 0 | 1 | 2 | 0 |
| 1 | 8 | 3 | 2 | 10 | 0 |

For each weight, this shows the number of false positives ranked above each of the 23 true root causes from Table 2.

**Table 4**. Accuracy when using different weights




Each dataset shows server throughput with and without X-ray recording, normalized to the number of requests per second without X-ray. Higher values are better. Each result is the mean of at least 10 trials; error bars are 95% confidence intervals.

**Figure 4**. X-ray online overhead

recorded by X-ray to results when the applications run on default Linux without recording. For Apache and lighttpd, we used ab to send 5000 requests for a 35 KB static Web page with a concurrency of 50 requests at a time over an isolated network. For Postfix, we used smtp-source to send 1000 64 KB mail messages. For PostgreSQL, we used pgbench to measure the number of transactions completed in 60 seconds with a concurrency of 10 transactions at a time. Each transaction has one `SELECT`, three `UPDATE`s, and one `INSERT` command.

Figure 4 shows X-ray adds an average of 2.3% throughput overhead: 0.1% for Apache, 4.7% for Postfix, 3.5% for PostgreSQL, and 0.8% for lighttpd. These values include the cost of logging data races previously detected by our offline data race detector. This overhead is consistent with similar deterministic replay approaches [18]. Latency overheads for Apache, PostgreSQL, and lighttpd are equivalent to the respective throughput overheads; Postfix has no meaningful latency measure since its processing is asynchronous. The recording log sizes were 2.8 MB for Apache, 1.6 MB for lighttpd, 321 MB for PostgreSQL, and 15 MB for Postfix. Apache and lighttpd have smaller logs because they use `sendfile` to avoid copying data.

### 6.6 Discussion

X-ray's accuracy has exceeded our expectations. One reason for this is that many performance issues, like the Postfix example in Section 6.2.2, are obvious once explained. Without explanation, however, searching for the root cause is a frustrating, "needle-in-a-haystack" process. Performance summarization is essentially a brute-force method for searching through that haystack. The obvious-once-explained nature of many performance problems has another nice property: X-ray's accuracy is not very sensitive to the exact heuristics it employs, so many reasonable choices lead to good results.

X-ray's most significant limitation is that it does not track taint inside the OS so it cannot observe causal dependencies among system calls. For instance, X-ray cannot determine when one thread blocks on a kernel queue waiting for other threads or kernel resources. In addition, system call parameter values often affect the amount of work performed by the kernel. X-ray currently addresses this on an ad-hoc basis; e.g., it attributes amount of the work performed by `read` and `write` system calls to the `size` input parameter for each call. However, X-ray currently only supports a small number of system call parameters in this fashion. We hope to address these limitations more thoroughly, either by instrumenting the kernel or by creating more detailed performance models that observe system calls, parameters, and selected kernel events to infer such dependencies.

X-ray only considers configuration settings and program inputs as possible root causes. If the true root cause is a program bug or any other cause not considered by X-ray, X-ray cannot diagnose the problem. X-ray will produce an ordered list of possible causes, all of which will be incorrect. Thus, one potential improvement is to require a minimal level of confidence before X-ray adds a root cause to the ordered list—this may enable X-ray to better identify situations where the true root cause is not in its domain of observation.

While X-ray adds only a small overhead on the production system, its offline analysis runs 2–3 orders of magnitude slower than the original execution. Thus, while logging may be continuously enabled, we envision that only portions of the log will be analyzed offline.

## 7 Conclusion

Diagnosing performance problems is challenging. X-ray helps users and administrators by identifying the root cause of observed performance problems. Our results show that X-ray accurately identifies the root cause of many real-world performance problems, while imposing only 2.3% average overhead on a production system.

## Acknowledgments

We thank the anonymous reviewers and our shepherd, Dejan Kostić, for comments that improved this paper. This research was supported by NSF award CNS-1017148. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing NSF, Michigan, Google, or the U.S. government.

## References


[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP* (October 2003), pp. 74–89.

[2] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proc. SOSP* (October 2009), pp. 193–206.

[3] Apache HTTP server version 2.4 documentation: Authentication, authorization, and access control. `http://httpd.apache.org/docs/2.2/howto/autho.html`.

[4] Apache HTTP server version 2.4 documentation: Apache performance tuning. `http://httpd.apache.org/docs/current/misc/perf-tuning.html`.

[5] Apache performance tuning. `http://perlcode.org/tutorials/apache/tuning.html`.

[6] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. OSDI* (October 2010).

[7] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proc. OSDI* (December 2004), pp. 259–272.

[8] BHATIA, S., KUMAR, A., FIUCZYNSKI, M. E., AND PETERSON, L. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proc. OSDI* (December 2008), pp. 103–116.

[9] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM TOCS 14*, 1 (February 1996), 80–107.

[10] BROWN, A. B., AND PATTERSON, D. A. To err is human. In *DSN Workshop on Evaluating and Architecting System Dependability* (July 2001).

[11] BROWN, A. B., AND PATTERSON, D. A. Undo for operators: Building an undoable e-mail store. In *Proc. USENIX ATC* (June 2003).

[12] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI* (December 2008), pp. 209–224.

[13] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proc. USENIX ATC* (June 2004), pp. 15–28.

[14] CHEN, H., JIANG, G., ZHANG, H., AND YOSHIHIRA, K. Boosting the performance of computing systems through adaptive configuration tuning. In *Proc. SAC* (March 2009), pp. 1045–1049.

[15] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-based failure and evolution management. In *Proc. NSDI* (March 2004).

[16] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic Internet services. In *Proc. DSN* (June 2002), pp. 595–604.

[17] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in vivo multi-path analysis of software systems. In *Proc. ASPLOS* (March 2011).

[18] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC* (June 2008), pp. 1–14.

[19] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. OSDI* (December 2004), pp. 231–244.

[20] DIAO, Y., HELLERSTEIN, J. L., PAREKH, S., AND BIGUS, J. P. Managing Web Server Performance with AutoTune Agent. *IBM Systems Journal 42*, 1 (January 2003), 136–149.

[21] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI* (December 2002), pp. 211–224.

[22] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proc. NSDI* (April 2007), pp. 271–284.

[23] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *Proc. USENIX ATC* (June 2006).

[24] GRAY, J. Why do computers stop and what can be done about it? In *Proc. Symp. Rel. Dist. Sofrware and DB Syst.* (1986).

[25] JUNQUEIRA, F., SONG, Y. J., AND REED, B. BFT for the skeptics. In *Proc. SOSP: WIP Session* (October 2009).

[26] KASICK, M. P., TAN, J., GANDHI, R., AND NARASIMHAN, P. Black-box problem diagnosis in parallel file systems. In *Proc. FAST* (February 2010).

[27] LAURIE, B., AND LAURIE, P. *Apache: The Definitive Guide, 3rd Edition*. O'Reilly Media, Inc., December 2002.

[28] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI* (June 2005), pp. 190–200.

[29] http://msdn.microsoft.com/en-us/library/bb968803(v=VS.85).aspx.

[30] MURPHY, B., AND GENT, T. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International 11*, 5 (1995).

[31] MYERS, E. W. An O(ND) difference algorithm and its variations. *Algorithmica 1*, 1–4 (1986), 251–266.

[32] NAGARAJA, K., OLIVERIA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. Understanding and dealing with operator mistakes in Internet services. In *Proc. OSDI* (December 2004), pp. 61–76.

[33] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proc. NDSS* (February 2005).

[34] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *Proc. USITS* (March 2003).

[35] http://oprofile.sourceforge.net/.

[36] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP* (October 2009), pp. 177–191.

[37] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proc. CGO* (March 2010).

[38] http://www.karoltomala.com/blog/?p=576.

[39] Postfix stress-dependent configuration. `http://www.postfix.org/STRESS_README.html`.

[40] Postfix tuning guide. `http://www.postfix.org/TUNING_README.html`.

[41] POZNIANSKY, E., AND SCHUSTER, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. PPoPP* (June 2003), pp. 179–190.

[42] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium* (July 2005), pp. 49–64.

[43] QIAN, F., QUAH, K. S., HUANG, J., ERMAN, J., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. Web caching on smartphones: Ideal vs. reality. In *Proc. MobiSys* (June 2012).

[44] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: A fully integrated practical record/replay system. *ACM TOCS 17*, 2 (May 1999), 133–152.

[45] RUAN, Y., AND PAI, V. Making the "box" transparent: System call performance as a first-class result. In *Proc. USENIX ATC* (June 2004), pp. 1–14.

[46] SAMBASIVAN, R. R., ZHENG, A. X., ROSA, M. D., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *Proc. NSDI* (March 2011), pp. 43–56.

[47] SMITH, G. *PostgreSQL 9.0 High Performance*. October 2010.

[48] SMITH, G., TREAT, R., AND BROWNE, C. Tuning your postgresql server. `http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server`.

[49] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proc. USENIX ATC* (June 2004), pp. 29–44.

[50] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proc. SOSP* (October 2007), pp. 237–250.

[51] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proc. SOSP* (October 2011).

[52] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proc. ASPLOS* (March 2011).

[53] http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[54] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proc. OSDI* (December 2004), pp. 245–257.

[55] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proc. LISA* (October 2003), pp. 159–172.

[56] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proc. OSDI* (December 2004), pp. 77–90.

[57] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. MoBS* (June 2007).

[58] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L., AND PASUPATHY, S. An emprirical study on configuration errors in commerica1 and open source systems. In *Proc. SOSP* (October 2011).

[59] YU, M., GREENBERG, A., MALTZ, D., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *Proc. NSDI* (March 2011), pp. 57–70.

[60] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. USENIX ATC* (June 2009).

[61] ZHENG, W., BIANCHINI, R., AND NGUYEN, T. D. Automatic configuration of Internet services. In *Proc. EuroSys* (March 2007), pp. 219–229.

# Pasture: Secure Offline Data Access Using Commodity Trusted Hardware

Ramakrishna Kotla, Microsoft Research, Silicon Valley
Tom Rodeheffer, Microsoft Research, Silicon Valley
Indrajit Roy*, HP Labs, Palo Alto
Patrick Stuedi*, IBM Research, Zurich
Benjamin Wester*, Facebook, Menlo Park

## Abstract

This paper presents Pasture, a secure messaging and logging library that enables rich mobile experiences by providing secure offline data access. Without trusting users, applications, operating systems, or hypervisors, Pasture leverages commodity trusted hardware to provide two important safety properties: *access-undeniability* (a user cannot deny any offline data access obtained by his device without failing an audit) and *verifiable-revocation* (a user who generates a verifiable proof of revocation of unaccessed data can never access that data in the future).

For practical viability, Pasture moves costly trusted hardware operations from common data access actions to uncommon recovery and checkpoint actions. We used Pasture to augment three applications with secure offline data access to provide high availability, rich functionality, and improved consistency. Our evaluation suggests that Pasture overheads are acceptable for these applications.

## 1 Introduction

Mobile experiences are enriched by applications that support offline data access. Decentralized databases [50], file systems [24], storage systems [31], and email applications [36] support disconnected operation to provide better mobility and availability. With the increasing use of mobile devices—such as laptops, tablets, and smart phones—it is important that a user has access to data despite being disconnected.

However, support for disconnected operation is at odds with security when the user is not trusted. A disconnected untrusted user (assumed to be in full control of the user device) could perform arbitrary actions on whatever data was available and subsequently lie about it. This tension between mobility and security limits the use of disconnected operation in many potentially useful scenarios, for example:

- *Offline video rental services:* Most web-based video rental services require users to be online to watch a streaming video. Some allow users to download movies and watch them offline but the movies must be purchased ahead of time, with no refund possible for unwatched movies. It would be useful if a user could rent some movies, download them when online (for example, in an airport), selectively watch some of them offline (on the plane), delete the unwatched movies, and then get a refund later (when back online after landing). Similar offline services could be provided for electronic books.
- *Offline logging and revocation:* Secure logging of offline accesses is required by law in some cases. For example, HIPAA [48] mandates that (offline) accesses to confidential patient information be logged securely to enable security audits so as to detect and report privacy violations due to unauthorized accesses. Furthermore, accidental disclosures [23] to unauthorized entities are not exempt from civil penalties under HIPAA. Hence, it is also important to enable verifiable revocation of (unread) data from an unintended receiver's device in order to mitigate liability arising out of accidental disclosures.

Support for secure offline data access raises two important security problems: How do you know that an untrusted user is not lying about what data was accessed while offline? And, if the user claims data has been deleted, how do you know that he did not secretly keep a copy for later access?

These problems cannot be solved simply by using encryption because the untrusted user must be able to get offline access to the decryption keys in order to read the data. Rather, it is an issue of (1) securely detecting and logging access to the decryption keys when data is accessed and (2) securely retracting unused keys to prevent future access when unaccessed data is deleted. These problems are hard because the untrusted user is disconnected when he makes offline accesses and he is in full physical control of his device.

Indeed, these problems limit application functionality and deployability. Consequently, as observed above, current online services provide restricted offline functionality that does not allow a refund for downloaded but (allegedly) unaccessed movies or books.

Fortunately, recent advances have resulted in widespread availability of commodity trusted hardware in the

*Work done while at Microsoft Research, Silicon Valley.

form of Trusted Platform Modules (TPMs) [51] and secure execution mode (SEM) extensions (Intel's TXT [22] and AMD's SVM [1] technology) in many modern day laptops, tablets, and PCs [17].

**Secure offline data access using trusted hardware.** We present *Pasture*, a secure messaging and logging library, that leverages commodity trusted hardware to overcome the above problems by providing following safety properties:

- *Access undeniability:* If a user obtains access to data received from a correct sender, the user cannot lie about it without failing an audit. (Destruction or loss of the user device automatically fails an audit.)
- *Verifiable revocation:* If a user revokes access to unaccessed data received from a correct sender and generates a verifiable proof of revocation, then the user did not and cannot ever access the data.

Pasture uses a simple yet powerful *bound key* TPM primitive to provide its safety properties. This primitive ensures access undeniability by releasing the data decryption key only after the access operation is logged in the TPM. It provides verifiable revocation by permanently destroying access to the decryption key and generating a verifiable proof of revocation if a delete operation is logged in the TPM instead.

**Making secure offline data access practical.** Flicker [34] and Memoir [37] have demonstrated the use of TPMs and SEM to run trusted application code on untrusted devices in isolation from the OS and hypervisor. However, using SEM requires disabling interrupts and suspending all but one core, which can result in poor user responsiveness, resource underutilization, and higher overhead. Furthermore, these systems are vulnerable to memory and bus snooping attacks [16, 21]. Pasture avoids these drawbacks by carefully using trusted hardware operations to ensure practicality without compromising on safety.

First, unlike Flicker and Memoir, Pasture does not use SEM for the common case data access operations, and thus provides better user interaction and improved concurrency by enabling interrupts and cores. Perhaps surprisingly, we found that Pasture could maintain its safety properties even though SEM is limited to the uncommon operations of recovery and checkpoint.

Second, similar to Memoir, we significantly reduce overhead and improve durability by limiting NVRAM and non-volatile monotonic counter update operations (which are as slow as 500 ms and have limited lifetime write cycles [37]) to uncommon routines and not using them during the regular data access operations.

**Contributions.** We make two key contributions in this paper. First, we present the design and implementation of Pasture, providing secure and practical offline data access using commodity trusted hardware. We wrote a formal specification [40] of Pasture and its safety proofs in TLA+, checked the specification with the TLC model checker [27], and mechanically verified the proofs using the TLA+ proof system. Our evaluation of a Pasture prototype shows that common case Pasture operations such as offline data access takes about 470 ms (mainly due to the TPM decryption overhead) while offline revocation can be as fast as 20 ms.

Second, we demonstrate the benefits of Pasture by (a) providing rich offline experiences in video/book rental services, (b) providing secure logging and revocation in healthcare applications to better handle offline accesses to sensitive patient health information, and (c) improving consistency in decentralized data sharing applications [2, 31, 50] by preventing read-denial attacks.

## 2 Overview and Approach

The goal of Pasture is to improve mobility, availability, and functionality in applications and services by allowing untrusted users to download data when online and make secure offline accesses to data. In this section, we state our requirements, review features of commodity trusted hardware, define our adversary, and present our approach.

### 2.1 Requirements

*(1) Access undeniability.* The access-undeniability property prevents an untrusted node[1] from lying about offline accesses to data sent by *correct nodes*. It is beyond the scope of this paper to address a more general problem of detecting or preventing data exchanges between colluding, malicious nodes. (For example, a malicious user could leak data to another malicious user during an unmonitored phone conversation.)

*(2) Verifiable revocation.* While access undeniability prevents users from lying about past data accesses, the verifiable-revocation property allows a user to permanently revoke access to unaccessed data. Furthermore, the user can supply a proof that its access was indeed securely revoked. The user will not be able to decrypt and read the data at a later time even if he keeps a secret copy of the encrypted data.

*(3) Minimal trusted computing base (TCB).* To defend against an adversary who has full physical access to the device when offline, we want to have a small TCB. Hence, we do not trust the hypervisor, OS, or the application to provide Pasture's safety properties. We do not want to trust the bus or memory to not leak any information as hardware snooping attacks [16, 21] are possible in our setting. However, we have to trust something, since it is impossible to track a user's offline actions without trusting anything on the receiver device.

[1] We use the terms "node" and "user" interchangeably to refer to an untrusted entity that takes higher level actions on data.

*(4) Low cost.* We want to keep the costs low by using only commodity hardware components.

## 2.2 Commodity trusted hardware

Pasture exploits commodity trusted hardware to meet its requirements. Here we explain some of the key features we use and refer readers to the textbook [5] for details. TPMs [51] are commodity cryptographic coprocessors already shipped in more than 200 million PCs, tablets and laptops [17]. TPMs are designed to be tamper-resistant and cannot be compromised without employing sophisticated and costly hardware attacks. TPMs are currently used for secure disk encryption [4] and secure boot [8] in commodity OS.

*Keys and security.* Each TPM has a unique identity and a certificate from the manufacturer (Infineon for example) that binds the identity of the TPM to the public part of a unique endorsement key (EK). The private part of EK is securely stored within the TPM and never released outside. For privacy, TPMs use internally generated Attestation Identity Keys (AIKs) for attestations and AIKs are certified by trusted privacy CAs after confirming that they are generated by valid TPMs with proper EK certificates. It is cryptographically impossible to spoof hardware TPM attestations and emulate it in software. TPMs can securely generate and store other keys, make attestations of internal state, sign messages, and decrypt and encrypt data.

*Registers and remote attestation.* TPMs have volatile registers called Platform Configuration Registers (PCRs) which can be updated only via the TPM_Extend operation, which concatenates a value to the current value, and then overwrites the PCR with a SHA1 hash of the concatenated value. Intuitively, a PCR serves as a chained SHA1 hash of events in a log. TPMs also support remote attestation using a TPM_Quote operation which returns the signed contents of one or more PCRs along with a specified nonce. TPMs (v1.2) typically have 24 PCRs and we refer interested readers to other sources [5, 51] for details.

*Bound keys.* TPMs provide a TPM_CreateWrapkey operation that creates a bound public-private key pair. The encryption key can be used anywhere but the corresponding decryption key is shielded and can be used only in the TPM when specified PCRs contain specified values.

*Transport sessions.* TPMs support a form of attested execution using TPM transport sessions. Operations within a transport session are logged and signed so that a verifier can securely verify that a given TPM executed a specified sequence of TPM operations with specified inputs and outputs.

*Non-volatile memory and counters.* TPMs provide a limited amount of non-volatile memory (NVRAM) which can be used to persist state across reboots. A region of NVRAM can be allocated and protected so that it can be accessed only when specified PCRs contain specified values. TPMs also provide non-volatile monotonic counters which can be updated only by an increment operation (TPM_IncrementCounter).

*Secure Execution Mode (SEM).* AMD and Intel processors provide special processor instructions to securely launch and run trusted code in isolation from DMA devices, the hypervisor and the OS. Roughly speaking, the processor sets DMA memory protection so that DMA devices cannot access the trusted code, disables interrupts and other cores to prevent other software from taking control or accessing the trusted code, resets and extends a special $\mathrm{PCR}_{\mathrm{SEM}}$ register with the SHA1 hash of the trusted code, and then starts executing the trusted code.

The AMD and Intel details are different, but in each case the reset of $\mathrm{PCR}_{\mathrm{SEM}}$ produces a value that is not the same as the initial value produced by a reboot, and hence the result obtained by extending $\mathrm{PCR}_{\mathrm{SEM}}$ by the SHA1 hash of the trusted code is cryptographically impossible to produce in any other way. This PCR value can be specified to restrict access to secrets (that are encrypted by bound keys) or NVRAM locations to the trusted code only.

When the trusted application finishes its execution, the SEM exit code extends $\mathrm{PCR}_{\mathrm{SEM}}$ to yield access privileges, scrubs memory of any secrets it wants to hide, then reenables DMA, interrupts, and other cores. Flicker [34] demonstrated how to use SEM to execute arbitrary trusted code.

## 2.3 Adversary model

The adversary's goal is to violate safety. We do not consider violations of liveness, since the adversary could always conduct a denial-of-service attack by simply destroying the device. Violating safety means to violate either access undeniability or verifiable revocation. The adversary wins if he can obtain access to data from a correct sender and then subsequently survive an audit while claiming that access was never obtained. The adversary also wins if he can produce a valid, verifiable proof of revocation for data from a correct sender and yet at some time, either before or after producing the proof, obtain access to the data.

The adversary can run arbitrary code in the OS, hypervisor, or application. The adversary controls power to the device and consequently can cause reboots at arbitrary points in time, even when the processor is executing in secure execution mode. As opposed to Memoir [37], we assume that the adversary can perform hardware snooping attacks on the main memory [16] or CPU-memory bus [21] to deduce the contents of main memory at any time. We also assume that the adversary can snoop on the CPU-TPM bus.

However, we assume that the adversary cannot extract

secrets from or violate the tamper-resistant TPM and compromise the processor's SEM functionality. We also assume that the adversary cannot break cryptographic secrets or find collisions or preimages of cryptographic hashes. We assume the correctness of processor's CPU and the trusted Pasture code that runs in SEM.

In Pasture, we ensure that a sender's data is accessed securely on a receiver node, and it is not our goal to keep track of who accessed the data or for what purpose. Other techniques [33, 34, 55] can be used to provide isolation across various entities on the receiver node based on the receiver's trust assumptions. Pasture's safety guarantees are for the sender and are independent of the receiver's trust assumptions.

Since an audit is possible only when the receiver is online, the adversary can prevent an audit by disconnecting or destroying the device. We let applications decide how to deal with long disconnections and device failures. In the offline video rental scenario, for example, the fact that a node might have been disconnected for a long time is irrelevant, because the user has already paid for the movies and can get refunds only by providing verifiable proofs of revocation.

To be conservative, an application cannot assume that data has been destroyed in a device failure or lost during a long network disconnection. In such situations it is impossible to tell comprehensively what data has been accessed. In spite of this, we guarantee that if there is a valid, verifiable proof of revocation of data received from a correct sender, then the adversary can never access that data.

### 2.4 Pasture's approach

We carefully use TPM and SEM operations to provide secure offline data access to meet Pasture's requirements.

*(1) Summary log state in a PCR.* Pasture uses a Platform Configuration Register $\mathrm{PCR}_{\mathrm{APP}}$ to capture a cryptographic summary of all decisions to obtain access or revoke access to data. Since a PCR can be updated only via the TPM_Extend operation, it is cryptographically impossible to produce a specified value in a PCR in any other way than by a specified sequence of extensions.

*(2) Minimal use of SEM and NV operations.* By carefully mapping the process of obtaining and revoking access onto TPM primitives, Pasture avoids using SEM or NV updates during normal operation and limits their use to bootup and shutdown.

*(3) Prevent rollback and hardware snooping attacks.* Since Pasture state is stored in a volatile PCR, an adversary could launch a rollback attack [37] to attempt to retract past offline actions. The rollback attack would proceed by rebooting the system, which resets $\mathrm{PCR}_{\mathrm{APP}}$ to its initial value, followed by re-extending $\mathrm{PCR}_{\mathrm{APP}}$ to a valid, but older state. The adversary would truncate the full (untrusted) log to match the state in $\mathrm{PCR}_{\mathrm{APP}}$.

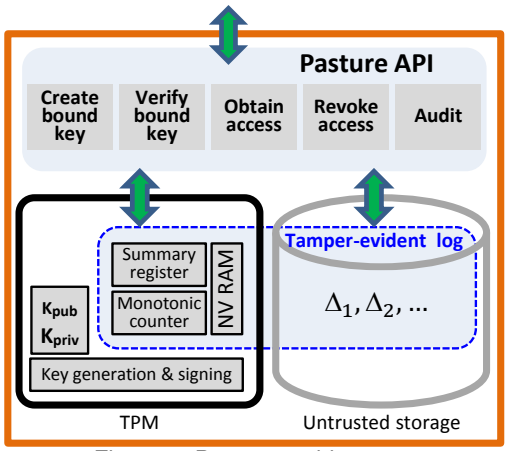


Figure 1: Pasture architecture.

Memoir prevents this attack by incorporating a secret into each PCR extension. The adversary cannot perform re-extensions without knowing the secret, which is shielded so that it is accessible only to the Memoir trusted code running in SEM. This is a reasonable approach in Memoir, which handles general applications and runs every operation in SEM. But since we grant the adversary the power to snoop on hardware busses, this defense is insufficient for Pasture.

Pasture prevents this attack by exploiting the fact that SEM is not needed for normal operations. Instead of making the required contents of $\mathrm{PCR}_{\mathrm{APP}}$ impossible for the adversary to reproduce, Pasture conjoins a requirement that $\mathrm{PCR}_{\mathrm{SEM}}$ contain a value that is cryptographically impossible to produce except by executing Pasture's trusted reboot recovery routine and verifying that $\mathrm{PCR}_{\mathrm{APP}}$ has been restored to the latest value.

Pasture uses Memoir's approach of saving the latest value of $\mathrm{PCR}_{\mathrm{APP}}$ in protected NVRAM on shutdown so that its correct restoration can be verified on the subsequent reboot.

## 3 Pasture Design

Figure 1 shows the high-level architecture of Pasture. Each node runs a Pasture instance which is uniquely identified by a public/private key pair securely generated and certified (using AIK) by its corresponding TPM. All proofs and messages generated by a Pasture instance are signed using the private part. Receivers verify signatures using the public part. Since the private part of the key is protected inside the TPM, it is impossible for an adversary to spoof Pasture's proofs or messages.

Each Pasture instance maintains a tamper-evident append-only log of decisions $\Delta_1, \Delta_2, \ldots$ about offline decisions to access or revoke a key. The full log is kept in the node's untrusted storage and a cryptographic summary of the log is maintained inside the TPM on a PCR. The application running on a Pasture instance uses the Pasture API to *Create* and *Verify* bound encryption keys

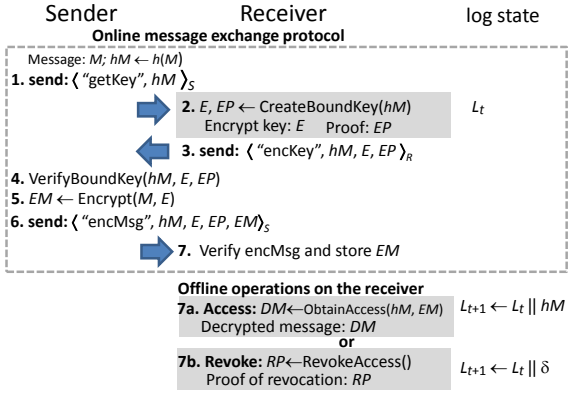



Figure 2: Pasture data transfer protocol, offline operations, and log state. Shaded regions represent TPM operations. Protocol messages are signed by their corresponding message sender (S or R).

during the data transfer protocol, to *Obtain* and *Revoke* access to the corresponding decryption keys based on offline decisions, and to respond to an *Audit*.

Many of the following subsections contain **implementation details**. Skipping these details on the first reading may help the reader.

### 3.1 Data transfer protocol

Figure 2 shows the secure data transfer protocol used by a sender to transfer encrypted data to a receiver. When a sender wants to send data $M$ to a receiver, the sender gets an (asymmetric) encryption key $E$ generated by the receiver's TPM and sends the encrypted data *EM*. The receiver then can make offline decision to access the data $M$ by decrypting *EM* or revoke access to $M$ by permanently deleting access to the decryption key in its TPM and generating a proof of revocation.

We describe the protocol with a single sender below and defer discussion of concurrent data transfers from multiple senders to §3.6. At the beginning of the protocol, the receiver's log state is $L_t$. We use the subscript $t$ to indicate the state before creating entry $t+1$ in the log. The subscript $t+1$ indicates the next state.

In step 1, the sender provides the cryptographic hash $hM = h(M)$ when requesting an encryption key.

In step 2, the receiver generates a key pair bound to a hypothetical future log state $L_t||hM$, in which the current log $L_t$ is extended by a decision to obtain access to the bound decryption key. The cryptographic hash *hM* represents this decision. The TPM only permits decryption using the bound key when the log has the hypothesized state. Pasture creates a proof *EP* that the bound key was generated by the receiver's TPM in the proper manner using a transport session.

In step 3 the proof *EP* and the encryption key $E$ are returned to the sender in the "encKey" message.

In step 4, The sender checks the proof to verify that the receiver acted properly. If the "encKey" message is

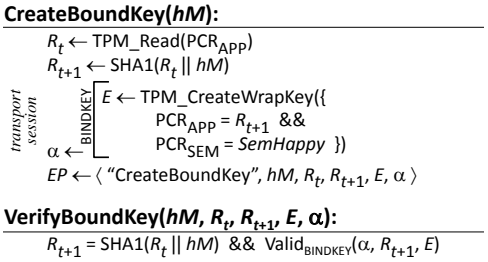



Figure 3: Create and verify bound key operations.

malformed or if the proof is not valid then the sender aborts the protocol and does not interact further with the faulty receiver. In this way a correct sender can protect itself against a malicious receiver.

In step 5, the sender encrypts its message using $E$. In step 6, the encrypted data *EM* is sent to the receiver, along with the original hash *hM*, the encryption key $E$, and the receiver's proof *EP*.

In step 7, after receiving "encMsg" and verifying that it is properly signed by the sender, the receiver checks that *hM*, *E*, and *EP* match with the corresponding values it sent in its "encKey" message. Then the receiver stores the "encMsg" with the encrypted data *EM* on its local storage. The receiver can later make an offline decision to obtain access to the bound decryption key (and thus obtain access to the sender's data) or to revoke access.

In case of communication failures, neither the sender nor the receiver need to block because they can always discard their current state and restart.

**Implementation details.** For simplicity, we described the protocol as if $M$ is the actual data $X$ but we use the standard practice [35] of allowing the sender to choose a nonce symmetric key $K_{Sym}$ to encrypt $X$ and then encrypting $M = \langle K_{Sym}, h(X) \rangle$ using the asymmetric key acquired in this protocol. Securely obtaining or revoking access to $M$ is equivalent to obtaining or revoking access to the actual data $X$. Note that $K_{Sym}$ also acts as a unique and random nonce to prevent a faulty receiver from inferring $M$ from the hash $hM$ without decrypting the message even if the receiver has seen data $X$ in prior exchanges with this or other senders.

Figure 3 shows the implementation of the `CreateBoundKey` and `VerifyBoundKey` operations. Pasture exploits the TPM primitive TPM_CreateWrapKey, which creates a key pair in which the decryption key is usable only when specified PCRs contain specified values. Pasture keeps a cryptographic summary of the log in a PCR called $\mathrm{PCR}_{\mathrm{APP}}$.

`CreateBoundKey` reads the current summary value $R_t$ from $\mathrm{PCR}_{\mathrm{APP}}$, computes what the new summary value $R_{t+1}$ would be if *hM* were appended to the log, and then invokes TPM_CreateWrapKey to create a key pair with the decryption key usable only when

**ObtainAccess(*hM*, *EM*):**

append *hM* to full log
TPM_Extend($\text{PCR}_{\text{APP}}$, *hM*)
*DM* ← TPM_Unbind(*EM*)

**RevokeAccess():**

$R_t$ ← TPM_Read($\text{PCR}_{\text{APP}}$)
append $\delta$ to full log
TPM_Extend($\text{PCR}_{\text{APP}}$, $\delta$)
$R'_{t+1}$, $S'_{t+1}$, $\alpha$ ← TPM_Quote($\text{PCR}_{\text{APP}}$, $\text{PCR}_{\text{SEM}}$)
*RP* ← ⟨ "RevokeAccess", $\delta$, $R_t$, $R'_{t+1}$, $S'_{t+1}$, $\alpha$ ⟩

Figure 4: Obtain and revoke access operations.

$\text{PCR}_{\text{APP}}$ contains $R_{t+1}$. (The additional constraint that $\text{PCR}_{\text{SEM}}$ contains *SemHappy* is discussed in §3.4.) TPM_CreateWrapKey is invoked inside a TPM transport session, which provides an attestation $\alpha$ signed by the TPM that the BINDKEY sequence of TPM operations were performed with the indicated inputs and outputs.

The sender uses `VerifyBoundKey` to check the contents of the proof *EP*. Note that there is no need to verify that $R_t$ is a correct cryptographic summary of the receiver's log. The attestation $\alpha$ proves that $E$ is bound to $R_{t+1}$, which is the correct extension from $R_t$ by *hM*. It is cryptographically impossible to find any other value from which an extension by *hM* would produce $R_{t+1}$.

## 3.2 Secure offline access and revocation

To obtain access, as shown in step 7a of Figure 2, the receiver irrevocably extends its log by *hM*. The TPM now permits decryption using the bound key, and the decrypted message *DM* can be obtained. A faulty sender could play a bait-and-switch trick by sending an encrypted text for a different data than initially referenced in its "getKey" message, but the receiver can catch this by noticing that $hM \neq h(DM)$. If the sender is faulty, the receiver can form a proof of misbehavior by exhibiting *DM* and the sender's signed "encMsg" message, which includes *hM*, $E$, *EP*, and *EM*. Any correct verifier first verifies that the receiver is not faulty by checking that $Encrypt(DM, E) = EM$. Then the verifier checks that $hM \neq h(DM)$, which proves that the sender is faulty. In this way a correct receiver can protect itself against a malicious sender.

To revoke access, in step 7b, the receiver irrevocably extends its log by $\delta$, a value different from *hM*. This makes it cryptographically impossible ever to attain the state in which the TPM would permit decryption using the bound key. In effect, the bound decryption key has been revoked and the receiver will never be able to decrypt *EM*. Pasture constructs a proof *RP* of this revocation, which any correct verifier can verify.

**Implementation details.** Figure 4 shows the implementation of the `ObtainAccess` and `RevokeAccess` operations. Step 7a calls `ObtainAccess` to obtain offline data access. `ObtainAccess` appends *hM* to the full log on untrusted storage and extends the cryptographic

**Audit(*nonce*):**

$R_t$, $S_t$, $\alpha$ ← TPM_Quote($\text{PCR}_{\text{APP}}$, $\text{PCR}_{\text{SEM}}$, *nonce*)
*AP* ← ⟨ "Audit", full log, $R_t$, $S_t$, *nonce*, $\alpha$ ⟩

Figure 5: Audit operation.

summary maintained in $\text{PCR}_{\text{APP}}$. Since $\text{PCR}_{\text{APP}}$ now contains the required summary value, the TPM permits use of the decryption key to decrypt the data, which is performed via the TPM_Unbind primitive.

Step 7b calls `RevokeAccess` to revoke data access. `RevokeAccess` appends $\delta$ to the log and extends the cryptographic summary accordingly. Since $\delta \neq hM$, this produces a summary value $R'_{t+1} \neq R_{t+1}$. Since it is cryptographically impossible to determine any way of reaching $R_{t+1}$ except extending from $R_t$ by *hM*, this renders the decryption key permanently inaccessible. Pasture uses TPM_Quote to produce an attestation $\alpha$ that $\text{PCR}_{\text{APP}}$ contains $R'_{t+1}$. (The simultaneous attestation that $\text{PCR}_{\text{SEM}}$ contains $S'_{t+1}$ is discussed in §3.4.) The exhibit of the prior summary value $R_t$ along with $R'_{t+1}$ and the attestation $\alpha$ proves that $R_{t+1}$ is unreachable.

There are several ways in which the code in `RevokeAccess` can be optimized. First, Pasture can skip the TPM_Read operation by tracking updates to $\text{PCR}_{\text{APP}}$ with the CPU. Second, if the proof of revocation is not needed immediately, Pasture can delay executing the TPM_Quote until some later time, possibly coalescing it with a subsequent TPM_Quote. A multi-step sequence of extensions from $R_t$ to the current attested value of $\text{PCR}_{\text{APP}}$, in which the first step differs from the bound key value $R_{t+1}$, is cryptographically just as valid as a proof of revocation as a single-step sequence. Coalescing is a good idea, since TPM_Quote involves an attestation and hence is fairly slow as TPM operations go.

## 3.3 Audit

A verifier can audit a receiver to determine what decisions the receiver has made. The receiver produces a copy of its full log along with a proof signed by the TPM that the copy is current and correct. Hence, a faulty receiver cannot lie about its past offline actions.

**Implementation details.** Figure 5 shows the implementation of the `Audit` operation, which computes the response *AP* to an audit request. *AP* contains a copy of the entire log along with an attestation $\alpha$ of the current summary value $R_t$ contained in $\text{PCR}_{\text{APP}}$. The attestation also includes the nonce sent by the auditor to guarantee freshness. (The simultaneous attestation that $\text{PCR}_{\text{SEM}}$ contains $S_t$ is discussed in §3.4.) Any correct verifier can check the attestation and check that $R_t$ is the correct cryptographic summary value for the purported log contents, and thereby determine exactly what decisions $\Delta_1, \Delta_2, \ldots$ the audited node has made.

### 3.4 Dealing with reboots

The main difficulty faced by Pasture is dealing with reboots. Since decryption keys are bound to the log summary value contained in $\mathrm{PCR}_{\mathrm{APP}}$, anything that the adversary can do to break Pasture's guarantees of access undeniability and verifiable revocation must involve rebooting the node. Rebooting the node causes the TPM to reset PCRs to their initial values, which opens the door to rollback attacks.

Pasture's solution is inspired by Memoir-Opt [37] and in §2.4 we outlined the novel aspects of our approach. Since $\mathrm{PCR}_{\mathrm{APP}}$ is volatile, an adversarial reboot will cause its contents to be lost. So, like Memoir-Opt, Pasture uses a protected module containing a checkpoint routine that runs in SEM and saves the latest contents of $\mathrm{PCR}_{\mathrm{APP}}$ in a region of TPM NVRAM accessible only to the protected module. The checkpoint routine is installed to run during system shutdown and as part of a UPS or battery interrupt handler [37]. Note that our system does not assume correct operation of such mechanisms for safety.

As a node cycles through shutdown, reboot, recovery, and operation, it is important to keep track of where the current log summary value is located. During operation, it lives in $\mathrm{PCR}_{\mathrm{APP}}$. Shutdown moves it from $\mathrm{PCR}_{\mathrm{APP}}$ to Pasture's protected NVRAM region. Reboot and recovery moves it back into $\mathrm{PCR}_{\mathrm{APP}}$. To keep track of this, Pasture's protected NVRAM region contains two fields: $R$ and *current*. $R$ is used to hold a log summary value and *current* is a boolean flag indicating that the value in $R$ is indeed current.

The checkpoint routine sets *current* to TRUE after it has copied $\mathrm{PCR}_{\mathrm{APP}}$ into $R$. On reboot, Pasture recovers by reading the full log $\Delta_1, \Delta_2, \ldots$ and extending $\mathrm{PCR}_{\mathrm{APP}}$ by each entry in turn. Then Pasture uses SEM to enter its protected module and check that the value recorded in the NVRAM is indeed current and that it matches the value contained in $\mathrm{PCR}_{\mathrm{APP}}$. If so, the recovery routine sets *current* to FALSE, indicating that the current log summary value now lives in $\mathrm{PCR}_{\mathrm{APP}}$.

Observe that failure to run the checkpoint routine before a reboot will erase $\mathrm{PCR}_{\mathrm{APP}}$ but leave *current* $=$ FALSE, a state from which recovery is impossible. This is a violation of liveness, but not of safety.

**Implementation details.** Figure 6 shows the implementation of the `Recover` and `Checkpoint` operations. `Recover` extends $\mathrm{PCR}_{\mathrm{APP}}$ with each entry $\Delta$ on the full log and then enters SEM to check that the saved copy in NVRAM is current and matches the value in $\mathrm{PCR}_{\mathrm{APP}}$. If a shutdown happens precisely at the wrong time in the middle of `ObtainAccess` or `RevokeAccess`, it is possible for the full log to contain one final entry not represented in the saved log summary value. In this case, that final decision was not committed and the implementation

**Recover():**

```
FOR EACH entry Δ on full log: TPM_Extend(PCR_APP, Δ)
secure execution mode:
  IF nv.current && nv.R = TPM_Read(PCR_APP)
  THEN
      nv.current ← FALSE
      TPM_Extend(PCR_SEM, Happy)
  ELSE
      TPM_Extend(PCR_SEM, Unhappy)
```

**Checkpoint():**

```
transport session:
  SEAL:
    R_t ← TPM_Read(PCR_APP)
    S_t ← TPM_Read(PCR_SEM)
    C_t ← TPM_ReadCounter(CTR)
  α ← TPM_Extend(PCR_SEM, Unhappy)
secure execution mode:
  nv.R ← R_t
  IF Valid_SEAL(α, R_t, S_t, C_t) && S_t = SemHappy
      && C_t = TPM_ReadCounter(CTR)
  THEN
      TPM_IncrementCounter(CTR)
      nv.current ← TRUE
  TPM_Extend(PCR_SEM, Unhappy)
```

Figure 6: Recover and checkpoint operations.

described here will fail to recover. However, it is a simple matter to add a SEM routine that merely reads nv.$R$ so that `Recover` can tell whether or not it should remove the final entry from the full log. If recovery is successful, *current* is set to FALSE, indicating that the current log summary value now lives in $\mathrm{PCR}_{\mathrm{APP}}$.

However, there is an additional important detail: how do normal operations know that the current log summary lives in $\mathrm{PCR}_{\mathrm{APP}}$. These operations do not use SEM, so they cannot access *current*. Moreover, checking *current* is not enough, because they need to know that $\mathrm{PCR}_{\mathrm{APP}}$ was correctly restored on the *most recent* reboot. An adversary could mount a rollback attack by crashing, rebooting and partially re-extending $\mathrm{PCR}_{\mathrm{APP}}$, which would leave *current* as FALSE from the prior reboot.

Pasture exploits secure execution mode to prevent this attack. When the CPU enters secure execution mode, the TPM resets $\mathrm{PCR}_{\mathrm{SEM}}$ to -1 (different from its reboot reset value of 0) and then extends $\mathrm{PCR}_{\mathrm{SEM}}$ by the cryptographic hash of the protected module. For Pasture's protected module, this produces a result *SemProtected* in $\mathrm{PCR}_{\mathrm{SEM}}$ that is cryptographically impossible to produce in any other way. The constraint that $\mathrm{PCR}_{\mathrm{SEM}}$ contains *SemProtected* is used to control access to Pasture's protected TPM NVRAM.The adversary cannot undetectably modify Pasture's NVRAM, because TPM_DefineSpace resets the protected NVRAM locations whenever their access control is changed.

When execution of Pasture's protected module is finished, it extends $\mathrm{PCR}_{\mathrm{SEM}}$ to disable access to the protected NVRAM. Pasture defines two constants that it may use for this extension. *Happy* is used during recovery after a reboot if $\mathrm{PCR}_{\mathrm{APP}}$ has been correctly restored. *Unhappy* is used in all other cases. We define

*SemHappy* = SHA1(*SemProtected*||*Happy*), which is the value that $\mathrm{PCR}_{\mathrm{SEM}}$ will contain if the recovery is correct. Since PCRs are volatile, any adversarial attempt to reboot the node and reinitialize $\mathrm{PCR}_{\mathrm{APP}}$ will also reinitialize $\mathrm{PCR}_{\mathrm{SEM}}$. Pasture maintains the invariant that whenever $\mathrm{PCR}_{\mathrm{SEM}}$ = *SemHappy*, $\mathrm{PCR}_{\mathrm{APP}}$ contains the correct cryptographic summary of the log and decisions to obtain access or revoke access can be made. (Of course, adversarial entries in the log are permitted, but they cannot be rolled back.)

Since `CreateBoundKey` binds the key to require both $\mathrm{PCR}_{\mathrm{APP}} = R_{t+1}$ and $\mathrm{PCR}_{\mathrm{SEM}}$ = *SemHappy*, the decryption key will only be useable if $\mathrm{PCR}_{\mathrm{APP}}$ was correctly restored on the most recent reboot. Since `RevokeAccess` and `Audit` include $\mathrm{PCR}_{\mathrm{SEM}}$ in their TPM_Quote, a verifier can verify that the quoted $\mathrm{PCR}_{\mathrm{APP}}$ could only have been extended from a value that was correctly restored on the most recent reboot.

`Checkpoint` uses SEM to save the current value of $\mathrm{PCR}_{\mathrm{APP}}$ in the NVRAM and set *current* to TRUE. However, there is a difficulty. Before `Checkpoint` saves the current value of $\mathrm{PCR}_{\mathrm{APP}}$, it needs to know that $\mathrm{PCR}_{\mathrm{APP}}$ was correctly restored on the most recent reboot; otherwise `Checkpoint` itself would be vulnerable to a rollback attack. `Checkpoint` has to perform its checking and saving activities in SEM, so that the adversary cannot tamper with them. But the way of knowing that $\mathrm{PCR}_{\mathrm{APP}}$ was correctly restored is to check $\mathrm{PCR}_{\mathrm{SEM}}$ = *SemHappy*, and this information is erased by entering SEM.

The solution is to get a simultaneous attestation $\alpha$ of $\mathrm{PCR}_{\mathrm{APP}}$ and $\mathrm{PCR}_{\mathrm{SEM}}$ before entering SEM. Then, once in SEM, $\alpha$ can be checked, which proves that $\mathrm{PCR}_{\mathrm{APP}}$ contained $R_t$ when $\mathrm{PCR}_{\mathrm{SEM}}$ contained *SemHappy*.

Unfortunately, there is another vulnerability: how do we know this is the *most recent* such $\alpha$, and not some earlier one from the adversary. In defense, `Checkpoint` uses a TPM counter CTR whose value is read into $\alpha$ and then incremented in SEM once $\alpha$ is accepted as valid and current. This prevents any earlier $\alpha$ from being accepted.

There is yet a final vulnerability: how do we know that the adversary did not do anything bad between the time $\alpha$ was made and SEM was entered. For example, the adversary could make $\alpha$, then extend $\mathrm{PCR}_{\mathrm{APP}}$ to obtain access to a decryption key, then crash and reboot, re-extend $\mathrm{PCR}_{\mathrm{APP}}$ back to where it was when $\alpha$ was made, and then finally enter SEM in `Checkpoint`. To prevent this, `Checkpoint` extends $\mathrm{PCR}_{\mathrm{SEM}}$ inside $\alpha$, which erases *SemHappy*, which makes $\mathrm{PCR}_{\mathrm{APP}}$ useless for any attempt to obtain or revoke access until the next successful recovery. The above steps are performed in a transport session in the SEAL subroutine as shown in Figure 6.

### 3.5 Log truncation

Pasture allows applications to truncate logs in order to reduce storage, auditing, and recovery overheads. Truncation proceeds in several steps. (1) The subject node requests a trusted auditor to perform an audit and sign a certificate attesting to the subject node's current cryptographic log summary. The auditor has to be available at this time but an application could arrange for any desired number of trusted auditors. For example, the video service provider could act as an auditor for the offline video application. (2) The certificate is returned to the subject node, which discards all entries in its full log, creates a single log entry containing the certificate, and then performs a version of `Checkpoint` that verifies the certificate is current and if so saves the cryptographic summary of the new log in the NVRAM. (3) Then the subject node reboots to reinitialize $\mathrm{PCR}_{\mathrm{APP}}$.

Observe that truncating the log implicitly revokes access to a decryption key whose decision was pending. However, since in such a case no explicit decision was made to revoke access, the normal way of obtaining a proof of revocation is not available. If such proofs are important to the application, it should make explicit revocation decisions before truncating the log.

### 3.6 Handling concurrent messages

The basic protocol (§3.1) constrains the receiver to handle one sender's message at a time. Here we describe how to support multiple outstanding messages.

First, it is easy to extend the design to employ multiple PCRs. Current TPMs support 13 unreserved PCRs that Pasture can use. The design is modified to track the usage of these PCRs, allocating a free one in `CreateBoundKey`, passing it through the protocol in the "encKey" and "encMsg" messages, and freeing it after the decision is made to obtain or revoke access. Logically, a separate log is used for each PCR. `Audit` is extended to copy all the logs and quote all of the PCRs.

Second, in situations where the number of PCRs is insufficient, there is nothing to prevent a receiver from performing steps 2-3 in parallel with any number of senders using the same PCR, creating keys bound to different values of the next state. Of course, with the same PCR, only one sender's message can be accessed at step 7a. The other messages effectively are revoked. The receiver can ask those senders to retry using a new bound key.

Given multiple concurrent "getKey" requests and only one available PCR, the receiver could form a speculative decision tree multiple steps into the future. For each message, the receiver would generate multiple keys, binding each key to a different sequence of possible prior decisions. Each sender encrypts its message (actually, just its message's symmetric key) with all the keys so that the receiver can decide offline about access to the messages in any order it desired. So that the receiver can prove that a

key was revoked because the receiver failed to follow the speculated sequence of decisions, each key's revocation proof $RP$ would have to show the entire sequence starting from the receiver's current log state. Of course, the number of keys per message explode with the number of pending decisions, so this approach would be viable for only a very small number of outstanding messages.

### 3.7 Correctness

Using the TLA+ language and proof system [6,27], we wrote a formal specification of Pasture and mechanically verified a proof of correctness [40]. The correctness of Pasture when there are no reboots is trivial as it follows from the properties of bound keys and PCRs. Preventing rollback attacks in the presence of reboots is critical. The following invariants ensure correctness:

- If $\mathrm{PCR}_{\mathrm{SEM}} =$ *SemHappy* then *current* = FALSE, $\mathrm{PCR}_{\mathrm{APP}}$ contains the current log summary value, and there exists no acceptable SEAL attestation.
- If *current* = TRUE then $\mathrm{PCR}_{\mathrm{SEM}} \neq$ *SemHappy*, the NVRAM contains the current log summary value, and there exists no acceptable SEAL attestation.
- There exists at most one acceptable SEAL attestation.

These invariants are maintained by the use of *current*, the way $\mathrm{PCR}_{\mathrm{SEM}}$ is extended, and the reboot counter CTR.

Some comments on our methodology may be illuminating. After formulating a proof sketch to assure ourselves that Pasture was correct, we wrote a 19-page TLA+ specification. Several CPU-months spent on a large many-core server model-checking various configurations of this specification found no errors. To increase our confidence that we would have found errors if there were any, we then intentionally introduced some bugs and the resulting safety violations were easily detected.

Armed with confidence in the correctness of the specification, we then spent about two man-weeks writing a 68-page formal proof. The proof followed the reasoning of our original proof sketch, although with excruciating attention to detail, containing 1505 proof obligations. The TLA+ proof system checks them in half an hour.

Subsequently, we made a slight optimization to Pasture's operations, arriving at the version of Pasture described in this paper. It took only a few hours to revise the initial formal specification and proof to account for the optimization. The hierarchical structure of TLA+ proofs was a great benefit here, because the proof system highlighted precisely those few proof obligations that had to be revised in order to make the proof go through.

## 4 Applications

We use Pasture to prototype three applications with secure offline data access to (a) improve experience in a video rental service by allowing offline access to downloaded movies while enabling refunds for unwatched movies, (b) provide better security and mobility in a healthcare setting by allowing secure logging of offline accesses as required by HIPAA regulations, and (c) improve consistency in a decentralized storage system [2, 31, 50] by preventing read-denial attacks.

### 4.1 Video rental and healthcare

For wider deployability, we extended an email client application with Pasture to provide secure offline data access, and used the secure email as a transport and user interface mechanism to implement offline video rental and healthcare mockup application prototypes on top.

We implemented a generic Pasture add-in for Microsoft's Outlook email client [36]. Our video rental and health care applications are built on top of this secure email prototype to transfer data opportunistically to receivers when they are online, allow offline access to data, and permit remote auditing. The add-in calls the Pasture API (§3) to interact with the TPM.

The Pasture add-in allows senders to selectively encrypt sensitive attachments using the Pasture protocol. Users can choose messages that need the enhanced security properties while not paying overhead for the others. The add-in internally uses email to exchange Pasture protocol messages and acquires the encryption key from the receiver before sending the encrypted message.

On receiving a Pasture encrypted message, the user is given the option to access the attachment or delete it without accessing it. The user can use context (for example, the movie title, cast and a short description) included in the email body to make the decision. We assume that the sender is correct and motivated to include correct context. This assumption is reasonable for video rental and healthcare service providers. We also assume that the emails are signed to prevent spoofing attacks.

The user's decision to access or delete the attachment is permanently logged by Pasture. The Pasture add-in also provides an audit and truncate interface that a trusted entity can use to audit and truncate user logs

The Pasture-enhanced video rental service works as follows. When the user surfs the video rental service and selects movies for offline watching, he receives emails with encrypted symmetric keys as attachments. The encrypted movies are downloaded via https, which avoids sending the entire movie as an attachment. The user can watch movies offline by decrypting the attachments to extract the keys and then decrypting the movies. The user can revoke any movies not accessed. When the user comes back online, the video rental service provider audits the Pasture log to determine which movies were revoked and refunds the user accordingly.

Regulations [18, 48] in the healthcare industry impose fines [20] on healthcare providers if they allow unauthorized access to sensitive patient health information and they mandate that all accesses be logged securely. We

used the Pasture email add-in to provide secure offline access to medical records, for example, when a nurse goes for a home visit.

Access undeniability ensures that offline accesses by the nurse are securely logged. Verifiable revocation allows the nurse to securely delete unaccessed records and prove it to the hospital. Verifiable revocation also helps hospitals in assessing and mitigating damages due to accidental disclosures [13, 23] by allowing them to retract emails after they are sent to unintended recipients by asking the receivers to delete the confidential email and send them back the proof of revocation.

### 4.2 Decentralized storage systems

Decentralized storage systems (such as Bayou [50] and Practi [2]) provide high availability by allowing nodes to perform disconnected update operations on local state when they are offline, send updates to other nodes opportunistically when there is connectivity, and read received updates from other nodes. Depot [31] builds upon Practi to provide a storage system that tolerates any number of malicious nodes at the cost of weakening the consistency guarantee. One attack Depot cannot prevent is a *read-denial* attack, in which a malicious node denies reading updates from correct nodes before making its own updates.

We used Pasture to build a decentralized storage system that prevents read-denial attacks, and thereby provides stronger consistency than Depot. Preventing read-denial is a simple consequence of access undeniability. In addition, where Depot detects *equivocation* [7, 28], in which a malicious node sends conflicting updates, Pasture prevents equivocation by attesting updates using the Pasture log (the same approach as A2M [7]). Formalizing our consistency guarantee is an avenue of future work.

## 5 Evaluation

We implemented Pasture in Windows 7 and evaluated the system on three different computers: an HP xw4600 3GHz Intel Core2 Duo with a Broadcom TPM, an HP Z400 2.67GHz Intel Quad Core with an Infineon TPM, and a Lenovo X300 1.2Ghz Intel Core2 Duo laptop with an Atmel TPM. We implemented everything except we were unable to port Flicker [34] and get SEM work on the HP machines because the HP BIOS disables the SENTER instruction, and we haven't completely ported Pasture to run on Atmel TPM although we ran some TPM microbenchmarks on it. Missing functionality of SEM does not affect our evaluation or conclusions because SEM is not needed for the common case data access operations. Furthermore, for the Pasture operations that would use SEM, as we show later, our measured overheads are already significantly greater than the cost of setting up the SEM environment [34, 37].

| System | Isolation from malicious OS | Crash resilience | Invulnerable to snoop attack | No disabling of cores, interrupts per operation | No NV update per operation | Protected operations |
|---|---|---|---|---|---|---|
| Flicker | ✓ | × | × | × | ✓ | general |
| Memoir | ✓ | ✓ | × | × | ✓ | general |
| TrInc | ✓ | ✓ | ✓ | ✓ | × | attest |
| Pasture | ✓ | ✓ | ✓ | ✓ | ✓ | access revoke attest |

Table 1: Comparison of trusted hardware based systems.

### 5.1 Qualitative analysis

Table 1 compares Pasture against some recent systems that use trusted hardware to protect the execution of application operations. Flicker [34] and Memoir [37] use SEM and TPMs to provide protected execution of trusted modules on untrusted devices. TrInc [28] uses trusted smart cards to securely attest messages and prevent equivocation attacks in distributed systems. Pasture provides secure offline data access using SEM and TPMs. (We discuss additional related work in §7.)

All of these systems provide isolation from a malicious OS, hypervisor, or other application code. Flicker provides a general abstraction but it is vulnerable to snoop attacks and does not provide crash resilience [37], and thus fails to ensure state continuity across crashes. Furthermore, its use of SEM disables cores and interrupts for every operation. Memoir suffers from the same drawbacks as Flicker except that it is resilient to crashes. TrInc is crash resilient but provides limited functionality of secure attestation, and it is less durable due to NV updates on attest operations. Pasture provides offline access, revocation, and attestation without needing SEM or NV writes with each operation while providing crash resilience and defending against snoop attacks.

**Minimal TCB.** Pasture trusts just the `Checkpoint` and `Recover` routines which run during bootup and shutdown, and it does not trust any software during the common case data access operations. Pasture achieves this by exploiting the TPM primitives.

### 5.2 Pasture microbenchmarks

#### 5.2.1 Computational overhead

Our first experiment measures the computational overheads imposed by TPM operations on Infineon, Broadcom and Atmel TPM chips. Figure 7(a) plots the execution time of register and NV operations; Figure 7(b) plots the execution time of TPM cryptographic operations to create and load a key, sign data, unbind a key, release a transport session, and quote PCR state; and Figure 7(c) plots the execution time of Pasture operations that build upon the TPM operations. We use 1024 bit RSA keys for our experiments. We make the following observations.

**Slow NV updates.** Incrementing an NV counter or writing NVRAM is far slower than reading or extending

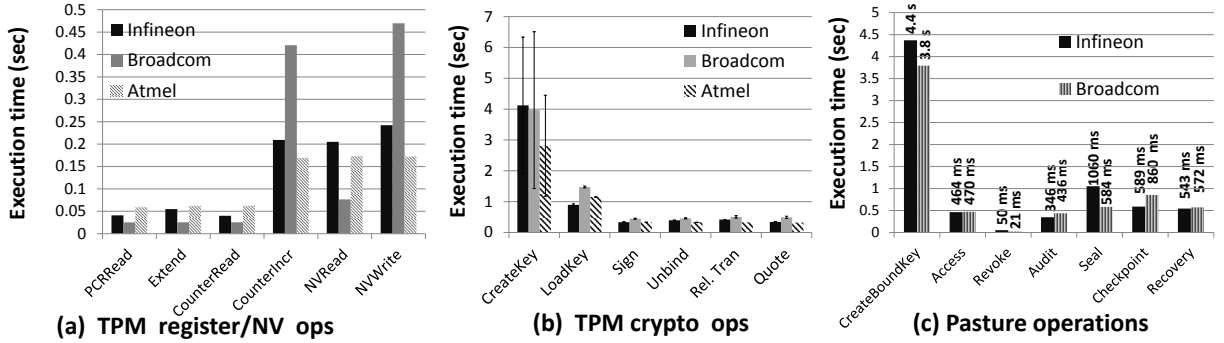



Figure 7: Computational overhead. Error bars represent one standard deviation of uncertainty over 10 samples.

a PCR. This supports our decision to avoid NV updates during common case operations.

**Data exchange protocol.** Creating a key in TPMs is enormously expensive, and it takes about 2.8 s (Atmel TPM) to 4 s (Broadcom TPM). This overhead accounts for almost all of Pasture's `CreateBoundKey` execution time. Key creation also has a huge variance. We study the impact of this overhead on perceived latencies to the end user in §5.3.1.

**Offline operations.** The offline operations to obtain or revoke access have acceptable performance. Revoking access is fast (21 ms) because it just requires a PCR extension (using the optimization discussed in §3.2 to defer the proof of revocation). Obtaining access, however, takes much longer (470 ms) as almost all of its time is spent in TPM_Unbind to decrypt and extract the symmetric key, which is fairly slow. We cannot hide this latency because the symmetric key is needed to decrypt the data. While the latency may be acceptable for an offline video or email application, our throughput is limited by these overheads. For some applications, batching can be used to amortize overheads across multiple data items as shown in §5.3.2.

Note that we hide the TPM overhead of LoadKey by loading it in the background while the data is being fetched from the sender and before the data is accessed offline.

**Checkpoint and recover operations.** (Note that our microbenchmarks do not include the 100 ms to 200 ms overhead required to set up the SEM environment [34].) `Checkpoint` takes about 1600 ms to seal and copy volatile state to the NVRAM, which is reasonable for a battery-backup shutdown routine. `Recover` takes about 600 ms to check the correctness of $\mathrm{PCR}_{\mathrm{APP}}$ when the system is booting up. Furthermore, before correctness is checked, `Recover` must read Pasture's log from the disk and extend $\mathrm{PCR}_{\mathrm{APP}}$ to its pre-shutdown state. Each TPM_Extend takes about 20 ms, so the additional overhead depends on the number of entries, which can be kept low by periodically truncating the log. If the log is truncated every 128 entries, at most 3 s would be spent extending $\mathrm{PCR}_{\mathrm{APP}}$. While the total overhead may seem high, modern operating systems already take tens to hundreds of seconds to boot up.

**Audit.** Pasture takes about 400 ms to generate a proof of revocation or a response to an audit request. Given that this operation is usually performed in the background, it does not much affect the user experience.

#### 5.2.2 Network and storage overheads

Pasture incurs network and storage overhead due to (1) additional messages exchanged for fetching keys and (2) inclusion of an attested proof of execution in the protocol messages and the log. With 1024 bit encryption keys, Pasture exchanges about 1732 bytes of additional data in the data transfer protocol, and stores less than 540 bytes of data on disk for logging each operation.

Pasture's proofs contain hashes of messages instead of raw messages. Hence, network and storage overheads do not increase with data size. Pasture imposes considerable overhead when the message sizes are smaller but the overhead becomes insignificant for large messages.

### 5.3 Pasture applications

Pasture uses various optimizations—that hide latency and batch operations—to reduce overheads. Here we evaluate the end-to-end performance of the latency-oriented (offline video and healthcare) and throughput-oriented (shared folder application) applications.

#### 5.3.1 Secure email transport

Our offline video rental and health care applications use Pasture-based secure email to allow secure offline accesses. For evaluating the overheads added by the Pasture system, we compare our applications with that of the corresponding applications that use regular email transport mechanism to send data (patient health records or the decryption key of the movie) but without secure offline data access guarantees.

Our Pasture-based applications incur an additional overhead in establishing encryption keys (one round trip delay) and generating keys in the receiver's TPM. Assuming there will be some *read slack time* between when an email is received in a user's inbox and when the user reads it, Pasture hides its latency by executing its pro-

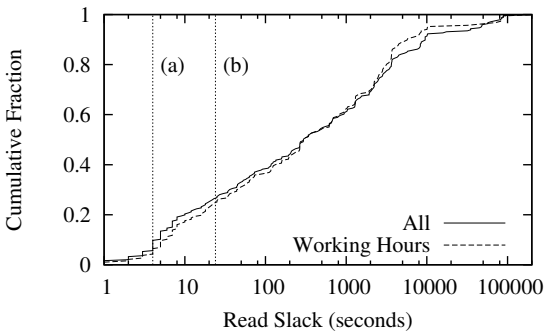



Figure 8: CDF of email read slack times.
Line (a) shows minimum Pasture overhead of 4s. Line (b) shows the total overhead of 24s including email server queuing and network delays. Only emails to the left of the lines are affected by Pasture.

tocol in the background. We evaluated the effectiveness of this approach via a small scale user study involving 5 users in a corporate setting. We logged the receive time and read time of the emails received over a period of one week for these users. To be conservative, we omitted unread emails.

Figure 8 shows the cumulative distribution of the read slack time of all the emails. We also include a separate line that considers only the emails that are received during work hours on a weekday. This removes any bias due to inactivity after work hours and over the weekend. For comparison, we measured the latency introduced by Pasture. The average additional latency introduced by Pasture was 24 s, with 4 s spent generating the encryption key at the receiver and about 20 s of network delay introduced by the processing queues of intermediate mail servers (the WAN network latencies to the mail servers was negligible, on the order of 40 ms).

We plot two vertical lines to show what fraction of emails are affected by the Pasture overhead. The affected emails are the ones whose recipients would have to wait. As shown in the figure, about 75% of emails are *not* affected by the total overhead introduced by Pasture, as their recipients checked their inbox more than 24 s after the email arrived. Even in the remaining 25% of emails which are affected, the Pasture protocol adds a delay of 15 s on average. If the processing and queuing delays at the mail servers are to be discounted, more than 90% (and 95% during the work time) of the emails are unaffected, and the rest experience an average delay of only 1 s. We conclude that Pasture can exploit the slack time of users to effectively hide the additional latency introduced for security. Furthermore, in the video rental application, Pasture latencies incurred can also be hidden while the encrypted movie downloads.

#### 5.3.2 Decentralized storage system

We implemented a shared folder application on top of the Pasture decentralized storage system. The shared folder application allows nodes to update files locally when they are disconnected (similar to other weakly-connected decentralized storage systems [31, 39, 50]), attest updated files, and share file updates with other peers opportunistically when they are connected.

**Scalable throughput using batching.** Given that nodes are expected to read all the received updates, the Pasture shared folder application amortizes overheads by batching and performing secure attestation of updates, message exchange protocol operations, and read operations on an entire batch of received updates. Pasture scales linearly to about 1000 requests/sec (with a batch size of 460) because the overheads to attest updates (430 ms) and read (460 ms) received updates are independent of the batch size. This is because Pasture uses a constant-size hash of the message when performing an attest or read operation. We conclude that Pasture provides scalable throughput for applications where nodes have the opportunity to batch updates. Batching also reduces the recovery response time as we amortize the PCR extensions across multiple updates.

**High durability.** Pasture does not perform NVRAM writes or counter increments during common case operations. Only two NVRAM writes and one counter increment are required for each reboot cycle. Hence, assuming only one reboot per day, it would take Pasture more than 130 years to exhaust the 100K lifetime NVRAM and counter update limit of current TPMs [37]. Conversely, if a 5 year lifetime is acceptable, Pasture can perform a reboot cycle on an hourly basis and truncate the log to reduce audit and recover times.

## 6 Discussion

Pasture effectively hides and amortizes TPM overheads for applications with low concurrency. However, TPM's limited resources and high overheads hurt the scalability of Pasture with increasing concurrent requests. We discuss three key limitations and suggest improvements through modest enhancements in future TPMs.

First, Pasture's concurrency is limited by the number of available PCRs (13 or fewer) for binding keys. Increasing the number of PCRs would improve Pasture's ability to bind more keys to PCRs and have more requests in flight to support applications with high concurrency.

Second, Pasture spends a significant amount of time in TPM_CreateWrapKey to generate keys. This overhead could be reduced by simply separating key generation from the operation of key binding. For example, TPMs could generate keys whenever they are idle (or in parallel with other operations) and store them in an internal buffer for future use.

Third, TPMs allow storage of only a few keys, and we have to pay significant overheads to load a key (around 1 s) if it has to be brought from the stable storage ev-

ery time. Increasing the buffer space to hold more keys would significantly reduce the overhead for highly concurrent applications.

## 7 Related work

Pasture builds upon related work in the areas of secure decentralized systems and trusted hardware.

**Custom hardware.** Many custom hardware architectures [30, 38, 46, 47, 53] have been proposed to protect trusted code from untrusted entities. These proposals have not gained widespread acceptance due to the high deployment cost and unavailability of custom hardware.

**OS and VMM based approaches.** A number of OS [26, 45, 49, 55], microkernel [25, 44] and VMM [11, 56] based approaches improve isolation and security by significantly reducing the TCB. A recent paper [45] provides an in depth review of the state of the art in this area. HiStar [55], Asbestos [52], and Flume [26] provide systemwide data confidentiality and integrity using decentralized information flow control techniques. Nexus [45] implements NAL logic [42] to provide general trustworthy assurances about the dynamic state of the system. While these systems provide powerful and general security primitives, they are vulnerable to hardware-snooping physical attacks [16, 21].

**Commodity trusted hardware.** Flicker [34] spurred research in secure execution by demonstrating how TPMs and SEM can be used to run trusted application code in isolation from the OS or hypervisor. TrustVisor [33] builds upon Flicker to protect sensitive application code in a single legacy guest VM. Memoir [37] prevents rollback attacks as described in §2.4. While these approaches provide strong isolation, they are vulnerable to hardware snooping attacks and also require frequent use of SEM, which can result in poor user responsiveness, resource underutilization and higher overhead.

Trusted hardware has been used to reduce email spam [14], in secure databases [32], for reasoning about network properties [43], for cloaked computation [9] by malware, and to provide trusted path [57] between a user's I/O device and trusted application code with minimal TCB. Recent approaches [3, 10] address an orthogonal issue of sharing TPM resources securely across multiple VMs. It is an avenue for future research to apply their approach to Pasture to share TPM resources across multiple receiver applications.

**Preventing attacks in decentralized systems.** Keypad [12] provides a simple online security mechanism for theft-prone devices by storing decryption keys at a server that client devices consult on every access attempt. A2M [7] prevents equivocation attacks [28, 29] by forcing attestation of updates into an append-only log using trusted hardware. TrInc [28] improves upon A2M by reducing the TCB to a trusted monotonic counter [41]. PeerReview [15], Nysiad [19], and other log-based replication systems [29, 31, 54] detect equivocation attacks without trusted hardware by using witness nodes and signed message logs. However, these systems do not prevent nor detect offline read-denial attacks.

## 8 Conclusion

Mobile user experiences are enriched by applications that support disconnected operations to provide better mobility, availability, and response time. However, offline data access is at odds with security when the user is not trusted, especially in the case of mobile devices, which must be assumed to be under the full control of the user.

Pasture provides secure disconnected access to data, enabling the untrusted user to obtain or revoke access to (previously downloaded) data and have his actions securely logged for later auditing. We implement Pasture using commodity trusted hardware, providing its security guarantees with an acceptable overhead using a small trusted computing base.

## Acknowledgements

We thank Doug Terry, Ted Wobber, and Peter Haynes for providing considerable suggestions to the Pasture project, and the anonymous OSDI reviewers for their detailed reviews and excellent feedback. We thank Martin Abadi, Rama Ramasubramanian, Jay Lorch, Dahlia Malkhi, JP Martin, and Marcos Aguilera for their feedback on earlier drafts of this paper. We are grateful to Stefan Thom for helping us with the Windows code base.

## References

[1] Advanced Micro Devices. *AMD64 virtualization: Secure virtual machine architecture reference manual*, 3.01 edition, 2005.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.

[3] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. Doorn. vTPM: Virtualizing the trusted platform module. In *USENIX Security*, 2006.

[4] BitLocker Drive Encryption Overview. http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview. Aug 31, 2012.

[5] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, Jan 2008.

[6] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, 2010.

[7] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.

[8] Delivering a secure and fast boot experience with UEFI. http://channel9.msdn.com/events/BUILD/BUILD2011/HW-457T. Aug 31, 2012.

[9] A. M. Dunn, O. S. Hofmann, B. Waters, and E. Witchel. Cloaking malware with the trusted platform module. In *Proceedings of the 20th USENIX conference on Security*.

[10] P. England and J. Loeser. Para-virtualized TPM sharing. In *TRUST*, 2008.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A VM-based platform for trusted computing. In *SOSP*, 2003.

[12] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *EuroSys*, 2011.

[13] Gmail delivery errors divulge confidential information. http://news.cnet.com/8301-13880_3-10438580-68.htm. Aug 31, 2012.

[14] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving service availability in the face of botnet attacks. In *NSDI*, 2009.

[15] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[16] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[17] T. Hardjono and G. Kazmierczak. Overview of the TPM Key Management Standard. http://www.trustedcomputinggroup.org/files/resource_files/ABEDDF95-1D09-3519-AD65431FC12992B4/Kazmierczak20Greg20-20TPM_Key_Management_KMS2008_v003.pdf. Aug 31, 2012.

[18] Health Information Technology for Economic and Clinical Health Act. http://en.wikipedia.org/wiki/HITECH_Act. Aug 31, 2012.

[19] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *NSDI*, 2008.

[20] Hospital fined over privacy breaches in days after deaths of Jackson, Fawcett. http://www.phiprivacy.net/?p=2888. Aug 31, 2012.

[21] A. Huang. *Hacking the XBOX: An Introduction to Reverse Engineering*. No Starch Press, 2003.

[22] Intel Corporation. *LaGrande technology preliminary architecture specification.*, d52212 edition, 2006.

[23] Johns Hopkins University e-mail attachment error exposed personal info. http://www.phiprivacy.net/?p=4583. Aug 31, 2012.

[24] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, 1992.

[25] G. Klein et al. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[26] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[27] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[28] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[30] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, 2010.

[32] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, 2000.

[33] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[34] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[35] A. Menezes, P. van Oorschot, and S. Vanstone, editors. *Handbook of Applied Cryptography (Discrete Mathematics and Its Applications)*. CRC Press, Dec. 1996.

[36] Microsoft Outlook 2012. http://office.microsoft.com/en-us/outlook/. Aug 31, 2012.

[37] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy*, 2011.

[38] A. Perrig, S. Smith, D. Song, and J. D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. In *IPDPS*, 1991.

[39] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.

[40] T. L. Rodeheffer and R. Kotla. Pasture node state specification. Technical Report MSR-TR-2012-84, Microsoft Research, Aug 2012.

[41] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *ACM Workshop on Scalable Trusted Computing*, 2006.

[42] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8:1–8:28, June 2011.

[43] A. Shieh, E. G. Sirer, and F. B. Schneider. NetQuery: A knowledge plane for reasoning about network properties. *SIGCOMM Comput. Commun. Rev.*, 41(4):278–289, Aug. 2011.

[44] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.

[45] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *SOSP*, 2011.

[46] S. W. Smith and J. D. Tygar. Security and privacy for partial order time. In *PDCS*, 1994.

[47] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor. *SIGARCH Comput. Archit. News*, 33(2):25–36, 2005.

[48] Summary of the HIPPA Security Rule. http://www.hhs.gov/ocr/privacy/hipaa/understanding/srsummary.html. Aug 31, 2012.

[49] R. Ta-min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[50] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[51] TPM Main Specification Level 2 Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. Aug 31, 2012.

[52] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4), Dec. 2007.

[53] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *USENIX Workshop on Electronic Commerce*, 1995.

[54] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *IIEEE Trans. on Storage*, 3(3), 2007.

[55] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[56] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

[57] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

# Dune: Safe User-level Access to Privileged CPU Features

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, Christos Kozyrakis
Stanford University

## Abstract

*Dune* is a system that provides applications with direct but safe access to hardware features such as ring protection, page tables, and tagged TLBs, while preserving the existing OS interfaces for processes. Dune uses the virtualization hardware in modern processors to provide a *process*, rather than a *machine* abstraction. It consists of a small kernel module that initializes virtualization hardware and mediates interactions with the kernel, and a user-level library that helps applications manage privileged hardware features. We present the implementation of Dune for 64-bit x86 Linux. We use Dune to implement three user-level applications that can benefit from access to privileged hardware: a sandbox for untrusted code, a privilege separation facility, and a garbage collector. The use of Dune greatly simplifies the implementation of these applications and provides significant performance advantages.

## 1 Introduction

A wide variety of applications stand to benefit from access to "kernel-only" hardware features. As one example, Azul Systems demonstrates significant speedups to garbage collection through use of paging hardware [15, 36]. As another example, process migration, though implementable within a user program, can benefit considerably from access to page faults [40] and system calls [32]. In some cases, it might even be appropriate to replace the kernel entirely to meet the needs of a particular application. For example, IBOS improves browser security by moving browser abstractions into the lowest OS layers [35].

Such systems require changes in the kernel because hardware access in userspace is restricted for security and isolation reasons. Unfortunately, modifying the kernel is not ideal in practice because kernel changes can be fairly intrusive and, if done incorrectly, affect whole system stability. Moreover, if multiple applications require kernel changes, there is no guarantee that the changes will compose.

Another strategy is to bundle applications into virtual machine images with specialized kernels [4, 14]. Many modern CPUs contain virtualization hardware with which guest operating systems can safely and efficiently access kernel hardware features. Moreover, virtual machines provide failure containment similar to that of processes—*i.e.,* buggy or malicious behavior should not bring down the entire physical machine.

Unfortunately, virtual machines offer poor integration with the host operating system. Processes expect to inherit file descriptors from their parents, spawn other processes, share a file system and devices with their parents and children, and use IPC services such as Unix-domain sockets. Moving a process to a virtual machine for the purposes of, say, speeding up garbage collection is likely to break many assumptions and may simply not be worth the hassle. Moreover, producing a kernel for an application-specific virtual machine is no small task. Production kernels such as Linux are complex and hard to modify. Yet implementing a special-purpose kernel with a simple virtual memory layer is also challenging. In addition to virtual memory, one must support a file system, a networking stack, device drivers, and a bootstrap process.

This paper introduces a new approach to application use of kernel hardware features: using virtualization hardware to provide a *process*, rather than a *machine* abstraction. We have implemented this approach for Linux on 64-bit Intel CPUs in a system called *Dune*. Dune provides a loadable kernel module that works with unmodified Linux kernels. The module allows processes to enter "Dune mode," an irreversible transition in which, through virtualization hardware, safe and fast access to privileged hardware features is enabled, including privilege modes, virtual memory registers, page tables, and interrupt, exception, and system call vectors. We provide a user-level library, libDune, to facilitate the use of these features.

For applications that fit its paradigm, Dune offers several advantages over virtual machines. First, a Dune process is a normal Linux process, the only difference being that it uses the `VMCALL` instruction to invoke system calls. This means that Dune processes have full access to the rest of the system and are an integral part of it, and

that Dune applications are easy to develop (like application programming, not kernel programming). Second, because the Dune kernel module is not attempting to provide a machine abstraction, the module can be both simpler and faster. In particular, the virtualization hardware can be configured to avoid saving and restoring several pieces of hardware state that would be required for a virtual machine.

With Dune we contribute the following:

- We present a design that uses hardware-assisted virtualization to safely and efficiently expose privileged hardware features to user programs while preserving standard OS abstractions.
- We evaluate three hardware features in detail and show how they can benefit user programs: exceptions, paging, and privilege modes.
- We demonstrate the end-to-end utility of Dune by implementing and evaluating three use cases: sandboxing, privilege separation, and garbage collection.

# 2 Virtualization and Hardware

In this section, we review the hardware support for virtualization and discuss which privileged hardware features Dune is able to expose. Throughout the paper, we describe Dune in terms of x86 CPUs and Intel VT-x. However, this is not fundamental to our design, and in Section 7, we broaden our discussion to include other architectures that could be supported in the future.

## 2.1 The Intel VT-x Extension

In order to improve virtualization performance and simplify VMM implementation, Intel has developed VT-x [37], a virtualization extension to the x86 ISA. AMD also provides a similar extension with a different hardware interface called SVM [3].

The simplest method of adapting hardware to support virtualization is to introduce a mechanism for trapping each instruction that accesses privileged state so that emulation can be performed by a VMM. VT-x embraces a more sophisticated approach, inspired by IBM's interpretive execution architecture [31], where as many instructions as possible, including most that access privileged state, are executed directly in hardware without any intervention from the VMM. This is possible because hardware maintains a "shadow copy" of privileged state. The motivation for this approach is to increase performance, as traps can be a significant source of overhead.

VT-x adopts a design where the CPU is split into two operating modes: *VMX root* and *VMX non-root* mode. VMX root mode is generally used to run the VMM and does not change CPU behavior, except to enable access to new instructions for managing VT-x. VMX non-root mode, on the other hand, restricts CPU behavior and is intended for running virtualized guest OSes.

Transitions between VMX modes are managed by hardware. When the VMM executes the `VMLAUNCH` or `VMRESUME` instruction, hardware performs a *VM entry*; placing the CPU in VMX non-root mode and executing the guest. Then, when action is required from the VMM, hardware performs a *VM exit*, placing the CPU back in VMX root mode and jumping to a VMM entry point. Hardware automatically saves and restores most architectural state during both types of transitions. This is accomplished by using buffers in a memory resident data structure called the VM control structure (VMCS).

In addition to storing architectural state, the VMCS contains a myriad of configuration parameters that allow the VMM to control execution and specify which type of events should generate VM exits. This gives the VMM considerable flexibility in determining which hardware is exposed to the guest. For example, a VMM could configure the VMCS so that the `HLT` instruction causes a VM exit or it could allow the guest to halt the CPU. However, some hardware interfaces, such as the interrupt descriptor table (IDT) and privilege modes, are exposed implicitly in VMX non-root mode and never generate VM exits when accessed. Moreover, a guest can manually request a VM exit by using the `VMCALL` instruction.

Virtual memory is perhaps the most difficult hardware feature for a VMM to expose safely. A straw man solution would be to configure the VMCS so that the guest has access to the page table root register, `%CR3`. However, this would place complete trust in the guest because it would be possible for it to configure the page table to access any physical memory address, including memory that belongs to the VMM. Fortunately, VT-x includes a dedicated hardware mechanism, called the *extended page table* (EPT), that can enforce memory isolation on guests with direct access to virtual memory. It works by applying a second, underlying, layer of address translation that can only be configured by the VMM. AMD's SVM includes a similar mechanism to the EPT, referred to as a nested page table (NPT).

## 2.2 Supported Hardware Features

Dune uses VT-x to provide user programs with full access to x86 protection hardware. This includes three privileged hardware features: exceptions, virtual memory, and privilege modes. Table 1 shows the corresponding privileged

| Mechanism | Privileged Instructions |
|---|---|
| Exceptions | LIDT, LTR, IRET, STI, CLI |
| Virtual Memory | MOV CRn, INVLPG, INVPCID |
| Privilege Modes | SYSRET, SYSEXIT, IRET |
| Segmentation | LGDT, LLDT |

Table 1: Hardware features exposed by Dune and their corresponding privileged x86 instructions.

instructions made available for each feature. Dune also exposes segmentation, but we do not discuss it further, as it is primarily a legacy mechanism on modern x86 CPUs.

Efficient support for exceptions is important in a variety of use cases such as emulation, debugging, and performance tracing. Normally, reporting an exception to a user program requires privilege mode transitions and an upcall mechanism (*e.g.,* signals). Dune can reduce exception overhead because it uses VT-x to deliver exceptions directly in hardware. This does not, however, allow a Dune process to monopolize the CPU, as timer interrupts and other exceptions intended for the kernel will still cause a VM exit. The net result is that software overhead is eliminated and exception performance is determined by hardware efficiency alone. As just one example, Dune improves the speed of delivering page fault exceptions, when compared to SIGSEGV in Linux, by more than 4×. Several other types of exceptions are also accelerated, including breakpoints, floating point overflow and underflow, divide by zero, and invalid opcodes.

User programs can also benefit from fast and flexible access to virtual memory [5]. Use cases include checkpointing, garbage collection (evaluated in this paper), data-compression paging, and distributed shared memory. Dune improves virtual memory access by exposing page table entries to user programs directly, allowing them to control address translations, access permissions, global bits, and modified/accessed bits with simple memory references. In contrast, even the most efficient OS interfaces [17] add extra latency by requiring system calls in order to perform these operations. Letting applications write their own page tables does not affect security because the underlying EPT exposes only the normal process address space, which is equally accessible without Dune.

Dune also gives user programs the ability to manually control TLB invalidations. As a result, page table updates can be performed in batches when permitted by the application. This is considerably more challenging to support in the kernel because it is difficult to defer TLB invalidations when general correctness must be maintained. In addition, Dune exposes TLB tagging by providing access to Intel's recently added process-context identifier (PCID) feature. This permits a single user program to switch between multiple page tables efficiently. All together, we show that using Dune results in a 7× speedup over Linux in the Appel and Li user-level virtual memory benchmarks [5]. This figure includes the use of exception hardware to reduce page fault latency.

Finally, Dune exposes access to privilege modes. On x86, the most important privilege modes are ring 0 (supervisor mode) and ring 3 (user mode), although rings 1 and 2 are also available. Two motivating use cases for privilege modes are privilege separation and sandboxing of untrusted code, both evaluated in this paper. Dune can support privilege modes efficiently because VMX non-root mode maintains its own set of privilege rings. Hence, Dune allows hardware-enforced protection within a process in exactly the way kernels protect themselves from user processes. The supervisor bit in the page table is available to control memory isolation. Moreover, system call instructions trap to the process itself, rather than to the kernel, which can be used for system call interposition and to prevent untrusted code from directly accessing the kernel. Compared to *ptrace* in Linux, we show that Dune can intercept a system call with 25× less overhead.

Although the hardware features Dune exposes suffice in supporting our motivating use cases, several other hardware features, such as cache control, debug registers, and access to DMA-capable devices, could also be safely exposed through virtualization hardware. We leave these for future work and discuss their potential in Section 7.

# 3 Kernel Support for Dune

The core of Dune is a kernel module that manages VT-x and provides user programs with greater access to privileged hardware features. We describe this module here, including a system overview, a threat model, and a comparison to an ordinary VMM. We then explore three key aspects of the module's operation: managing memory, exposing access to privileged hardware, and preserving access to kernel interfaces. Finally, we describe the Dune module we implemented for the Linux kernel.

## 3.1 System Overview

Figure 1 shows a high-level view of the Dune architecture. Dune extends the kernel with a module that enables VT-x, placing the kernel in VMX root mode. Processes using Dune are granted direct but safe access to privileged hardware by running in VMX non-root mode. The Dune module intercepts VM exits, the only means for a Dune pro-

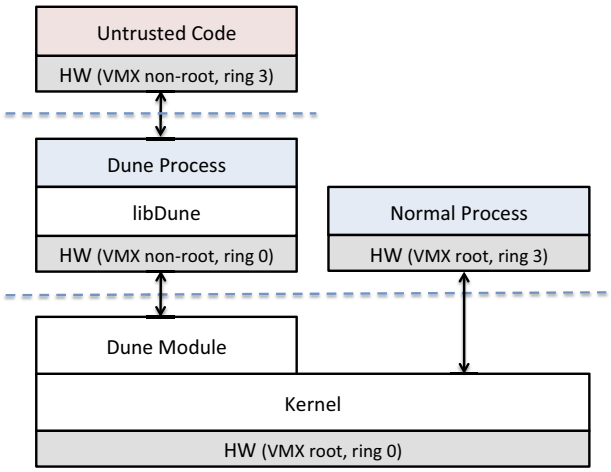



Figure 1: The Dune system architecture.

cess to access the kernel, and performs any necessary actions such as servicing a page fault, calling a system call, or yielding the CPU after a `HLT` instruction. Dune also includes a library, called *libDune*, to assist with managing privileged hardware features in userspace, discussed further in Section 4.

We apply Dune selectively to processes that need it; processes that do not use Dune are completely unaffected. A process can enable Dune at any point by initiating a transition through an *ioctl* on the `/dev/dune` device, but once in Dune mode, a process cannot exit Dune mode. Whenever a Dune process forks, the child process does not start in Dune mode, but can re-enter Dune if the use case requires it.

The Dune module requires VT-x. As a result, it cannot be used inside a VM unless there is support for nested VT-x [6]; the performance characteristics of such a configuration are an interesting topic of future consideration. On the other hand, it is possible to run a VMM on the same machine as the Dune module, even if the VMM requires VT-x, because VT-x can be controlled independently on each core.

## 3.2 Threat Model

Dune exposes privileged CPU features without affecting the existing security model of the underlying OS. Any external effects produced by a Dune-enabled process could be produced without Dune through the same series of system calls. However, by exposing hardware privilege modes, Dune enables additional privilege-separation techniques within a process that would not otherwise be practical.

We assume that the CPU is free of defects, although we acknowledge that in rare cases exploitable hardware flaws have been identified [26, 27].

## 3.3 Comparing to a VMM

Though all software using VT-x shares a common structure, Dune's use of VT-x deviates from that of standard VMMs. Specifically, Dune exposes a process environment instead of a machine environment. As a result, Dune is not capable of supporting a normal guest OS, but this permits Dune to be lighter weight and more flexible. Some of the most significant differences are as follows:

- Hypercalls are a common way for VMMs to support paravirtualization, a technique in which the guest OS is modified to use interfaces that are more efficient and less difficult to virtualize. In Dune, by contrast, the hypercall mechanism invokes normal Linux system calls. For example, a VMM might provide a hypercall to register an interrupt handler for a virtual network device, whereas a Dune process would use a hypercall to call *read* on a TCP socket.

- Many VMMs emulate physical hardware interfaces in order to support unmodified guest OSes. In Dune, only hardware features that can be directly accessed without VMM intervention are made available; in cases where this is not possible, a Dune process falls back on the OS. For example, most VMMs go to great lengths to present a virtual graphics card interface in order to support a frame buffer. By contrast, Dune processes employ the normal OS display service, usually an X server accessed over a Unix-domain socket and shared memory.

- A typical VMM must save and restore all state that is necessary to support a guest OS. In Dune, we can limit the differences in guest and host state because processes using Dune have a narrower hardware interface. This results in reductions to the overhead of performing VM entries and VM exits.

- VMMs place each VM in a separate address space that emulates flat physical memory. In Dune, we configure the EPT to reflect process address spaces. As a result, the memory layout can be sparse and memory can be coherently shared when two processes map the same memory segment.

Despite these differences, the Dune module could be considered a type-2 hypervisor [22] because it runs on top of an existing OS kernel.

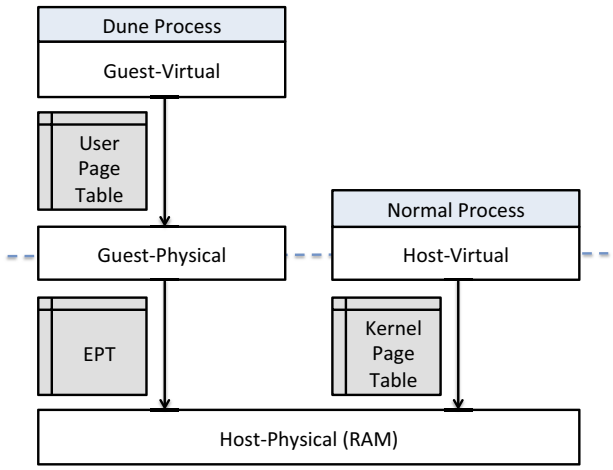



Figure 2: Virtual memory in Dune.

## 3.4 Memory Management

Memory management is one of the biggest responsibilities of the Dune module. The challenge is to expose direct page table access to user programs while preventing arbitrary access to physical memory. Moreover, our goal is to provide a normal process memory address space by default, permitting user programs to add just the functionality they need instead of completely replacing kernel-level memory management.

Paging translations occur in three separate cases in Dune, shown in Figure 2. One translation is specified by the kernel's standard page table. In virtualization terminology this is the host-virtual to host-physical (*i.e.,* raw memory) translation. Host-virtual addresses are ordinary virtual addresses, but they are only used by the kernel and normal processes. For processes using Dune, a user controlled page table maps guest-virtual addresses to guest-physical. Then the EPT, managed by the kernel, performs an additional translation from guest-physical to host-physical. All memory references made by processes using Dune can only be guest-virtual, allowing for isolation and correctness to be enforced in the EPT while application-specific functionality and optimizations can be applied in the user page table.

Ideally, we would like to match the EPT to the kernel's page table as closely as possible because of our goal to give processes using Dune access to the same address space they would have as normal processes. If it were permitted by hardware, we would simply point the EPT and the kernel's page table to the same page root. Unfortunately, two limitations make this impossible. First, the EPT requires a different binary format from the standard x86 page table. Second, Intel x86 processors limit the address width of guest-physical addresses to be the same as host-physical addresses. In a standard virtual machine environment this would not be a concern because any machine being emulated would have a realistically bounded amount of RAM. For Dune, however, the problem is that we want to expose the full host-virtual address space and yet the guest-physical address space is limited to a smaller size (*e.g.,* a 36-bit physical limit vs. a 48-bit virtual limit on many contemporary Intel processors). We note that this issue is not present when running in 32-bit protected mode, as physical addresses are at least as large as virtual addresses.

Our solution to EPT format incompatibility is to query the kernel for process memory mappings and to manually update the EPT to reflect them. We start with an empty EPT. Then, we receive an EPT fault (a type of VM exit) each time a missing EPT entry is accessed. The fault handler crafts a new EPT entry that reflects an address translation and permission reported by the kernel's page fault handler. Occasionally, address ranges will need to be unmapped. In addition, the kernel requires page access information, to assist with swapping, and page dirty status, to determine when write-back to disk is necessary. Dune supports all of these cases by hooking into an MMU notifier chain, the same approach used by KVM [30]. For example, when an address is unmapped, the Dune module receives an event. It then evicts affected EPT entries and sets dirty bits in the appropriate Linux page structures.

We work around the address width issue by allowing only some address ranges to be mapped in the EPT. Specifically, we only permit addresses from the beginning of the process (*i.e.,* the heap, code, and data segments), the mmap region, and the stack. Currently, we limit each of these regions to 4GB, allowing us to compress the address space to fit in the first 12GB of the EPT. Typically the user's page table will then expand the addresses to their original layout. This could result in incompatibilities in programs that use nonstandard portions of the address space, though such cases are rare. A more sophisticated solution might pack each virtual memory area into the guest-physical address space in arbitrary order and then provide the user program the additional information required to remap the segment to the correct guest-virtual address in its own page table, thus avoiding the possibility of unaddressable memory regions.

## 3.5 Exposing Access to Hardware

As discussed previously, Dune exposes access to exceptions, virtual memory, and privilege modes. Exceptions and privilege modes are implicitly available in VMX non-

root mode and do not require any special configuration. On the other hand, virtual memory requires access to the %CR3 register, which can be granted in the VMCS. We maintain a separate VMCS for each process in order to allow for per-process configuration of privileged state and to support context switching more easily and efficiently.

x86 includes a variety of control registers that determine which hardware features are enabled (*e.g.,* floating point, SSE, no execute, *etc.*) Although we could have permitted Dune processes to configure these directly, we instead mirror the configuration set by the kernel. This allows us to support a normal process environment; permitting many configuration changes would break compatibility with user programs. For example, it makes little sense for a 64-bit process to disable long mode. There are, however, a couple of important exceptions to this rule. First, we allow user programs to disable paging because it is the only method available on x86 to clear global TLB entries. Second, we give user programs some control over floating point hardware in order to allow for support of lazy floating point state management.

In some cases, Dune restricts access to hardware registers for performance reasons. For instance, Dune does not allow modification to MSRs in order to avoid the relatively high overhead of saving and restoring them during each system call. The FS and GS base registers are exceptions because they are not only used frequently but are also saved and restored by hardware automatically. MSR_LSTAR, which contains the address of the system call handler, is a special case where Dune allows read-only access. This allows a user process to map code for a system call handler at the existing address (by manipulating its page table) instead of changing the register to a new address and, as a result, harming performance.

Dune exposes raw access to the time stamp counter (TSC). By contrast, most VMMs virtualize the TSC in order to avoid confusing guest kernels, which tend to make timing assumptions that could be violated if time spent in the VMM is made visible.

## 3.6 Preserving OS Interfaces

In addition to exposing privileged hardware features, Dune preserves access to OS system calls. Normal system call invocation instructions will only trap within the process itself and do not cause a VM exit. Instead, processes must use VMCALL, the hypercall instruction, to make system calls. The Dune module vectors hypercalls through the kernel's system call table. In some cases, it must perform extra actions before calling the system call handler. For example, during an *exit* system call, Dune performs cleanup tasks.

Dune completely changes how signal handlers are invoked. Some signals are obviated by more efficient direct hardware support. For example, hardware page faults largely subsume the role of SIGSEGV. For other signals (*e.g.,* SIGINT), the Dune module injects fake hardware interrupts into the process. This is not only an efficient mechanism, but also has the advantage of correctly composing with privilege modes. For example, if a user process were running in ring 3 to sandbox untrusted code, hardware would automatically transition it to ring 0 in order to service the signal securely.

## 3.7 Implementation

Dune presently supports Linux running on Intel x86 processors in 64-bit long mode. Support for AMD CPUs and 32-bit mode are possible future additions. In order to keep changes to the kernel as unintrusive as possible, we developed Dune as a dynamically loadable kernel module. Our implementation is based partially on KVM [30]. Specifically, it shares code for managing low-level VT-x operations. However, high-level code is not shared with KVM because Dune operates differently from a VMM. Furthermore, our Dune module is simpler than KVM, consisting of only 2,509 lines of code.

In Linux, user threads are supported by the kernel, making them nearly identical to processes except they have a shared address space. As a result, it was easiest for us to create a VMCS for each thread instead of merely each process. One interesting consequence is that it is possible for both threads using Dune and threads not using Dune to belong to the same process.

Our implementation is capable of supporting thousands of processes at a time. The reason is that processes using Dune are substantially lighter-weight than full virtual machines. Efficiency is further improved by using virtual-processor identifiers (VPIDs). VPIDs enable a unique TLB tag to be assigned to each Dune process, and, as a result, hypercalls and context switches do not require TLB invalidations.

One limitation in our implementation is that we cannot efficiently detect when EPT pages have been modified or accessed, which is needed for swapping. Intel recently added hardware support for this capability, so it should be easy to rectify this limitation. For now, we take a conservative approach and always report pages as modified and accessed during MMU notifications in order to ensure correctness.

# 4 User-mode Environment

The execution environment of a process using Dune has some differences from a normal process. Because privilege rings are an exposed hardware feature, one difference is that user code runs in ring 0. Despite changing the behavior of certain instructions, this does not typically result in any incompatibilities for existing code. Ring 3 is also available and can optionally be used to confine untrusted code. Another difference is that system calls must be performed as hypercalls. To simplify supporting this change, we provide a mechanism that can detect when a system call is performed from ring 0 and automatically redirect it to the kernel as a hypercall. This is one of many features included in libDune.

libDune is a library created to make it easier to build user programs that make use of Dune. It is completely untrusted by the kernel and consists of a collection of utilities that aid in managing and configuring privileged hardware features. Major components of libDune include a page table manager, an ELF loader, a simple page allocator, and routines that assist user programs in managing exceptions and system calls. libDune is currently 5,898 lines of code.

We also provide an optional, modified version of libc that uses `VMCALL` instructions instead of `SYSCALL` instructions in order to get a slight performance benefit.

## 4.1 Bootstrapping

In many ways, transitioning a process into Dune mode is similar to booting an OS. The first issue is that a valid page table must be provided before enabling Dune. A simple identity mapping is insufficient because, although the goal is to have process addresses remain consistent before and after the transition, the compressed layout of the EPT must be taken into account. After a page table is created, the Dune entry *ioctl* is called with the page table root as an argument. The Dune module then switches the process to Dune mode and begins executing code, using the provided page table root as the initial `%CR3`. From there, libDune configures privileged registers to set up a reasonable operating environment. For example, it loads a GDT to provide basic flat segmentation and loads an IDT so that hardware exceptions can be captured. It also sets up a separate stack in the TSS to handle double faults and configures the GS segment base in order to easily access per-thread data.

## 4.2 Limitations

Although we are able to run a wide variety of Linux programs, libDune is still missing some functionality. First, we have not fully integrated support for signals despite the fact that they are reported by the Dune module. Applications are required to use *dune_signal* whereas a more compatible solution would override several libc symbols like *signal* and *sigaction*. Second, although we support pthreads, some utilities in libDune, such as page table management, are not yet thread-safe. Both of these issues could be resolved with further implementation.

One unanticipated challenge with working in a Dune environment is that system call arguments must be valid host-virtual addresses, regardless of how guest-virtual mappings are setup. In many ways, this parallels the need to provide physical addresses to hardware devices that perform DMA. In most cases we can work around the issue by having the guest-virtual address space mirror the host-virtual address space. For situations where this is not possible, walking the user page table to adjust system call argument addresses is necessary.

Another challenge introduced by Dune is that by exposing greater access to privileged hardware, user programs require more architecture-specific code, potentially reducing portability. libDune currently provides an x86-centric API, so it is already compatible with AMD machines. However, it should be possible to modify libDune to support non-x86 architectures in a fashion that parallels the construction of many OS kernels. This would require libDune to provide an efficient architecture independent interface, a topic worth exploring in future revisions.

# 5 Applications

Dune is generic enough that it lets us improve on a broad range of applications. We built two security-related applications, a sandbox and privilege separation system, and one performance-related application, a garbage collector. Our goals were simpler implementations, higher performance, and where applicable, improved security.

## 5.1 Sandboxing

Sandboxing is the process of confining code so as to restrict the memory it can access and the interfaces or system calls it can use. It is useful for a variety of purposes, such as running native code in web browsers, creating secure OS containers, and securing mobile phone applications. In order to explore Dune's potential for these types

of applications, we built a sandbox that supports native 64-bit Linux executables.

The sandbox enforces security through privilege modes by running a trusted sandbox runtime in ring 0 and an untrusted binary in ring 3, both operating within a single address space (on the left of Figure 1, the top and middle boxes respectively). Memory belonging to the sandbox runtime is protected by setting the supervisor bit in appropriate page table entries. Whenever the untrusted binary performs an unsafe operation such as trying to access the kernel through a system call or attempting to modify privileged state, libDune receives an exception and jumps into a handler provided by the sandbox runtime. In this way, the sandbox runtime is able to filter and restrict the behavior of the untrusted binary.

While we rely on the kernel to load the sandbox runtime, the untrusted binary must be loaded in userspace. One risk is that it could contain maliciously crafted headers designed to exploit flaws in the ELF loader. We hardened our sandbox against this possibility by using two separate ELF loaders. First, the sandbox runtime uses a minimal ELF loader (part of libDune), that only supports static binaries, to load a second ELF loader into the untrusted environment. We choose to use *ld-linux.so* as our second ELF loader because it is already used as an integral and trusted component in Linux. Then, the sandbox runtime executes the untrusted environment, allowing the second ELF loader to load an untrusted binary entirely from ring 3. Thus, even if the untrusted binary is malicious, it does not have a greater opportunity to attack the sandbox during ELF loading than it would while running inside the sandbox normally.

So far our sandbox has been applied primarily as a tool for filtering Linux system calls. However, it could potentially be used for other purposes, including providing a completely new system call interface. For system call filtering, a large concern is to prevent execution of any system call that could corrupt or disable the sandbox runtime. We protect against this hazard by validating each system call argument, checking to make sure performing the system call would not allow the untrusted binary to access or modify memory belonging to the sandbox runtime. We do not yet support all system calls, but we support enough to run most single-threaded Linux applications. However, nothing prevents supporting multi-threaded programs in the future.

We implemented two policies on top of the sandbox. Firstly, we support a null policy that allows system calls to pass through but still validates arguments in order to protect the sandbox runtime. It is intended primarily to demonstrate raw performance overhead. Secondly, we support a userspace firewall. It uses system call interposition to inspect important network system calls, such as *bind* and *connect*, and prevents communication with undesirable parties as specified by a policy description.

To further demonstrate the flexibility of our sandbox, we also implemented a checkpointing system that can serialize an application to disk and then restore execution at a later time. This includes saving memory, registers, and system call state (*e.g.,* open file descriptors).

## 5.2 Wedge

Wedge [10] is a privilege separation system. Its core abstraction is an *sthread* which provides *fork*-like isolation with *pthread*-like performance. An sthread is a lightweight process that has access to memory, file descriptors and system calls as specified by a policy. The idea is to run risky code in an sthread so that any exploits will be contained within it. In a web server, for example, each client request would run in a separate sthread to guarantee isolation between users. To make this practical, sthreads need fast creation (*e.g.,* one per request) and context switch time. Fast creation can be achieved through sthread *recycling*. Instead of creating and killing an sthread each time, an sthread is checkpointed on its first creation (while still pristine and unexploited) and restored on exit so that it can be safely reused upon the next creation request. Doing so reduces sthread creation cost to the (cheaper) cost of restoring memory.

Wedge uses many of Dune's hardware features. Ring protection is used to enforce system call policies; page tables limit what memory sthreads can access; dirty bits are used to restore memory during sthread recycling; and the tagged TLB is used for fast context switching.

## 5.3 Garbage Collection

Garbage collectors (GC) often utilize memory management hardware to speed up collection [28]. Appel and Li [5] explain several techniques that use standard user level virtual memory protection operations, whereas Azul Systems [15, 36] went to the extent of modifying the kernel and system call interface. By contrast, Dune provides a clean and efficient way to access relevant hardware directly. The features provided by Dune that are of interest to garbage collectors include:

- **Fast faults**. GCs often use memory protection and fault handling to implement read and write barriers.
- **Dirty bits**. Knowing what memory has been touched since the last collection enables optimizations and can be a core part of the algorithm.

- **Page table**. One optimization in a moving GC is to free the underlying physical frame without freeing the virtual page it was backing. This is useful when the data has been moved but references to the old location remain and can still be caught through page faults. Remapping memory can also be performed to reduce fragmentation.
- **TLB control**. GCs often manipulate memory mappings at high rates, making control over TLB invalidation very useful. If it can be controlled, mapping manipulations can be effectively batched, rendering certain algorithms more feasible.

We modified the Boehm GC [12] to use Dune in order to improve performance. The Boehm GC is a robust mark-sweep collector that supports parallel and incremental collection. It is designed either to be used as a conservative collector with C/C++ programs, or by compiler and run-time backends where the conservativeness can be controlled. It is widely used, including by the Mono project and GNU Objective C.

An important implementation question for the Boehm GC is how dirty pages are discovered and managed. The two original options were (i) utilizing *mprotect* and signal handlers to implement its own dirty bit tracking; or (ii) utilizing OS provided dirty bit read methods such as the Win32 API call *GetWriteWatch*. In Dune we support and improve both methods.

A direct port to Dune already gives a performance improvement because *mprotect* can directly manipulate the page table and a page fault can be handled directly without needing an expensive `SIGSEGV` signal. The GC manipulates single pages 90% of the time, so we were able to improve performance further by using the `INVPLG` instruction to flush only a single page instead of the entire TLB. Finally, in Dune, the Boehm GC can access dirty bits directly without having to emulate this functionality. Some OSes provide system calls for reading page table dirty bits. Not all of these interfaces are well matched to GC—for instance, SunOS examines the entire virtual address space rather than permit queries for a particular region. Linux provides no user-level access at all to dirty bits.

The work done on the Boehm GC represents a straightforward application of Dune to a GC. It is worth also examining the changes made by Azul Systems to Linux so that they could support their C4 GC [36] and mapping this to the support provided by Dune:

- **Fast faults**. Azul modified the Linux memory protection and mapping primitives to greatly improve performance, part of this included allowing hardware exceptions to bypass the kernel and be handled directly by usermode.
- **Batched page table**. Azul enabled explicit control of TLB invalidation and a shadow page table to expose a prepare and commit style API for batching page table manipulation.
- **Shatter/Heal**. Azul enabled large pages to be 'shattered' into small pages or a group of small pages to be 'healed' into a single large page.
- **Free physical frames**. When the Azul C4 collector frees an underlying physical frame, it will trap on accesses to the unmapped virtual pages in order to catch old references.

All of the above techniques and interfaces can be implemented efficiently on top of Dune, with no need for any kernel changes other than loading the Dune module.

# 6 Evaluation

In this section, we evaluate the performance and utility of Dune. Although using VT-x has an intrinsic cost, in most cases, Dune's overhead is relatively minor. On the other hand, Dune offers significant opportunities to improve security and performance for applications that can take advantage of access to privileged hardware features.

All tests were performed on a single-socket machine with an Intel Xeon E3-1230 v2 (a four core Ivy Bridge CPU clocked at 3.3 GHz) and 16GB of RAM. We installed a recent 64-bit version of Debian Linux that includes Linux kernel version 3.2. Power management features, such as frequency scaling, were disabled.

## 6.1 Overhead from Running in Dune

Performance in Dune is impacted by two main sources of overhead. First, VT-x increases the cost of entering and exiting the kernel—VM entries and VM exits are more expensive than fast system call instructions or exceptions. As a result, both system calls and other types of faults (*e.g.,* page faults) must pay a fixed cost in Dune. Second, using the EPT makes TLB misses more expensive because, in some cases, the hardware page walker must traverse two page tables instead of one.

We built synthetic benchmarks to measure both of these effects. Table 2 shows the overhead of system calls, page faults, and page table walks. For system calls, we manually performed *getpid*, an essentially null system call (worst case for Dune), and measured the round-trip latency. For page faults, we measured the time it took to fault in a pre-zeroed memory page by the kernel. Finally, for page table walks, we measured the time spent filling a TLB miss.

| | getpid | page fault | page walk |
|---|---|---|---|
| **Linux** | 138 | 2,687 | 35.8 |
| **Dune** | 895 | 5,093 | 86.4 |

Table 2: Average time (in cycles) of operations that have overhead in Dune compared to Linux.

| | ptrace | trap | appel1 | appel2 |
|---|---|---|---|---|
| **Linux** | 27,317 | 2,821 | 701,413 | 684,909 |
| **Dune** | 1,091 | 587 | 94,496 | 94,854 |

Table 3: Average time (in cycles) of operations that are faster in Dune compared to Linux.

Measuring TLB miss overhead required us to build a simple memory stress tool. It works by performing a random page-aligned walk across $2^{16}$ memory pages. This models a workload with poor memory locality, as nearly every memory access results in a last-level TLB miss. We then divided the total number of cycles spent waiting for page table walks, as reported by a performance counter, by the total number of memory references, giving us a cycle cost per page walk.

In general, the overhead Dune adds has only a small effect on end-to-end performance, as we show in Section 6.3. For system calls, the time spent in the kernel tends to be a larger cost than the fixed VMX mode transition costs. Page fault overhead is also not much of a concern, as page faults tend to occur infrequently during normal use, and direct access to exception hardware is available when higher performance is required. On the other hand, Dune's use of the EPT does impact performance in certain workloads. For applications with good memory locality or a small working set, it has no impact because the TLB hit rate is sufficiently high. However, for application with poor memory locality or a large working set, more frequent TLB misses result in a measurable slowdown. One effective strategy for limiting this overhead is to use large pages. We explore this possibility further in section 6.3.1.

## 6.2 Optimizations Made Possible by Dune

Access to privileged hardware features creates many opportunities for optimization. Table 3 shows speedups we achieved in the following OS workloads:

**ptrace** is a measure of system call interposition performance. This is the cost of a Linux process intercepting a system call (*getpid*) with *ptrace*, forwarding the system call to the kernel and returning the result. In Dune this is the cost of intercepting a system call directly using ring protection in VMX non-root mode, forwarding the system call through a `VMCALL` and returning the result. An additional scenario is where applications wish to intercept system calls but not forward them to the kernel and instead just implement them internally. `PTRACE_SYSEMU` is the most efficient mechanism for doing so since *ptrace* requires forwarding a call to the kernel. The latency of intercepting a system call with `PTRACE_SYSEMU` is 13,592 cycles. In Dune this can be implemented by handling the hardware system call trap directly, with a latency of just 180 cycles. This reveals that most of the Dune *ptrace* benchmark overhead was in fact forwarding the *getpid* system call via a `VMCALL` rather than intercepting the system call.

**trap** indicates the time it takes for a process to get an exception for a page fault. We compare the latency of a SIGSEGV signal in Linux with a hardware-generated page fault in Dune.

**appel1** is a measure of user-level virtual memory management performance. It corresponds to the TRAP, PROT1, and UNPROT test described in [5], where 100 protected pages are accessed, causing faults. Then, in the fault handler, the faulting page is unprotected, and a new page is protected.

**appel2** is another measure of user-level virtual memory management performance. It corresponds to the PROTN, TRAP, and UNPROT test described in [5], where 100 pages are protected. Then each is accessed, with the fault handler unprotecting the faulting page.

## 6.3 Application Performance

### 6.3.1 Sandbox

We evaluated the performance of our sandbox by running two types of workloads. First, we tested compute performance by running SPEC2000. Second, we tested IO performance by running lighttpd. The null sandbox policy was used in both cases.

Figure 3 shows the performance of SPEC2000. In general, the sandbox had very low overhead, averaging only 2.9% percent slower than Linux. However, the *mcf* and *ammp* benchmarks were outliers, with 20.9% and 10.1% slowdowns respectively. This deviation in performance can be explained by EPT overhead, as we observed a high TLB miss rate. We also measured SPEC2000 in VMware Player, and, as expected, EPT overhead resulted in very similar drops in performance.

We then adjusted the sandbox to avoid EPT overhead by backing large memory allocations with 2MB large

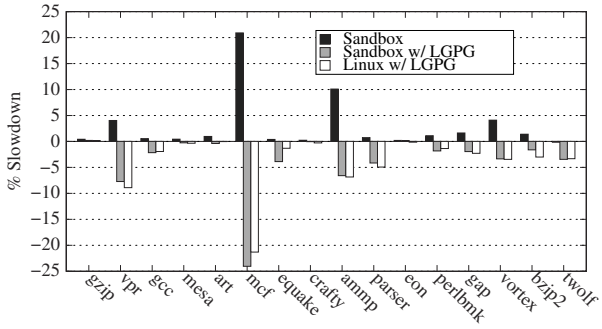



Figure 3: Average SPEC2000 slowdown compared to Linux for the sandbox, the sandbox using large pages, and Linux using large pages identically.

| | 1 client | 100 clients |
|---|---|---|
| **Linux** | 2,236 | 24,609 |
| **Dune sandbox** | 2,206 | 24,255 |
| **VMware Player** | 734 | 5,763 |

Table 4: Lighttpd performance (in requests per second).

pages, both in the EPT and the user page table. Supporting this optimization was straightforward because we were able to intercept *mmap* calls and transparently modify them to use large pages. Such an approach does not cause much memory fragmentation because large pages are only used selectively. In order to perform a direct comparison, we tested SPEC2000 in a modified Linux environment that allocates large pages in an identical fashion using libhugetlbfs [1]. When large pages were used for both, average performance in the sandbox and Linux was nearly identical (within 0.1%).

Table 4 shows the performance of lighttpd, a single-threaded, single-process, event-based HTTP server. Lighttpd exercises the kernel to a much greater extent than SPEC2000, making frequent system calls and putting load on the network stack. Lighttpd performance was measured over Gigabit Ethernet using the Apache *ab* benchmarking tool. We configured *ab* to repeatedly retrieve a small HTML page over the network with different levels of concurrency: 1 client for measuring latency and 100 clients for measuring throughput.

We found that the sandbox incurred only a slight slowdown, less than 2% for both the latency and throughput test. This slowdown can be explained by Dune's higher system call overhead. Using *strace*, we determined that lighttpd was performing several system calls per connection, causing frequent VMX transitions. However, VMware Player, a conventional VMM, experienced much greater overhead: 67% for the latency test and 77% for the throughput test. Although VMware Player pays VMX transition costs too, the primary reason for the slowdown is that each network request must traverse two network stacks, one in the guest and one in the host.

We also found that the sandbox provides an easily extensible framework that we used to implement checkpointing and our firewall. The checkpointing implementation consisted of approximately 450 SLOC with 50 of those being enhancements to the sandbox loader. Our firewall was around 200 SLOC with half of that being the firewall rules parser.

| | create | ctx switch | http request |
|---|---|---|---|
| **fork** | 81 | 0.49 | 454 |
| **Dune sthread** | 2 | 0.15 | 362 |

Table 5: Wedge benchmarks (times in microseconds).

### 6.3.2 Wedge

Wedge has two main benchmarks: sthread creation and context switch time. These are compared to *fork*, the system call used today to implement privilege separation. As shown in Table 5, sthread creation is faster than *fork* because instead of creating a new process each time, an sthread is reused from a pool and "recycled" by restoring dirty memory and state. Context switch time in sthreads is low because TLB flushes are avoided by using the tagged TLB. In Dune sthreads are created 40× faster than processes and the context switch time is 3× faster. In previous Wedge implementations sthread creation was 12× faster than *fork* with no improvement in context switch time [9]. Dune is faster because it can leverage the tagged TLB and avoid kernel calls to create sthreads. The last column of Table 5 shows an application benchmark of a web server serving a static file on a LAN where each request runs in a newly *fork*ed process or sthread for isolation. Dune sthreads show a 20% improvement here.

The original Wedge implementation consisted of a 700-line kernel module and a 1,300-line library. A userspace-only implementation of Wedge exists, though the authors lamented that POSIX did not offer adequate APIs for memory and system call protection, hence the result was a very complicated 5,000-line implementation [9]. Dune instead exposes the hardware protection features needed for a simple implementation, consisting of only 750 lines of user code.

| | GCBench | LinkedList | HashMap | XML |
|---|---|---|---|---|
| Collections | 542 | 33,971 | 161 | 10 |
| **Memory use (MB)** | | | | |
| Allocation | 938 | 15,257 | 10,352 | 1,753 |
| Heap | 28 | 1,387 | 27 | 1,737 |
| **Execution time (ms)** | | | | |
| Normal | 1,224 | 15,983 | 14,160 | 6,663 |
| Dune | 1,176 | 16,884 | 13,715 | 7,930 |
| Dune TLB | 933 | 14,234 | 11,124 | 7,474 |
| Dune dirty | 888 | 11,760 | 8,391 | 6,675 |

Table 6: Performance numbers of the GC benchmarks.

#### 6.3.3 Garbage Collector

We implemented three different sets of modifications to the Boehm GC. The first is the simplest port possible with no attempt to utilize any advanced features of Dune. This benefits from Dune's fast memory protection and fault handling but suffers from the extra TLB costs. The second version improves the direct port by carefully controlling when the TLB is invalidated. The third version avoids using memory protection altogether, instead it reads the dirty bits directly. The direct port required changing 52 lines, the TLB optimized version 91 lines, and the dirty bit version 82 lines.

To test the performance improvements of these changes we used the following benchmarks:

- GCBench [11]. A microbenchmark written by Hans Boehm and widely used to test garbage collector performance. In essence, it builds a large binary tree.
- Linked List. A microbenchmark that builds increasingly large linked lists of integers, summing each one after it is built.
- Hash Map. A microbenchmark that utilizes the Google sparse hash map library [23] (C version).
- XML Parser. A full application that uses the Mini-XML library [34] to parse a 150MB XML file containing medical publications. It then counts the number of publications each author has using a hash map.

The results for these benchmarks are presented in Table 6. The direct port displays mixed results due to the improvement to memory protection and the fault handler but slowdown of EPT overhead. As soon as we start using more hardware features, we see a clear improvement over the baseline. Other than the XML Parser, the TLB version improves performance between 10.9% and 23.8%, and the dirty bit version between 26.4% and 40.7%.

The XML benchmark is interesting as it shows a slowdown under Dune for all three versions: 19.0%, 12.2% and 0.2% slower for the direct, TLB and dirty version respectively. This appears to be caused by EPT overhead, as the benchmark does not create enough garbage to benefit from the modifications we made to the Boehm GC. This is indicated in Table 6; the total amount of allocation is nearly equal to the maximum heap size. We verified this by modifying the benchmark to instead take a list of XML files, processing each sequentially so that memory would be recycled. We then saw a linear improvement in the Dune versions over the baseline as the number of files was increased. With ten 150MB XML files as input, the dirty bit version of the Boehm GC showed a 12.8% improvement in execution time over the baseline.

## 7 Reflections on Hardware

While developing Dune, we found VT-x to be a surprisingly flexible hardware mechanism. In particular, the fine-grained control provided by the VMCS allowed us to precisely direct how hardware was exposed. However, some hardware changes to VT-x could benefit Dune. One noteworthy area is the EPT, as we encountered both performance overhead and implementation challenges. Hardware modifications have been proposed to mitigate EPT overhead [2, 8]. In addition, modifying the EPT to support the same address width as the regular page table would reduce the complexity of our implementation and improve coverage of the process address space. Further reductions to VM exit and VM entry latency could also benefit Dune. However, we were able to aggressively optimize hypercalls, and VMX transition costs had only a small effect on the performance of the applications we evaluated.

There are a few hardware features that we have not yet exposed, despite the fact that they are available in VT-x and possible to support in Dune. Most seem useful only in special situations. For example, a user program might want to have control over caching in order to prevent information leakage. However, this would only be effective if CPU affinity could be controlled. As another example, access to efficient polling instructions (*i.e.,* `MONITOR` and `MWAIT`) could be useful in reducing power consumption for userspace messaging implementations that perform cache line polling. Finally, exposing access to debug registers could allow user programs to more efficiently set up memory watchpoints.

It may also be useful to provide Dune applications with direct access to IO devices. Many VT-x systems include support for an IOMMU, a device that can be used to make DMA access safe even when it is available to untrusted software. Thus, Dune could be modified to safely expose certain hardware devices. A potential benefit could be reduced IO latency. The availability of SR-IOV makes this

possibility more practical because it allows a single physical device to be partitioned across multiple guests.

Recently, a variety of non-x86 hardware platforms have gained support for hardware-assisted virtualization, including ARM [38], Intel Itanium [25], and IBM Power [24]. ARM is of particular interest because of its prevalence in mobile devices, making the ARM Virtualization Extensions an obvious future target for Dune. ARM's support for virtualization is similar to VT-x in some areas. For example, ARM is capable of exposing direct access to privileged hardware features, including exceptions, virtual memory, and privilege modes. Moreover, ARM provides a System MMU, which is comparable to the EPT. ARM's most significant difference is that it introduces a new deeper privilege mode call *Hyp* that runs underneath the guest kernel. In contrast, VT-x provides separate operating modes for the guest and VMM. Another difference from VT-x is that ARM does not automatically save and restore architectural state when switching between a VMM and a guest. Instead, the VMM is expected to manage state in software, perhaps creating an opportunity for optimization.

# 8 Related Work

There have been several efforts to give applications greater access to hardware. For example, The Exokernel [18] exposes hardware features through a low-level kernel interface that allows applications to manage hardware resources directly. Another approach, adopted by the SPIN project [7], is to permit applications to safely load extensions directly into the kernel. Dune shares many similarities with these approaches because it also tries to give applications greater access to hardware. However, Dune differs because its goal is not extensibility. Rather, Dune provides access to privileged hardware features so that they can be used in concert with the OS instead of a means of modifying or overriding it.

The Fluke project [20] supports a nested process model in software, allowing OSes to be constructed "vertically." Dune complements this approach because it could be used to efficiently support an extra OS layer between the application and the kernel through use of privilege mode hardware. However, the hardware that Dune exposes can only support a single level instead of the multiple levels available in Fluke.

A wide range of strategies have been employed to support sandboxing, such as ptrace [16], dedicated kernel modifications [16, 21, 33], binary translation [19], and binary verification [39]. To our knowledge, Dune is the first system to support sandboxing entirely through user-level access to hardware protection, improving performance and reducing code complexity. For example, Native Client [39] reports an average SPEC2000 overhead of 5% with a worst case performance of 12%—anecdotally, we observed higher overheads on modern microarchitectures. By contrast, we were able to achieve nearly zero average overhead (1.4% worst case) for the same benchmarks in Dune. Our sandbox is similar to Native Client in that it creates a secure subdomain within a process. However, Native Client is more portable than Dune because it does not require virtualization hardware or kernel changes.

Like Dune, some previous work has used hardware virtualization for non-traditional purposes. For example, VT-x has been suggested as a tool for creating rootkits [29] that are challenging to detect. Moreover, IOMMU hardware has been used to safely isolate malicious device drivers by running them in Linux processes [13].

# 9 Conclusion

Dune provides ordinary applications with efficient and safe access to privileged hardware features that are traditionally available only to kernels. It does so by leveraging modern virtualization hardware, which enables direct execution of privileged instructions in unprivileged contexts. Our implementation of Dune for Linux uses Intel's VT-x virtualization architecture and provides application-level access to exceptions, virtual memory, and privilege modes. Our evaluation shows both performance and security benefits to Dune. For instance, we built a sandbox that approaches zero overhead, modified a garbage collector to improve performance by up to 40.7%, and created a privilege separation system with 3× less context switch overhead than without Dune.

In an effort to spur adoption, we have structured Dune as a module that works with unmodified Linux kernels. We hope the applications described in this paper are just the first of many uses people will find for the system. The hardware mechanisms exposed by Dune are at the core of many operating systems innovations; their new accessibility from user-level creates opportunities to deploy novel systems without kernel modifications. Dune is freely available at `http://dune.scs.stanford.edu/`.

# Acknowledgments

We wish to thank our shepherd, Timothy Roscoe, for his guidance and valuable suggestions. We would also like to thank Edouard Bugnion for feedback on several itera-

tions of this paper and for his valuable discussions during the early phases of Dune. Finally, we thank Richard Uhlig, Jacob Leverich, Ben Serebrin, and our anonymous reviewers for suggestions that greatly shaped our paper. This work was funded DARPA CRASH under contract #N66001-10-2-4088 and by a gift from Google. Adam Belay is supported by a Stanford Graduate Fellowship.

# References

[1] Libhugetlbfs. `http://libhugetlbfs.sourceforge.net`, Apr. 2012.

[2] J. Ahn, S. Jin, and J. Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 476–487, 2012.

[3] AMD. *Secure Virtual Machine Architecture Reference Manual*.

[4] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 13–15, 2007.

[5] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Apr. 1991.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, 1995.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

[9] A. Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, 2009.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, 2008.

[11] H. Boehm. GC Bench. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`, Apr. 2012.

[12] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, 1991.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, pages 9–9, 2010.

[14] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, 1997.

[15] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, 2005.

[16] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation*, OSDI'08, pages 339–354, 2008.

[17] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, Orcas Island, Washington, May 1995.

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995.

[19] B. Ford and R. Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 293–306, 2008.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, 1996.

[21] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[22] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[23] Google. sparsehash. `http://code.google.com/p/sparsehash/`, Apr. 2012.

[24] IBM. *Power ISA, Version 2.06 Revision B*.

[25] Intel. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*.

[26] Intel Corporation. Invalid Instruction Erratum Overview. `http://www.intel.com/support/processors/pentium/sb/cs-013151.htm`, Apr. 2012.

[27] K. Kaspersky and A. Chang. Remote Code Execution thorugh Intel CPU Bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[28] H. Kermany and E. Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, 2006.

[29] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, 2006.

[30] A. Kivity. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[31] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA. *IBM Syst. J.*, 30(1):34–51, Feb. 1991.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[33] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, SSYM'03, 2003.

[34] M. Sweet. Mini-XML: Lightweight XML Library. `http://www.minixml.org/`, Apr. 2012.

[35] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[36] G. Tene, B. Iyengar, and M. Wolf. C4: the Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, 2011.

[37] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48 – 56, May 2005.

[38] P. Varanasi and G. Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, 2011.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, 2009.

[40] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 13–24, 1987.

# Performance Isolation and Fairness for Multi-Tenant Cloud Storage

David Shue*, Michael J. Freedman*, and Anees Shaikh†
*Princeton University, †IBM TJ Watson Research Center

## Abstract

Shared storage services enjoy wide adoption in commercial clouds. But most systems today provide weak performance isolation and fairness between tenants, if at all. Misbehaving or high-demand tenants can overload the shared service and disrupt other well-behaved tenants, leading to unpredictable performance and violating SLAs.

This paper presents Pisces, a system for achieving datacenter-wide *per-tenant* performance isolation and fairness in shared key-value storage. Today's approaches for multi-tenant resource allocation are based either on per-VM allocations or hard rate limits that assume uniform workloads to achieve high utilization. Pisces achieves per-tenant weighted fair shares (or minimal rates) of the aggregate resources of the shared service, even when different tenants' partitions are co-located and when demand for different partitions is skewed, time-varying, or bottlenecked by different server resources. Pisces does so by decomposing the fair sharing problem into a combination of four complementary mechanisms—partition placement, weight allocation, replica selection, and weighted fair queuing—that operate on different time-scales and combine to provide system-wide max-min fairness.

An evaluation of our Pisces storage prototype achieves nearly ideal (0.99 Min-Max Ratio) weighted fair sharing, strong performance isolation, and robustness to skew and shifts in tenant demand. These properties are achieved with minimal overhead (<3%), even when running at high utilization (more than 400,000 requests/second/server for 10B requests).

## 1. Introduction

An increasing number and variety of enterprises are moving workloads to cloud platforms. Whether serving external customers or internal business units, cloud platforms typically allow multiple users, or *tenants*, to share the same physical server and network infrastructure, as well as use common platform services. Examples of these shared, multi-tenant services include key-value stores, block storage volumes, SQL databases, message queues, and notification services. These leverage the expertise of the cloud provider in building, managing, and improving common services, and enable the statistical multiplexing of resources between tenants for higher utilization.

Because they rely on shared infrastructure, however, these services face two key, related issues:

- ***Multi-tenant interference and unfairness:*** Tenants simultaneously accessing shared services contend for resources and degrade performance.
- ***Variable and unpredictable performance:*** Tenants often experience significant performance variations, e.g., in response time or throughput, even when they can achieve their desired mean rate [8, 16, 33, 35].

These issues limit the types of applications that can migrate to multi-tenant clouds and leverage shared services. They also inhibit cloud providers from offering differentiated service levels, in which some tenants can pay for performance isolation and predictability, while others choose standard "best-effort" behavior.

Shared back-end storage services face different challenges than sharing server resources at the virtual machine (VM) level. These stores divide tenant workloads into disjoint partitions, which are then distributed (and replicated) across different service instances. Rather than managing individual storage partitions, cloud tenants want to treat the entire storage system as a single black box, in which *aggregate* storage capacity and request rates can be elastically scaled on demand. Resource contention arises when tenants' partitions are co-located, and the degree of resource sharing between tenants may be significant higher and more fluid than with coarse VM resource allocation. Particularly, as tenants may use only a small fraction of a server's throughput and capacity,[1] restricting nodes to a few tenants may leave them highly underutilized.

To improve predictability for shared storage systems with a high degree of resource sharing and contention, we target global ***max-min fairness*** with ***high utilization***. Under max-min fairness, no tenant can gain an unfair advantage over another when the system is loaded, i.e., each tenant will receive its weighted fair share. Moreover, given its work-conserving nature, when some tenants use less than their full share, unconsumed resources are divided among the rest to ensure high utilization. While our mechanisms may be applicable to a range of services with shared-nothing architectures [31], we focus our design and evaluation on a replicated key-value storage service, which we call *Pisces* (*P*redictable *S*hared *C*loud *S*torage).

[1] Indeed, at today's Amazon S3 prices, a single server handling 50,000 GET reqs/second would cost $180/hour in request pricing alone.

Providing fair resource allocation and isolation at the *service* level is confounded by variable demand to different service partitions. Even if tenant objects are uniformly distributed across their partitions, per-object demand is often skewed, both in terms of request rate and size of the corresponding (read or write) operations. Moreover, different request workloads may stress different server resources (e.g., small requests may be interrupt limited, while large requests are bandwidth limited). In short, simply assuming that each tenant requires the same proportion of resources per partition can lead to unfairness and inefficiency. To address these issues, Pisces makes a number of contributions:

**(1) Global fairness.** To our knowledge, Pisces is the first system to provide per-tenant fair resource sharing across all service instances. Further, as total system capacity is allocated to tenants based on their normalized weights, such a max-min fair system can also provide minimal performance guarantees given sufficient provisioning. In comparison, recent commercial systems that offer request rate guarantees (i.e., Amazon DynamoDB [1] do not provide fairness, assume uniform load distributions across tenant partitions, and are not work conserving.

**(2) Novel mechanism decomposition.** Pisces introduces a clean decomposition of the global fairness problem into four mechanisms. While operating on different timescales and with different levels of system-wide visibility, these mechanisms complement one another to ensure fairness under resource contention and variable demand.

*(i) Partition Placement* ensures a fair allocation by (re)-assigning tenant partitions to nodes (long timescale).

*(ii) Weight Allocation* distributes overall tenant fair shares across the system by adjusting local per-tenant weights at each node (medium timescale).

*(iii) Replica Selection* load-balances requests between partition replicas in a weight-sensitive manner (real-time).

*(iv) Weighted Fair Queuing* at service nodes enforces performance isolation and fairness according to the local tenant weights (real-time).

**(3) Novel algorithms.** We introduce several novel algorithms to implement Pisces's mechanisms. These include a reciprocal swapping algorithm for weight allocation that shifts weights when tenant demand for local resources exceed their local share, while maintaining global fairness. We use a novel application of optimization-inspired congestion control for replica selection, which complements weight allocation by distributing load over partition replicas in response to per-node latencies. Finally, to manage different resource bottlenecks on contended nodes, we enforce *dominant resource fairness* [9] between tenants at the node level, while providing max-min fairness at the service level. To do so, we extend traditional deficit-weighted round robin queuing to handle per-tenant multi-resource scheduling.

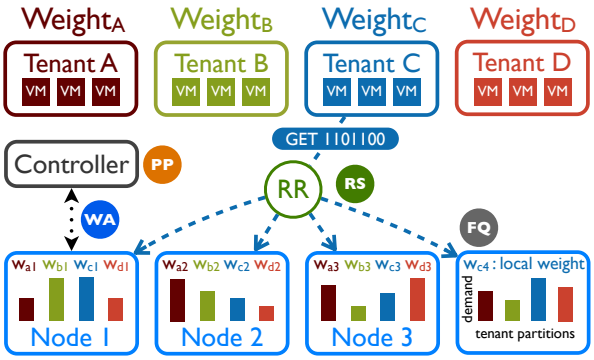



**Figure 1: Pisces multi-tenant storage architecture.**

**(4) Low overhead and high utilization.** Pisces is designed to support high server utilization, both high request rates (100,000s requests per second, per server) and full bandwidth usage (Gbps per server). To do so, its mechanisms must apply at real time without inducing significant throughput degradation. Through careful system design, our prototype achieves <3% overhead for 1KB requests and actually outperforms the unmodified, non-fair version for small requests. Commercial systems like DynamoDB, on the other hand, typically target lower rates (e.g., more than 10,000 reqs/s requires special arrangements [2]).

Through an extensive experimental evaluation, we demonstrate that Pisces significantly improves the multi-tenant fairness and isolation properties of our key-value store, built on Membase [3], across a range of tenant workloads. We also show that its replica selection and rebalancing policies optimize system performance, even as workload patterns shift dynamically. While this paper frames the partition placement problem and implements a simple greedy placement algorithm for our evaluation, we do not fully explore and evaluate its design space.

## 2. Architecture and Design

We consider multi-tenant cloud services with partitioned workloads (i.e., data sets), where each partition is disjoint but may be replicated on different service nodes. Client requests are routed to the appropriate node based on the partition mapping, and the replica selection policy in use. Ultimately, request arbitration for fairness and isolation between tenants occurs at the service nodes.

Figure 1 shows the high-level architecture of Pisces, a key-value storage service that provides system-wide, per-tenant fairness and isolation. Pisces provides the semantics of a persistent map between opaque keys (bit-strings) and unstructured data values (binary blobs) and supports simple key lookups (get), modifications (set), and removals (delete). To partition the workload, the keys are first hashed into a fixed-size key space, which is then subdivided into disjoint segments.

Pisces enforces per-tenant fairness at the system-wide level. As shown in Figure 1, each tenant $t$ is given a

single, global weight $w_t$ that determines its fair share of overall system resources (i.e., throughput). These weights are generally set according to the tenant's service-level objective (SLO). To support service models with rate guarantees, the provider can simply convert a specified rate into a corresponding system resource proportion (weight) given the current capacity. The service provider can also adjust the tenant weights, e.g., in response to new tenant requirements or changes in system capacity.

Pisces allows a cloud service provider to offer a flexible service model, in which customers pay for their consumed storage capacity, with an optional additional tiered charge for an "assured rate" service. Assured service users can reserve a minimum service throughput—which, when normalized, translates to a minimum fair share of the global service throughput—with the price dependent upon this rate. The system ensures this minimum rate, yet also allows users to exploit unused capacity (perhaps while charging an additional "overage" fee). Such multi-tiered charging is common in many Internet contexts, e.g., network transit and CDNs often use burstable billing and charge differently for rate commitments and overages.

While Pisces's general mechanisms should extend to other shared storage systems, our current prototype makes some simplifying assumptions. It does not support more advanced queries, such as scans over keys. It is designed primarily to serve keys out of an in-memory cache for high throughput, and only asynchronously writes data to disk (much like Masstree [19] and the MyISAM storage engine in MySQL). We do not focus on consistency issues, and assume that a separate protocol keeps the partition replicas in sync.[2] Further, we assume a well-provisioned network (e.g., one with full bisection bandwidth [5]); we do not deal explicitly with in-network resource sharing and contention, which has been considered by complementary work [26, 27, 28]. Finally, we assume a reasonably stable tenant workload distribution. While Pisces can support highly-skewed demand distributions across partitions and can handle short-term fluctuations in demand for particular keys, we assume that the relative popularity of entire partitions shifts relatively slowly (e.g., on the order of minutes). This provides the system with sufficient time to rebalance partition weights when needed.

## 2.1 Life of a Pisces Request

Before a tenant can read (get) and write (set) data in the system, a central controller first performs *partition placement (PP)* to assign its data partitions (and their replicas) to service nodes. Each tenant has its own key space, but partitions from different tenants may be co-located on the same service node. The controller then disseminates the partition mapping information to each of the request routers. These request routers can be implemented in client libraries running on the tenants' (virtual) machines, or deployed on intermediate machines (as illustrated in Figure 1). The controller also translates each tenant's global fair-share into local shares at individual storage nodes through *weight allocation (WA)*.

When a client tenant, *C*, issues a request to Pisces, the request router dispatches the request based on its key (e.g., 1101100) and partition location to an appropriate server. If enabled, *replica selection (RS)* allows the request router to flexibly choose which replica to use (e.g., Nodes 1 or 4). Otherwise, the router directs the request to the primary partition. Once the replica has been selected, the router adds the request to a windowed queue of outstanding operations—one queue per server, per tenant.

Since partitions from multiple tenants may reside on this node, the tenants' requests will contend for resources. To enforce fairness and isolation, the service node applies *fair queuing (FQ)* to schedule tenant requests according to each tenant's local weight ($w_{c,4}$). When a request reaches the server, the server adds the request to a queue specific to that tenant (*C*). Every "round" of execution, the server allocates tokens to each tenant according to its local weight ($w_{c,4}$), which it then consumes when processing requests from the tenant queues. If the request consumes more than the allocated resources, it must complete on a subsequent round after tenant *C*'s tokens have been refilled. This guarantees that each tenant will receive its local fair share in a given round of work, if multiple tenants are active. Otherwise, tenants can consume excess resources left idle by the others without penalty.

## 2.2 System-wide Fair Sharing: Example

The challenge of achieving system-wide fairness can be illustrated with a few scenarios, as shown in Figure 2. From these, we derive design lessons for how Pisces should (1) place partitions to enable a fair allocation of resources, (2) allocate local weights to maintain global fairness, (3) select replicas to achieve high utilization, and (4) queue requests to enforce fairness. In the examples, two tenants (*A* and *B*) with equal global shares access two Pisces nodes with equal capacity (100 kreq/s). Each tenant should receive the same aggregate share of 100 kreq/s.

**Partitions should be placed with respect to demand and node capacity constraints.** For per-tenant fairness to be *feasible*, there must exist some assignment of tenant partitions that can satisfy the global tenant shares without violating node capacities. Not all placements lead to a feasible solution, however, as shown in Figure 2a. Here, each tenant has the same skewed distribution of partition demand: 40, 30, 20, and 10. Arbitrary partition assignment can easily lead to capacity overflow: with *A* and *B* both demanding 60 kreq/s for the partitions on the

[2] Our implementation is built on Membase [3], which asynchronously replicates from a partition's primary copy to its backup(s), and by default reads only from the primary for strong consistency.

first node, each tenant receives 10 kreq/s less than their global share. If we take partition demand into account and shuffle tenant *B*'s partitions between the nodes, then we can achieve a fair and feasible placement. Although the skew in this example may be extreme, Internet workloads often exhibit a power-law (or Zipf) distribution across keys, which can induce skewed partition demand.

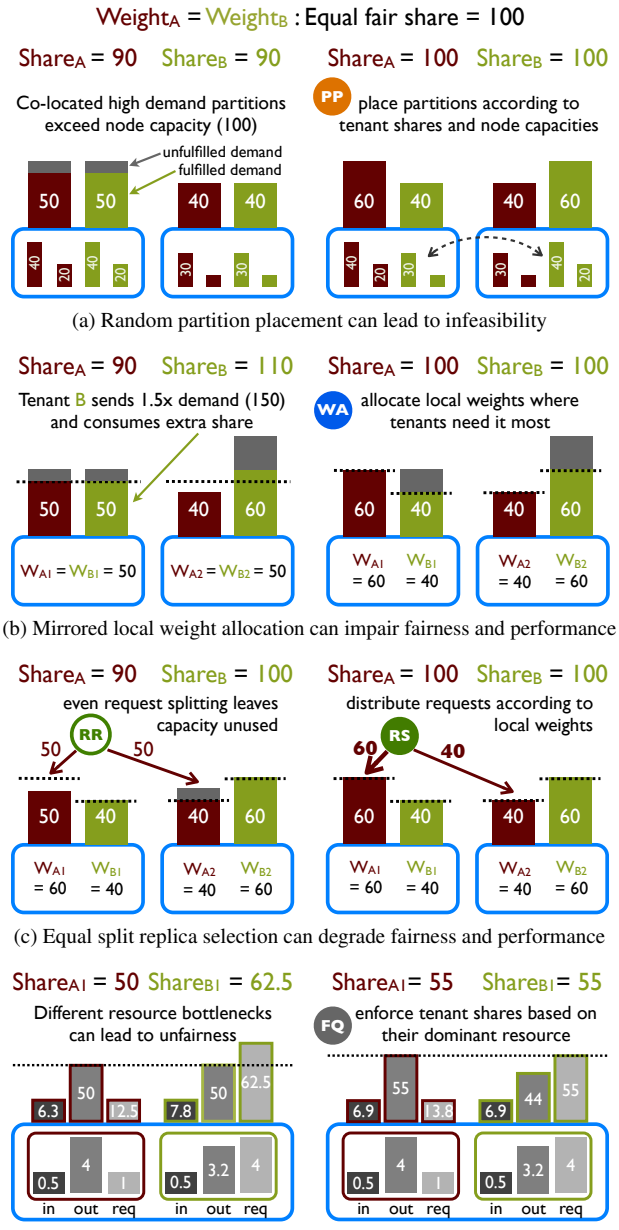
**Local weights should give tenants throughput where they need it most.** Even with a feasible fair partition placement, if the local weights simply mirror the global (uniform: 50 each) weights, as in Figure 2b, global fairness may still suffer. Although fair queuing allows tenant *B* to consume more than its local share (60 > 50) at the first node when *A* consumes less (40 < 50), if tenant *A* increases its demand, it will consume its remaining local allocation. This will increase its global share and eat into tenant *B*'s global share. However, if we adjust each tenant's local weights to match their demand (60/40 and 40/60), we can preserve fairness even under excess load.

**Replicas should be selected in a weight-sensitive manner.** When replica selection is enabled, the fairness problem becomes easier, in general, since each replica only receives a *fraction* of the original demand. By spreading partition demand across multiple servers, replica selection produces smoother distributions that makes partitions easier to place. Further, once placed, the reduced per-replica demand is easier to match with local weights.

However, the replica selection policy must be carefully tuned, otherwise fairness and utilization may still diverge from the system-wide goals. In Figure 2c, tenant *A* can send requests to replicas on either node. However, since its local weights are skewed to match the resident partition demand, simple replica-selection policies are insufficient to exploit the variation between the local weights. For example, equal-split round robin would lead to a 10 kreq/s drop in *A*'s global share. Instead, by adjusting per-tenant replica selection proportions to reflect the local weights, we can fully exploit replicated reads for both improving performance and facilitating fairness.

**Request queuing should enforce dominant resource fairness.** Up to this point, our examples have illustrated multiple tenants with identical weights competing for identical request rates, which implicitly assumes that all requests have equivalent cost. In practice, requests may be of different input or output sizes, and can activate different bottlenecks in the system (e.g., small requests may be bottlenecked by server interrupts, while large requests may be bottlenecked by network bandwidth). Thus, each tenant's workload may vary accordingly across the different resources, as seen in Figure 2d.

Each tenant's resource profile shows the relative proportion of each resource—bytes in, bytes out, and number of requests—that the tenant consumes. These resources are the likely limiting factor for writes, reads, and small




(a) Random partition placement can lead to infeasibility

(b) Mirrored local weight allocation can impair fairness and performance

(c) Equal split replica selection can degrade fairness and performance

(d) Single-resource queuing can violate fairness

**Figure 2: Illustrating the difficulty in achieving system-wide fairness. Both tenants have the same global weight (equal fair share); *Share* is the normalized rate actually achieved by each tenant. Left-hand figures correspond to settings lacking Pisces's mechanisms; right-hand figures apply its techniques.**

requests, correspondingly. Tenant *A* is read bandwidth bound, consuming 4% of the out bandwidth for every 1% of request capacity it uses. Tenant *B*, on the other hand, is interrupt-bound for its smaller reads, consuming more request resources (4%) than out bandwidth (3.2%). Applying fair queuing to a single resource (bytes out) gives each tenant a fair share of 50%, but also allows

tenant $B$ to receive a larger share (62.5%) of its dominant resource (requests). Instead, using *dominant resource fairness* (DRF) [9] ensures that each tenant will receive a fair share of its dominant resource relative to other tenants: tenant $A$ receives 55% of out bytes and tenant $B$ receives 55% of requests, while out bytes remains the bottleneck.

# 3. Pisces Algorithms

Pisces implements four complementary mechanisms according to the design lessons discussed above. Each mechanism operates on different parts of the system at different timescales. Together, they deliver system-wide fair service allocations with minimal interference to each tenant.

## 3.1 Partition Placement

The partition placement mechanism ensures the feasibility of the system-wide fair shares. It assigns partitions so that the load on each node (the aggregate of the tenants' per-partition fair-share demands) does not exceed the node's rate capacity. Since our prototype currently only implements a simple greedy placement scheme, here we only outline the algorithm without providing details. The centralized controller first collects the request rate for each tenant partition, as measured at each server. It then computes the partition demand proportions by normalizing the rates. Scaling each tenant's global fair share by these partition proportions determines the per-partition demand share that each tenant should receive. Next, the controller supplies the demand and node capacity constraints into a bin-packing solver to compute a new partition assignment relative to the existing one. Finally, the controller migrates any newly (re)assigned partitions and updates the mapping tables in its request router(s).

Partition placement typically runs on a long timescale (every few minutes or hours), since we assume that tenant demand distributions (proportions) are relatively stable, though demand intensity (load) can fluctuate more frequently. However, it can also be executed in response to large-scale demand shifts, severe fairness violations, or the addition or removal of tenants or service nodes. While the bin-packing problem is NP-hard, simple greedy heuristics may suffice in many settings, albeit not achieve as high a utilization. After all, to achieve fairness, we only need to find a feasible solution, not necessarily an optimal one. Further, many efficient approximation techniques can find near-optimal solutions [22, 29]. We intend to further explore partition placement in future work.

## 3.2 Weight Allocation

Once the tenant partitions are properly placed, weight allocation iteratively adjusts the local tenant shares ($R$) to match the demand distributions. As sketched in Algorithm 1, the weight-allocation algorithm on the controller (i) detects tenant demand-share mismatch and (ii) decides which tenant share(s) (weights) to adjust and by what amount, even while it (iii) maintains the global fair share. A key insight is that any adjustment to tenant shares requires a *reciprocal swap*: if tenant $t$ takes some rate capacity from a tenant $u$ on some server, $t$ must give the same capacity back to $u$ on a different server. This swap allows the tenants to maintain the same aggregate shares even while local shares change. Each tenant's local weight is initially set to its global weight, $w_t$, and then adapts over time. To adapt to distribution shifts yet allow replica selection to adjust to the current allocation, weight allocation runs in the medium timescale (seconds).

**Detecting mismatch:** While it is difficult to directly measure tenant demand, the central controller can monitor the rate each tenant receives at the service nodes. Weight allocation uses this information ($D$), together with the allocated shares ($R$), to approximate the demand-share mismatch that each tenant experiences on each node. We tried both a latency-based cost function using the M/M/1 queuing model of request latency—i.e., $l_n^t = 1 \,/\, (R_n^t - D_n^t)$ for tenant $t$ on node $n$—as well as a more direct rate-based cost. Unfortunately, in a work-conserving system, it can be difficult to determine how much rate $R$ was actually allocated to $t$, as it can vary depending on others' demand. For example, if a tenant $t$ has weight $w_n^t = \frac{1}{4}$ on a node with capacity $c_n$, then its local rate under load is just $\hat{R}_n^t = \frac{1}{4}c_n$, even though $R_n^t \geq \frac{1}{4}c_n$ if $t$ has excess demand and other tenants are not fully using their shares. Instead, by using the difference between the consumed rate and the configured local share, $e_n^t = \left| D_n^t - \hat{R}_n^t \right|$, we can largely ignore the variable allocation and instead focus on the tenant's desired rate under full load (i.e., $\hat{R}$). Fortunately, the allocation algorithm can easily accommodate any convex cost function to approximate demand mismatch.

**Determining swap:** Since the primary goal of Pisces is fairness, weight allocation seeks to minimize the *maximum* demand-share mismatch (or cost). However, giving additional rate capacity to the tenant $t$ that suffers maximal latency necessarily means taking away capacity from another tenant $u$ at the same node $n$. If too large, this rate (weight) *swap* may cause $u$'s cost to exceed the original maximum. To ensure a valid rate swap, the algorithm uses the linear bisection for latency, $\text{take}(t, u, n) = ((R_n^u - D_n^u) - (R_n^t - D_n^t)) \,/\, 2$, or the min of the differences for rate: $\min(e_n^t, e_n^u)$.

**Maintaining fair share:** Before committing to a final swap, weight allocation must first find a reciprocal swap to maintain the global fair share: if tenant $t$ takes from tenant $u$ at node $n$, then it must reciprocate at a different node. Given a reciprocal node $m$, the controller computes the rate swap as the minimum of the take and give swaps, $\text{swap} = \min(\text{take}(t, u, n), \text{give}(u, t, m))$, and translates the rates into the corresponding local weight settings.

**Algorithm 1** Weight Allocation: medium timescale (s)

$T$: tenants, $N$: nodes, $W$: global tenant weights,
$R$: local tenant resource share

**function** CONTROLLER.ALLOCATEWEIGHTS($T, N, W, R$)
  $D \leftarrow$ *monitor_node_rates*($T, N$)
  $C \leftarrow$ *compute_cost_estimates*($D, R$)
  $R \leftarrow$ *compute_weight_swap*(max($C$))
  *reassign_weight_allocations*($R$)

**Algorithm 2** Replica Selection: real-time (ms)

$j$: request/response, $t$: tenant, $M$: partition mappings
$n$: node, $q^t$: per-tenant request queue

**function** REQUESTROUTER.SENDREQUEST($j, t, M$)
  $p \leftarrow$ *get_partition_of*($j$)
  **for** $n \in M[t, p]$ **do**
    **if** window $w_n^t >$ outstanding $s_n^t$ **then**
      *send_request*($j, n$)
      $s_n^t \leftarrow s_n^t + 1$ **return**
  **if not** sent **then** *queue_request*($j, q^t$)

**function** REQUESTROUTER.RECVRESPONSE($j, t, n$)
  $l_{\text{resp},n} \leftarrow$ *latency_of_response*($j$)
  $s_n^t \leftarrow s_n^t - 1$
  *update_window*($w_n^t, l_{\text{resp},n}$)
  SendRequest(*dequeue_request*($q^t$), $t, M$)

**Algorithm 3** Fair Queuing: real-time ($\mu$s)

$j$: request, $t$: tenant, $r$: round, $R$: local tenant share

**function** SERVICENODE.QUEUEREQUESTS($R$)
  **while** $j, t \leftarrow$ *dequeue_request*() **do**
    **if** $t$.state = inactive **then**
      $t$.state $\leftarrow$ active
      *allocate_tenant_tokens*($R$)
    **if** tokens available for $t$ **then**
      *consume_request_resources*($j, t$)
      **if** $j$ unfinished **then**
        *queue_request*($j$)
      **else**
        **if** resources left for $j$ **then**
          *refund_unused_resources*($j$,$t$)
        **if** no requests left for $t$ **then**
          $t$.state $\leftarrow$ inactive
    **else**
      *queue_exhausted_request*($j, t$)
      $t$.state $\leftarrow$ exhausted
  **if** $\exists\, t \in T$ such that $t$.state = exhausted **then**
    $r \leftarrow r + 1$ (increment round)
    *allocate_tenant_tokens*($R$)

## 3.3 Replica Selection

To maintain fairness while balancing load, request routers distribute tenant requests to partition replicas in proportion to their local shares (i.e., normalized weights). While the controller (or alternatively, each server) could disseminate the local share information to each request router, Pisces avoids the need for explicit updates by exploiting implicit feedback. Explicit updates could be prohibitively expensive for a system with tens of thousands of tenants, request routers, and service nodes.

As delineated in Algorithm 2, when a request router (or client) sends a request, it round-robins between partition replicas (nodes), consuming slots from their respective request windows. Once the windows fill up, the request router locally queues requests until server responses free additional window slots. Due to per-tenant fair queuing at the nodes, requests sent to nodes with a larger tenant share experience lower queuing delay than nodes with a smaller share. The request router uses the relative response latency between replicas as a proxy for the size differential between the local rate allocations. It thus adjusts the request windows according to the FAST-TCP [34] update:

$$w(m+1)_n^t \;=\; (1-\alpha)\cdot w(m)_n^t \;+\; \alpha\cdot\left(\frac{l_{\text{base}}}{l_{\text{est}}}\right)$$

Each iteration of the algorithm adjusts the window based on how close the tenant demand is to its local rate allocation, which is represented by the ratio of a desired average request latency, $l_{\text{base}}$, to the smoothed (EWMA) latency estimate, $l_{\text{est}}$. The $\alpha$ parameter limits the window step size. Thus, each request router makes proper adjustments in a fully decentralized fashion: it only uses local request latency measurements to compute the replica proportions. The convergence and stability guarantees for this approach follow from those given by FAST-TCP.

Because replica selection balances a tenant's demand distribution in real-time, the per-tenant demand at each service node equilibrates before the next iteration of the weight-allocation algorithm. Weight allocation then attempts to match the local tenant shares (normalized weights) to the new demand distribution. The convergence and stability of this interlocking coordinate-ascent algorithm arises from the convex nature of the problem.

## 3.4 Fair Queuing

Ultimately, system-wide fairness and isolation comes down to mediating resource contention between tenants at the individual storage nodes. To implement fair queuing, Pisces uses the deficit (weighted) round robin [30] (DWRR) scheduling discipline for its simplicity, low time complexity, and bounded deviation from the ideal Generalized Processor Sharing model. In DWRR, the basic unit of work is called a token, which represents a normalized request or quantum of work. Pisces applies Dominant Resource Fairness, as mentioned in Section 2.2, to translate a token into a tenant-specific resource allocation vector. Currently, our implementation accounts for the number of bytes received, bytes sent, and requests (the latter for request-bound workloads). The queuing algorithm can also support additional resources like disk IOPs, which we intend to explore in the future. Multiple tokens may be needed to serve a large request, or a single token's resources may span several small requests. In any given

round of request processing, each tenant can consume up to its weighted share of the total fixed number of available tokens. By bounding the number of tokens per round, the scheduler can ensure fairness within a definite timeframe.

Request processing in DWRR proceeds in rounds. Per Algorithm 3, on each scheduling round, the scheduler allocates tokens to each active tenant (those with queued requests) in proportion to their local weight. As it processes requests, the scheduler consumes resources from the token resource allocation. If a request requires additional resources, the scheduler adds it back on the request queue to mitigate head-of-line blocking. Otherwise, it refunds the tenant with any unconsumed resources. If a tenant runs out of tokens, the scheduler adds its outstanding requests to an exhausted queue. The scheduler advances the round and refreshes tokens only when every tenant is either inactive (no work) or exhausted (work but no tokens) and there is work to do in the next round.

To compute the proper Dominant Resource Fair (DRF) shares, the scheduler on each node tracks the resource consumption of each tenant. Periodically (every half second in our prototype), it recomputes the resource allocation. First, the scheduler determines each tenant's resource utilization (e.g., $U^t_{\text{bytes-out}} = \frac{\text{bytes-out}^t}{\text{bytes-out}^{\text{cap}}}$) and its dominant resource ($U^t_{\text{dom}} = \max_i(U^t_i)$), using the latter to normalize each utilization. The scheduler computes the limiting DRF allocation by finding the minimum of the inverse of the weighted utilization sums: $\min_i(\frac{1}{\sum_i w^t U^t_i})$. Any excess resources are distributed equally among all tenants.

Despite the fact that Pisces only enforces DRF on a per-node level, it still provides global max-min fair shares for each tenant. When two tenants have different dominant resources at a node, DRF allocates each tenant a larger local share than it would have received if the tenants had contended for the same resource. In other words, each tenant's share of its dominant resource is lower bounded by the max-min fair share of a single common resource. Thus, even if a tenant's dominant resource varies from node to node, its aggregate share will still equal or exceed its max-min fair share (proportion) of total system resources. This allows Pisces to use a single weight to represent the per-tenant global and local shares, rather than a more complicated weight vector.

# 4. Pisces Optimizations

Pisces's algorithms and its four part decomposition can be conceptually derived as a distributed optimization problem [24]. In designing Pisces, we consider a multi-timescale decomposition of the fair-sharing problem into the corresponding Pisces mechanisms. Such a formulation allows us to craft the Pisces algorithms in a principled way and to make assertions about their feasibility, optimality, stability, and convergence under the standard assumptions of convex optimization. In the remainder of this section, we discuss additional design and implementation considerations in the Pisces weight allocation and fair queuing mechanisms.

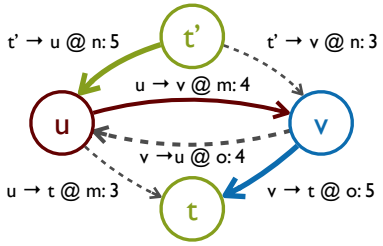



**Figure 3: Reciprocal weight exchange modeled as a Maximum Bottleneck Flow problem.**

## 4.1 Finding Multilateral Weight Swaps

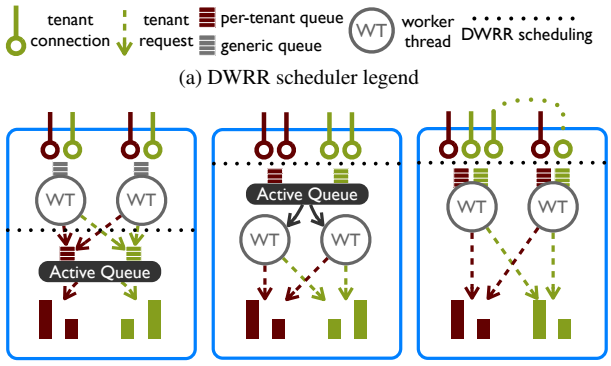
While a bilateral exchange between two tenants (as described in Section 3.2) may suffice, a multilateral exchange may optimize local shares even further. We model this exchange as a path through a flow graph, as shown in Figure 3. Nodes in the graph represent tenants and each directed edge represents a possible swap with capacity equal to the swap rate (e.g., tenant $u$ can take rate 4 from tenant $v$ at server $m$). The swap rate must be positive, and it is computed as the maximum over all server nodes where the edge's tenants have co-located partitions.

The max latency tenant $t$ is modeled as both the source $t'$ and sink of the flow graph, as it must first take rate to minimize its cost (latency or rate-distance) and then reciprocate to maintain the global fair-share invariant. In the example, both bilateral exchanges ($t' \rightarrow u \rightarrow t$ and $t' \rightarrow v \rightarrow t$) are bottlenecked by the edge with smallest capacity (i.e., 3). Instead, weight allocation should choose the multi-hop path ($t' \rightarrow u \rightarrow v \rightarrow t$) with bottleneck 4 that corresponds to the Maximum Bottleneck Flow (MBF). The MBF not only minimizes the max cost for $t$ by the greatest extent, but also reduces the cost for $u$ and $v$.

On each iteration of the weight allocation loop in Algorithm 1, the controller constructs the flow graph from the collected node latency data and solves the MBF problem using a variant of Dijkstra's shortest path algorithm. Then, just as with bilateral swaps, the algorithm converts the rates into weights and sets the local tenant shares accordingly. To avoid oscillations around the optimal point, both the tenant latencies and the swap rate must exceed minimal thresholds, in order to ensure that the weight adjustment results in the desired rate change.

## 4.2 Getting Fair Queuing Right

**Enforce queuing at the appropriate software layer.** Implementing the server DWRR scheduler may seem straightforward at first, but it presents an engineering challenge to do so with low overhead. The most natural approach is simply to place tenant request queues right after request processing in the application, as in Figure 4b.




(a) DWRR scheduler legend

(b) Per-request DWRR (c) Per-connection (d) Non-blocking

**Figure 4: Scheduling per-connection (c) instead of per-request (b) achieves fairness. Decoupling threads and work-stealing (d) optimizes performance.**

The problem, however, is that resources have already been consumed (to receive bytes and parse the request), which prevents the scheduler from enforcing fairness and isolation for certain network I/O bound workloads. Thus, the Pisces scheduler instead operates prior to application request handling, as in Figure 4c, and mediates between *connections* before any resources are consumed. Now, however, the scheduler no longer knows how much work remains in each connection queue. To handle this uncertainty, the DWRR scheduler allocates a fixed number of tokens from the tenant's token pool when a connection is dequeued for processing. As in Algorithm 3, any unused tokens and resources are *refunded* back to the tenant.

**Avoid queue locking at all costs.** Even with the scheduler in the right place, we still need to worry about the queue's implementation and efficiency. Maintaining a centralized active queue—per-connection DWRR in Figure 4c—is a natural design point for fair-queuing. A single thread enqueues connections, and separate worker threads pull tenant connections off this queue one-at-a-time, servicing them by consuming token resources. While simple and fair, this design is flawed: whenever the active queue is empty, the worker threads must wait on a conditional lock. We found that the overhead of this conditional waiting and waking can reduce the processing of small requests by over 30%. To combat this overhead, Pisces uses a combination of non-blocking per-tenant connection queues [21] and a distributed form of DWRR [18]. In this scheme, each worker thread handles its own set of tenant connections by sub-allocating tenant tokens for local use. If a worker thread's connections have either quiesced or run out of tokens, it tries to steal work or tokens from the other threads. If nothing can be stolen, then the worker thread can safely advance the round, since all tenants are either inactive or exhausted.

**Eliminate intermediary queuing effects.** While this non-blocking design (Figure 4d) is highly efficient, it faces one final barrier to max-min fairness: our measurements showed good performance for small requests bottlenecked by server interrupts, but poor fairness for bandwidth-bound workloads. This arises for two reasons. First, if the scheduler does not properly wait for write-blocked connections (EAGAIN) to finish consuming resources before advancing the round, then high-weight, bandwidth-bound tenants could see their remaining tokens wiped out prematurely. Second, over-sized TCP send buffers (128 KB) mask network back pressure. On a 1Gbps link with sub-ms delay, the bandwidth delay product is on the order of tens of kilobytes. When multiple tenant connections (>64) contend for output bandwidth, writes can succeed even when the outbound link is congested, which again causes the scheduler to advance the round too soon. In response, Pisces uses small connection send buffers (6 KB), and the scheduler waits for I/O-bound connections to finish consuming resources before advancing the round. To prevent worker threads from excessive idling due to non-local network congestion, the scheduler uses a short timeout (2 ms) that wakes all I/O quiescent threads and allows the round to advance.

# 5. Prototype on Membase

We implemented Pisces on top of the open-source Membase [3] key-value storage system (part of the Couchbase suite). Built around the popular memcached in-memory caching engine, Membase adds object persistence, data replication, and multi-tenancy. Membase relies heavily on the in-memory key-value cache to serve requests and dispatches disk-bound requests to a background thread. Key-value set or delete operations are committed first in memory and later asynchronously written to disk.

Membase creates even-sized explicit partitions and directly maps the partitions to server nodes. It can replicate and migrate partitions for fault tolerance, and synchronizes primary and secondary replicas in an eventually consistent manner. For evaluation purposes, we replaced Membase's uniform partition-placement mechanism with one based on a simple greedy heuristic, with which we pre-compute a feasible fair placement based on known (oracle) tenant demand distributions.

We integrated Pisces's fairness and isolation mechanisms into Membase using a mix of languages. Implementing the optimized multi-tenant, non-blocking DWRR scheduler in the core server codebase required an extensive overhaul of the connection threading model in addition to adding the scheduling and queuing code in C (3000 LOC). Replica selection was implemented in Java (1300 LOC) and integrated directly in the spymemcached [4] client library. Our centralized controller, which implemented both weight allocation and partition placement, comprised approximately 5000 LOC of Python.

# 6. Evaluation

In our evaluation, we consider how the mechanisms in Pisces build on each other to provide fairness and performance isolation by answering the following questions:

- Are each of the four mechanisms (PP, WA, FQ and RS) necessary to achieve fairness and isolation?
- Can Pisces provide global weighted shares and enforce local dominant resource fairness?
- How well does Pisces handle demand dynamism?

We quantify fairness as the Min-Max Ratio (MMR) of the dominant resource (typically throughput) across all tenants, $\frac{x^{\min}}{x^{\max}}$. This corresponds directly to a max-min notion of fairness.

## 6.1 Experimental Setup

To evaluate Pisces's fairness properties, we setup a testbed comprised of 8 clients, 8 servers, and 1 controller host. Each machine has two 2.4 GHz Intel E5620 quad-core CPUs with GigE interfaces, 12GB of memory, and run Ubuntu 11.04. Pisces server instances are configured with 8 threads and two replicas per partition. All machines are connected directly to a single 1 Gbps top-of-rack switch to provide full bisection bandwidth and avoid network contention effects. Similarly, replica selection and request routing are handled directly in a client-side library to minimize proxy bottleneck effects.

On the clients, we use the Yahoo Cloud Storage Benchmark (YCSB) [7] to generate a Zipf distributed key-value request workload ($\alpha$ = 0.99). Each client machine runs multiple YCSB instances, one for each tenant, to mimic a virtualized environment while avoiding the overhead of virtualized networking. Each tenant is pre-loaded with a fully cached data set of 100,000 objects which are hashed over its key space and divided into 1024 partitions. Object sizes are set to 1kB unless otherwise noted. Tenant request workloads include read-only (all GET), read-heavy (90% GET, 10% SET) and write-heavy (50% GET, 50% SET). All clients send their requests over TCP.

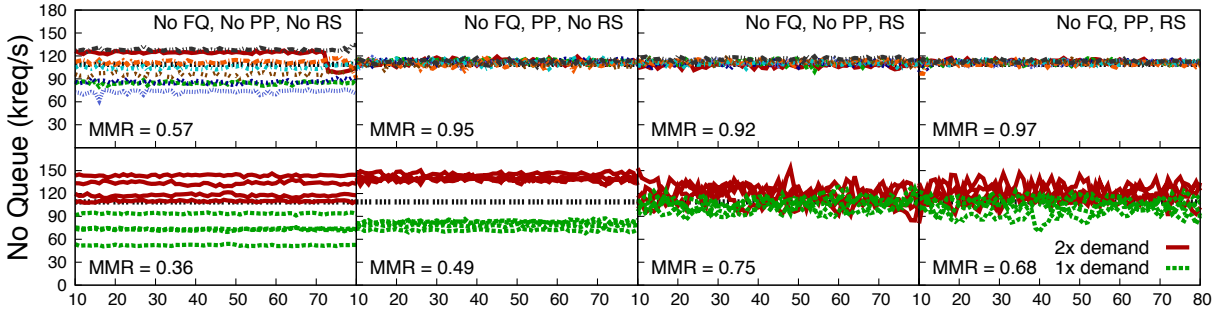
## 6.2 Achieving Fairness and Isolation

To understand how Pisces's mechanisms affect fairness and isolation, we evaluated each mechanism in turn and in combination, as shown in Figure 5. Starting with an unmodified base system (Membase) without fair queuing, we alternately add in partition placement (PP), which we pre-compute using our simple greedy heuristic, and replica selection (RS). We then repeat the combination with fair queuing (FQ) and static (uniform) local weights and complete the set of permutations with weight allocation (WA + FQ). In each experiment, 8 tenants with equal global weights attempt to access the 8-node system with the same demand. For illustrative purposes, we first present results for a simple GET workload, and summarize results for more complex workloads in Table 1. For the 1kB GET workload, the fair share (1.0 MMR) is bandwidth-limited at 109 kreq/s per node.

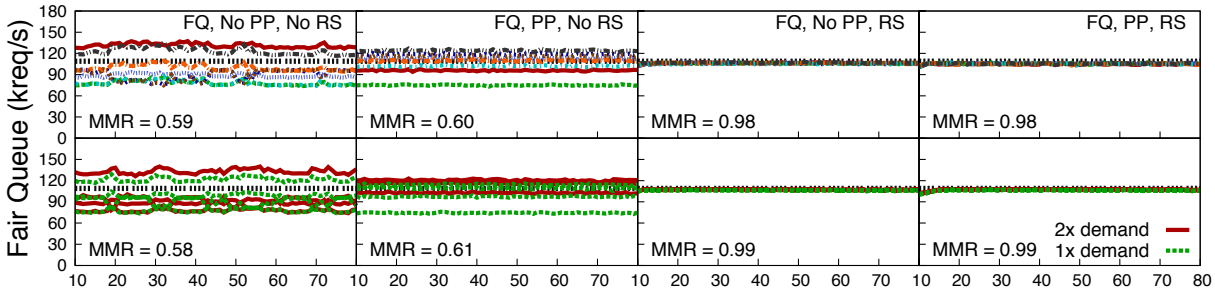
**Unmodified Membase:** The unmodified system provides poor throughput fairness (0.57 MMR) between tenants. This is largely due to the inherent skew in the tenant demand distribution which, per Section 2.2, can lead to an infeasible partition mapping. Figure 6a shows tenants 3 and 7 contending for hot partitions at server 2, while tenants 2, 4, and 6 collide on server 3 under the default, demand-oblivious placement. In contrast, Figure 6b shows how packing the partitions according to demand resolves the node capacity violations, which improves fairness. Once replica selection is enabled, though, the infeasibility issue largely disappears. By splitting request demand across replicas, RS smoothes out the hot spots. This relaxes partition bin-packing problem and allows the system to achieve high fairness both with and without PP ($\geq$ 0.92 MMR). While PP and RS help improve fairness, without request scheduling, Membase still fails to provide performance isolation. In the bottom half of Figure 5a, half the tenants double their demand, which degrades fairness across the board.

**Multi-tenant Weighted FQ:** Unsurprisingly, with weighted dominant resource fair queuing, fairness barely improves under the default placement due to over-contention for resources on the servers, as shown in Figure 5b. Even with PP, fairness fails to improve despite the feasible placement, since the tenants still access each server at different rates, as seen in Figure 6b. Although FQ enforces local (equal) weights, without aligning those weights to tenant demand, tenants with more weight than fair partition demand on a given node can still consume up to their limit and violate the global fair share. However, despite the need for weight tuning, fair queuing still proves essential for performance isolation. Where Membase falls flat under excess demand from the 2x tenants, fair queuing maintains fairness in all conditions. Unmatched weights may allow tenants to consume more than their fair share, but the tenants cannot consume more than their local allocation, which bounds the deviation from global fairness. Again, by smoothing out hot spots, replica selection resolves the demand imbalance and improves fairness with or without PP ($\sim$ 99 MMR).

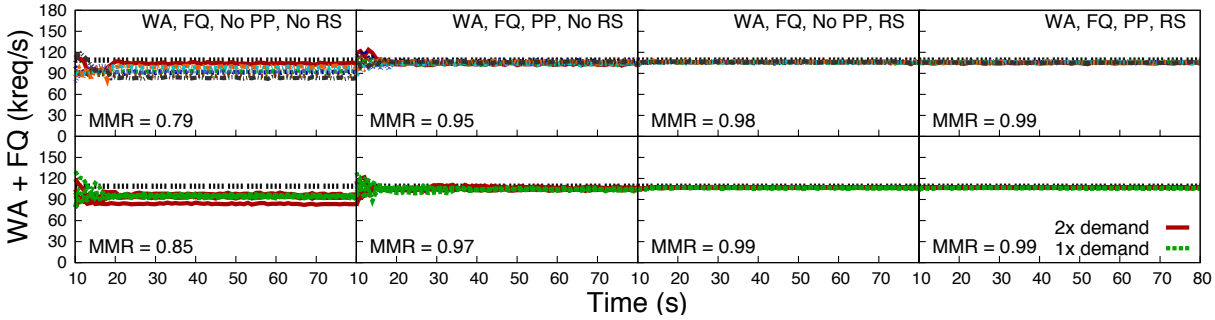
**Weight Allocation and FQ:** Enabling weight allocation unlocks Pisces's full potential, especially when replica selection is disabled (e.g. for consistency). By adapting to local tenant demands, weight allocation optimizes the system for fairness even when the placement is infeasible (0.79 MMR), as seen in Figure 5c. However, since the tenant shares are limited by the over-loaded nodes, overall throughput diminishes. Under a feasible placement, weight allocation is able to find the optimal




(a) Membase (no queuing): Unfair unless partition placement or replica selection are enabled and provides no isolation.




(b) Fair Queuing (equal local weights): Provides strong isolation, but struggles with fairness unless replica selection is enabled.




(c) Weight Allocation + Fair Queuing: Achieves the fairness under all conditions, even under infeasible partition mappings (no PP).

**Figure 5: System-wide fairness and isolation under a combination of Pisces mechanisms.**

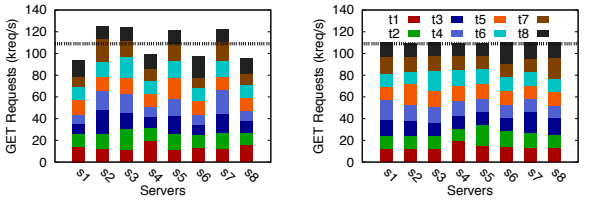
weights for the tenant demand on each server, as depicted in Figure 6c, while preserving global fairness (0.95 MMR). Since replica selection balances demand, weight allocation has little additional affect on fairness, but remains necessary to adapt to demand fluctuations. Excess demand can actually heighten the awareness of demand mismatch without PP, though, again, global throughput suffers. For all other cases, weight allocation tunes fair queuing to mediate between the tenants, which ensures high fairness ($\geq$ 0.97 MMR) and performance isolation.

In addition to throughput fairness and isolation, Pisces also provides a measure of latency isolation as well. The first two groups in Figure 7 show the average median latencies for the 1x and 2x demand tenants in the previous experiments. The max error bar indicates the 95th percentile, while the min error bar indicates the spread between the tenants' median latencies. Without fair queuing, there is little isolation with median latencies for both 1x and 2x tenants hovering around 4 ms. Adding replica selection improves isolation where the 2x tenants experience about 1.7 times the median latency as the 1x tenants. However, the latency spread (3.2 ms) spans the entire gap between them. With fair queuing, the median latencies are far more well-behaved. The 2x tenants receive a 2.1 times latency penalty with minimal ($<$1 ms) spread.

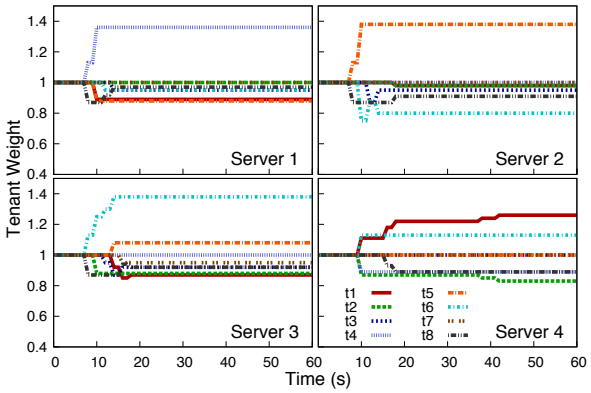
For a more thorough evaluation of Pisces fairness, we experimented over a range of log-normal distributed object sizes, where each tenant has object sizes drawn from a distribution with the same mean value (1kB or 10B) and standard deviation (0, 0.5, or 1). We also varied the workloads between all read-only, all read-heavy, or a mix of read-only, read-heavy, and write-heavy. Table 1 summarizes these results for Pisces with all mechanisms enabled. We measured mean bandwidth consumption for 1kB objects and mean request rates for 10B objects, as well as median request latency (1ms averaged) and average fairness (in terms of bandwidth consumed or request rates, respectively). For mixed workloads, values for both

| | Fixed Size Objects | | | | LogNormal 0.5 | | | | LogNormal 1.0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1kB Mean** | **BW (Gbps)** Out/In | **Lat (ms)** GET/SET | **MMR** Out/In | **MMR ratio** Out/In | **BW (Gbps)** Out/In | **Lat (ms)** GET/SET | **MMR** Out/In | **MMR ratio** Out/In | **BW (Gbps)** Out/In | **Lat (ms)** GET/SET | **MMR** Out/In | **MMR ratio** Out/In |
| *Read-Only* | 6.95 | 2.48 | 0.99 | 1.73 | 6.93 | 3.67 | 0.99 | 1.57 | 6.94 | 5.20 | 0.99 | 2.15 |
| *Read-Heavy* | 6.87/1.30 | 2.99/2.15 | 0.98/0.98 | 1.56/1.58 | 6.82/1.32 | 3.34/2.62 | 0.99/0.98 | 1.32/1.29 | 6.90/1.35 | 4.70/3.45 | 0.99/0.99 | 1.65/1.57 |
| *Mixed* | 6.81/2.06 | 2.55/2.41 | 0.68/0.77 | 1.62/1.05 | 6.81/2.08 | 3.11/2.75 | 0.67/0.77 | 1.81/0.92 | 6.76/2.01 | 4.01/3.71 | 0.63/0.74 | 1.85/1.19 |
| **10B Mean** | **Tput (kreq/s)** GET/SET | **Lat (ms)** GET/SET | **MMR** Requests | **MMR ratio** Requests | **Tput (kreq/s)** GET/SET | **Lat (ms)** GET/SET | **MMR** Requests | **MMR ratio** Requests | **Tput (kreq/s)** GET/SET | **Lat (ms)** GET/SET | **MMR** Requests | **MMR ratio** Requests |
| *Read-Only* | 3,160 | 3.37 | 0.97 | 1.33 | 3,151 | 3.52 | 0.97 | 1.26 | 3,104 | 3.36 | 0.96 | 1.35 |
| *Read-Heavy* | 2,567/285 | 3.04/8.19 | 0.98 | 1.28 | 2,593/288 | 2.96/8.30 | 0.98 | 1.42 | 2,540/282 | 3.00/8.49 | 0.94 | 1.34 |
| *Mixed* | 2,343/445 | 2.80/7.79 | 0.96 | 1.5 | 2,325/444 | 2.88/7.84 | 0.96 | 1.32 | 2,295/437 | 2.94/7.78 | 0.94 | 1.38 |

**Table 1: Pisces performance over a range of log-normally distributed object sizes and GET/SET workloads**




(a) Uniform partition placement (b) Demand-aware placement




(c) Local tenant weights evolving with weight allocation

**Figure 6: Skewed demand can lead to infeasible partition mappings (a). Placing partitions according to tenant demand and node capacities ensures feasibility (b). Weight allocation, in turn, adjusts the per-node tenant weights to the demand (c) (4 of 8 shown).**

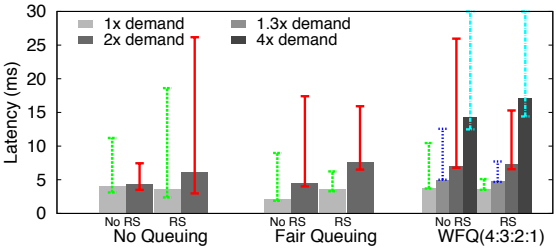



**Figure 7: Fair queuing protects request latencies for even and weighted tenant shares.**

GET/SET (Out/In bandwidth) requests are shown.We also include a fairness comparison in terms of the MMR ratio between Pisces and unmodified Membase.

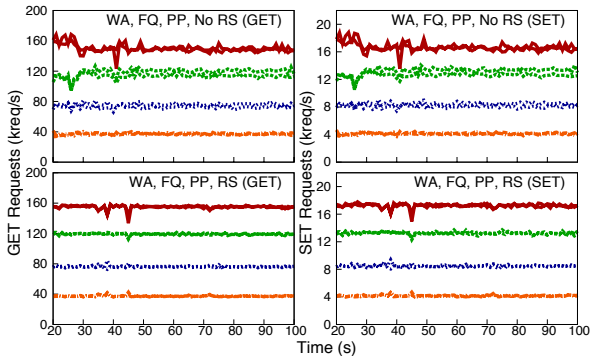
Pisces is able to achieve over 0.94 MMR fairness for most size variations and workloads. Moreover, Pisces exceeds the fairness of the unmodified base system by more than 1.3 times for most cases as well. The mixed workload for 1kB objects, however, proved to be a particularly troublesome combination. While the read-heavy and write-heavy tenants received their appropriate shares of inbound write bandwidth (0.99 MMR), the read-only tenants were able to consume more outbound read bandwidth than either one, resulting in an overall fairness between 0.63 and 0.64 MMR. One reason for this is the head-of-line blocking of a tenant's read requests due to its write requests fully consuming its current inbound bandwidth allocation. This can push subsequent read requests to the next scheduler round, despite having unconsumed outbound bandwidth remaining. Since read-only tenants never bottleneck on inbound bandwidth, they can fully consume their share (and more) of outbound bandwidth.

To fix this issue, we modified the clients to prioritize read requests over write requests when sending requests to the servers. While this largely resolved the issue for read-heavy tenants (> 0.9 MMR), the write-heavy tenants remained obstructed by the bandwidth bottleneck. The low MMR ratio for writes (bytes in) compared to Membase is also attributable to this performance write-bottlenecking variance. For 10B workloads, the problem disappears (≥ 0.95 MMR) since both read and write workloads share a single bottleneck: request-rate without suffering packet-level congestion effects. Pisces is able to once again claim its fairness crown from Membase for these cases (> 1.28x more fair than Membase).

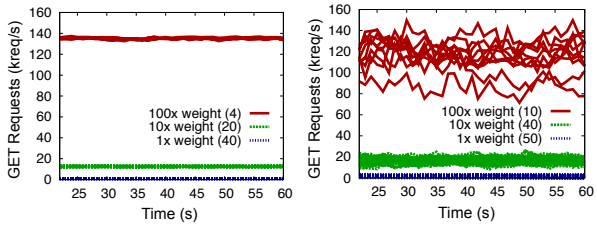
## 6.3 Service Differentiation

So far, we have demonstrated that Pisces's mechanisms, working in concert, can achieve both isolation (FQ) and nearly ideal even fair sharing (PP + WA or RS). We now turn our attention to providing *weighted* fairness at the global level (for service differentiation) and at the local level (for dominant resource fairness).

**Global Differentiation:** In Figure 8a, the tenant are assigned global weights in decreasing order 4:4:3:3:2:2:1:1 to differentiate their aggregate share of the system resources. Both with and without replica selection, Pisces is able to achieve high global weighted fairness (> 0.9 MMR) for both in and out bandwidth under a 1kB request read-heavy workload. Similarly, request latency remains strongly isolated between the tenants. Since all tenants

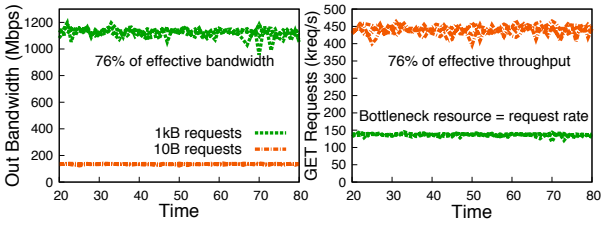



(a) Global weighted 90% GET / 10% SET throughput

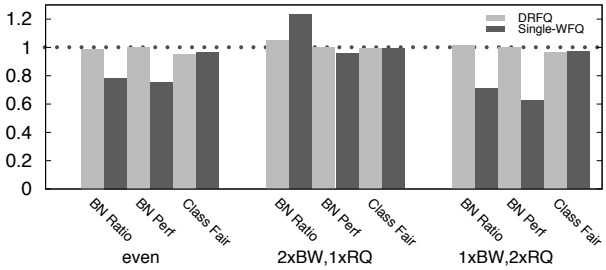



(b) 64 tenant weighted thruput (c) 100 tenant weighted thruput

**Figure 8: Subfigure (a) demonstrates global 4:3:2:1 weighted fair sharing for a read-heavy workload. In (b) and (c), Pisces abides by the skewed tenant weights on an 8 and 20 node cluster respectively.**




(a) Dominant resource fairness between 1kB and 10B workloads




(b) Bottleneck ratio fairness and performance with and without multi-resource queuing

**Figure 9: Pisces provides dominant resource fairness between bandwidth (10kb) and request-rate (10B) bound tenants (a). Compared to single-resource fair queuing, DRF provides a better share of the bottleneck resource (BN Ratio) and better performance (BN Perf) over different tenant weights.**

generate the same demand, the lower weight tenants (3, 2, and 1) exceed their share by 1.3, 2, and 4 times respectively. Per Figure 7, the latencies of these tenants closely mirror their demand ratios. In both cases, the median (or mean) latency spread is small (<1 ms), except for that of the smallest-weight tenant (<2.7 ms).

To further stress the fairness properties of the system, we ran two larger scale experiments where the number of tenants far exceeded the number of servers, to mimic more realistic service scenarios. In Figure 8b, each of 64 tenants reside on 4 out of the 8 available servers with each server housing 32 tenants. To reflect the highly skewed nature of tenant shares, i.e. a few heavy hitters and many small, low-demand users, we configured the tenant weights along a log-scale: 4 tenants with weight 100, 20 with weight 10, and 40 with weight 1. Within each weight class, Pisces achieves > 0.91 MMR. Between classes, however, weighted fairness decreases to 0.56 MMR. This deviation is due to the limits of the DWRR scheduler token granularity. With such highly skewed weights, tenants in the smallest weight class (1) only receive fractional tokens, which means their requests are processed once every few rounds, which results in a lower relative share. Fairness between the weight 100 and weight 10 tenants, however, remains high (0.91 MMR). In practice, to work around the limited resolution of the WFQ scheduler, the service provider can cap the number of tenants per server to ensure reasonable local weights (token) and match the desired rate guarantees. Figure 8c shows a larger scaled out experiment, with 100 tenants resident on 6 of 20 servers (30 tenants per server). While we see a qualitatively similar result, the actual fairness degrades considerably (average 0.68 MMR between high and medium weight tenants, 0.46 MMR across all classes). However, this is mostly due to performance variance on the scale-out testbed [25] arising from CPU scheduling and network bottleneck effects, which affects the high-weight tenants disproportionately. As a result, the low-weight tenants can consume a larger share.

**Local Dominant Resource Fairness:** While Pisces provides weighted fairness on the global level, we want to ensure that each tenant receives a weighted share of its dominant resource on the local level. To stress different dominant resources and their corresponding bottlenecks, we experimented with four tenants requesting 1kB objects (bandwidth limited), while another four operated on 10B objects (request-rate limited). Under even weighting, Figure 9a shows how Pisces enforces fairness within each dominant resource type (>0.95 MMR) and evenly splits the dominant resource shares between types: the 1kB tenants receive ~76% of the effective outbound bandwidth and the 10B tenants receive ~76% of the effective request rate.[3] Although the tenants differ in dominant resource, they share the same bottleneck resource (request rate in this case). By computing the bottleneck resource ratio (BNR) between the different dominant resource tenants,

[3]This is lower than the optimal rates since transmitting a 1kB request takes longer than processing a 10B request.

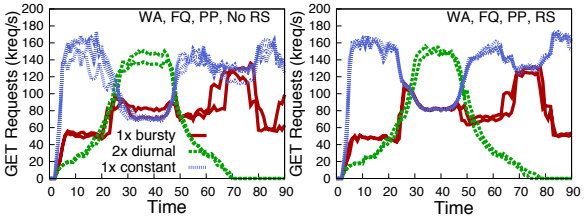
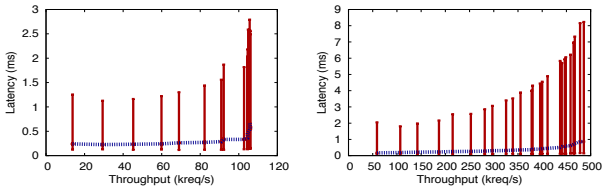



**Figure 10: Pisces responds to demand dynamism according to tenant weights.**




(a) Pisces $\mu$-benchmark (1kB) (b) Pisces $\mu$-benchmark (10B)

**Figure 11: Median throughput versus latency microbenchmark with 99th-percentile error bars.**

we can directly determine the dominant resource shares as shown in section 2.2 without having to estimate the effective resource rates. In this instance, the BNR is around 3.2, which gives the 10B tenants 76% of the request rate, and allows the 1kB tenants to consume 76% of the bandwidth.

Using BNR, we compare the dominant resource fairness of the DRF-enabled scheduler versus the single-resource (request) version for even weighted tenants, 2x weighted 1kB (bw) vs. 1x weighted 10B (rq) tenants, and 1x weighted 1kB vs. 2x weighted 10B tenants. For the even and 1x bandwidth, 2x request cases, the bottleneck resource is request rate. In the 2x bandwidth, 1x request scenario, the tenants bottleneck on bandwidth. As Figure 9b shows, the DRF scheduler outperforms the single resource version in all cases, achieving the best normalized BNR value and bottleneck resource performance (BN Perf). The single-resource scheduler holds a slight edge in fairness between tenants of the same dominant resource class (Class Fair) in the even and 1x bandwidth, 2x request cases. As in the mixed workload experiments, achieving the highest 10B request throughput required additional packet prioritization. We enabled the linux priority queue scheduling discipline to reduce the scheduler delay for small requests while waiting for connection send buffers to clear for the 1kB tenants.

## 6.4 Dynamic Workloads

Dynamic workloads present a challenge for any system to provide consistent, predictable performance. In Figure 10, 2 bursty demand tenants (weight 1), 2 diurnal demand tenants (weight 2), and 4 constant demand tenants (weight 1) access the storage system. Initially, the tenants consume less than the full system capacity which allows the constant tenants to consume a larger proportion. As the diurnal tenants ramp up between 0 and 20 seconds, they begin to consume their share of throughput which cuts into the excess share consumed by the constant tenants. Around 20s, both the bursty tenants and diurnal tenants ramp up to their full load, which results in a nearly 2 to 1 ratio, according to the tenant weights. The diurnal tenants tail off around 50s along with the bursty tenants which allows the constant demand tenants to, once again, consume in excess of their fair share (~ 80 kreq/s). Lastly, at 70s, the bursty tenants issue one last barrage of requests, which forces the constant tenants to share the throughput equally. Both with and without replica selection enabled, Pisces is able to handle the demand fluctuations and provide fair access to the storage resources.

## 6.5 Efficiency and Overhead

To assess the efficiency and overhead, we ran a microbenchmark of a single WFQ Pisces node. In this experiment, tenants issue requests at increasing rates to a single service node. As shown in Figure 11, Pisces is able to achieve over 106 kreq/s for 1kB requests, which is > 96% of Membase throughput, with the same average request latency (0.14 ms). For 10B requests, Pisces actually outperforms Membase by 20% (485 vs. 405 kreq/s) with lower average request latency (0.15 vs. 0.16 ms) due to the DWRR scheduler's work stealing mechanism.

# 7. Related Work

**Sharing the network.** Most work on sharing datacenter networks has used either static allocation or VM-level fairness. The static allocations by SecondNet [13] and Oktopus [6] guarantee bandwidth but can leave the network underutilized. While more throughput efficient, Gatekeeper's ingress and egress scheduling [27], Seawall's congestion-controlled VM tunneling [28], and FairCloud's per-endpoint sharing [26], respectively provide fairness on a per-VM, VM-pair, or communicating-VM-group basis, rather than on a per-tenant basis.

NetShare [17] and DaVinci [14] take a per-tenant network-wide perspective, but the former allocates local per-link weights statically, while the latter requires non-work-conserving link queues and does not consider fairness. In contrast, Pisces achieves per-tenant fairness by leveraging replica selection and adapting local weights according to demand, while maintaining high utilization.

**Sharing services.** Recent work on cloud service resource sharing has focused mainly on single-tenant scenarios. Parda [11] applies FAST-TCP congestion control to provide per-VM fair access, which Pisces uses as well, but for replicated service nodes. mClock [12] adds limits and reservations to the hypervisor's fair I/O scheduler to differentiate between local VMs, while Pisces's DWRR scheduler operates on a per-tenant level. Argon [32] uses cache management and time-sliced disk scheduling for performance insulation on a single shared file server. Pisces

could adapt these techniques for memory and disk I/O resources. Stout [20] exploits batch processing to minimize request latency, but does not address fairness.

Several other systems focused on course-grained allocation. Autocontrol [23] and Mesos [15] allocate per-node CPU and memory to schedule batch jobs and VMs, using utility-driven optimization and dominant resource fairness, respectively. They operate on a coarse per-task or per-VM level, however, rather than on per-application requests. In [10], the authors apply DRF to fine-grained multi-resource allocation, specifically to enforce per-flow fairness in middleboxes. However, their DRF queuing algorithm relies on virtual time, and it scans each per-flow packet queue for the lowest virtual start time.

# 8. Conclusion

This paper seeks to provide system-wide *per-tenant* weighted fair sharing and performance isolation in multi-tenant, key-value cloud storage services. By decomposing this problem into a novel combination of four mechanisms—partition placement, weight allocation, replica selection, and fair queuing—our Pisces system can fairly share the aggregate system throughput, even when tenants contend for shared resources and demand distributions vary across partitions and over time. Our prototype implementation achieves near ideal fairness (0.99 Min-Max Ratio) and strong performance isolation.

**Acknowledgments.** We thank Jennifer Rexford for helpful discussions early in this project. Funding was provided through NSF CAREER Award #0953197.

# References

[1] `http://aws.amazon.com/dynamodb/faqs/`, 2012.

[2] `http://docs.amazonwebservices.com/amazondynamodb/latest/developerguide/Limits.html`, 2012.

[3] `http://www.couchbase.org/`, 2012.

[4] `http://code.google.com/p/spymemcached/`, 2012.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *SIGCOMM*, 2008.

[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[8] S. L. Garfinkel. An evaluation of Amazon's grid computing services: EC2, S3 and SQS. Technical Report TR-08-07, Harvard Univ., 2007.

[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[10] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource scheduling for middleboxes. In *SIGCOMM*, 2012.

[11] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[12] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, 2010.

[13] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNext*, 2010.

[14] J. He, R. Zhang-Shen, Y. Li, C.-Y. Lee, J. Rexford, and M. Chiang. Davinci: dynamically adaptive virtual networks for a customized internet. In *CoNext*, 2008.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[16] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *CCGrid*, 2011.

[17] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. Netshare: Virtualizing data center networks across services. Technical Report CS2010-0957, UCSD, 2010.

[18] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, 2009.

[19] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[20] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: an adaptive interface to scalable cloud storage. In *USENIX Annual*, 2010.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[22] R. M. Nauss. Solving the generalized assignment problem: An optimizing and heuristic approach. *INFORMS J. Computing*, 15 (Summer):249–266, 2003.

[23] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, 2009.

[24] D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *JSAC*, 24(8):1439–1451, 2006.

[25] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton CS, 2011.

[26] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[27] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, 2011.

[28] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *NSDI*, 2011.

[29] D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Math. Prog.*, 62(1):461–474, 1993.

[30] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Trans. Networking*, 4(3):375–385, 1996.

[31] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bulletin*, 9(1):4–9, 1986.

[32] M. Wachs, M. Abd-el-malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[33] J. Wang, P. Varman, and C. Xie. Optimizing storage performance in public cloud platforms. *J. Zhejiang Univ. – Science C*, 11(12): 951–964, 2011.

[34] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. Fast TCP: Motivation, architecture, algorithms, performance. *Trans. Networking*, 14(6): 1246–1259, 2006.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.