OWL: Understanding and Detecting Concurrency Attacks

Shixiong Zhao*, Rui Gu†, Haoran Qiu*, Tsz On Li*, Yuexuan Wang*, Heming Cui*, and Junfeng Yang†
*Department of Computer Science, The University of Hong Kong
†Department of Computer Science, Columbia University
Email: {sxzhao, hrqiu, amywang, heming}@cs.hku.hk, {ruigu, junfeng}@cs.columbia.edu

***Abstract*—Just like bugs in single-threaded programs can lead to vulnerabilities, bugs in multithreaded programs can also lead to concurrency attacks. We studied 31 real-world concurrency attacks, including privilege escalations, hijacking code executions, and bypassing security checks. We found that compared to concurrency bugs' traditional consequences (e.g., program crashes), concurrency attacks' consequences are often implicit, extremely hard to be observed and diagnosed by program developers. Moreover, in addition to bug-inducing inputs, extra subtle inputs are often needed to trigger the attacks. These subtle features make existing tools ineffective to detect concurrency attacks.**

To tackle this problem, we present OWL, the first practical tool that models general concurrency attacks' implicit consequences and automatically detects them. We implemented OWL in Linux and successfully detected five new concurrency attacks, including three confirmed and fixed by developers , and two exploited from previously known and well-studied concurrency bugs. OWL has also detected seven known concurrency attacks. Our evaluation shows that OWL eliminates 94.1% of the reports generated by existing concurrency bug detectors as false positive, greatly reducing developers' efforts on diagnosis. All OWL source code, concurrency attack exploit scripts, and results are available on github.com/hku-systems/owl.

***Keywords*-Software Testing; Concurrency Attacks**

I. Introduction

Driven by the rise of multi-core hardware, multi-threaded programs are already pervasive. Unfortunately, these programs are plagued with concurrency bugs (i.e., shared memory accesses without proper synchronization among threads) [1]. Harmful concurrency bugs often corrupt critical global memory and immediately lead to explicit consequences such as wrong outputs and program crashes [2]–[4].

Worse, recent studies [5], [6] reveal that concurrency bugs can lead to *concurrency attacks*: by triggering concurrency bugs, hackers may leverage the corrupted memory to conduct much broader types of security consequences, including privilege escalations [7], hijacking code execution [8], bypassing security checks [9], and breaking database integrity [5]. In this paper, we studied 31 concurrency attacks across multiple platforms such as `Windows` and `Linux`. We also built scripts to successfully exploit 12 attacks across six real-world programs in `Linux`.

S. Zhao and R. Gu equally contribute to this work.

Our study identified two subtle features for concurrency attacks compared to concurrency bugs and their traditional consequences. First, the corrupted memory often resides a long time in programs, and then abruptly triggers severe consequences which are often *implicit*: for 14 out of 31 concurrency attacks we studied, developers were often not aware of the attacks consequences (e.g., bypassing security checks) when the attacks succeeded. This feature makes concurrency attacks extremely hard to be detected, diagnosed, or fixed.

Second, in addition to the inputs for inducing concurrency bugs, 10 out of the 12 concurrency attacks we reproduced require *extra subtle inputs* to conduct the attacks. For instance, in a new `Linux` OS privilege escalation detected by us (CVE-2017-7533 [10]), triggering the memory corruption only required two threads using crafted inputs (calling `inotify()` and `rename()`). Nevertheless, to trigger the root privilege escalation, a third thread is required to call a `socket()` to allocate a SELinux label structure using `kmalloc32()`. This structure must be close to the memory corrupted by the two threads, so that a kernel heap overflow can be triggered to construct a privilege escalation.

These extra attack-inducing inputs are extremely hard to be inferred by developers or existing tools. For seven out of the 12 attacks for which we had source code and that we were able to reproduce, their concurrency bug triggering sites and their attack triggering sites are widely spanned across different functions. Conducting this long bug-to-attack propagation often needs subtle inputs. Moreover, this propagation often infects other threads disrelated to trigger the concurrency bugs, and these threads are often driven by extra subtle inputs to reach vulnerable sites (e.g., `setuid()`).

These two subtle features make existing concurrency bug consequence analysis tools ineffective on detecting concurrency attacks. Existing tools go along two directions. First, 2AD [5] proposes a specific concurrency attack model for database programs, but this tool is unable to detect concurrency attacks in other programs. The other tools go along another direction (e.g., ConMem [4]), which targets general programs' concurrency bugs and their explicit consequences (e.g., program crashes). Unfortunately, most consequences in the concurrency attacks we studied are implicit and they are ignored by these tools. Moreover, none of existing tools in the two directions can infer the extra attack-inducing inputs.

Overall, despite much effort, a general concurrency attack analysis tool is highly desirable but missing.

To address this problem, we present a general concurrency attack inference model (§III-B) which incorporates the two features. In this model, the lifecycle of a concurrency attack consists of three phases. First, a concurrency bug is triggered and it corrupts shared memory. Second, the corrupted memory propagates along control-flow and data-flow, spreading across functions (i.e., inter-procedural propagation). Meanwhile, the corrupted memory can go across memory boundaries (e.g., buffer overflows) and infect other threads disrelated to trigger the concurrency bugs (i.e., inter-thread propagation). Finally, corrupted memory leads to severe security consequences when it flows to vulnerable sites (e.g., `eval()` and `setuid()`).

Leveraging this model, we present OWL, the first general concurrency attack detection tool (§III-C). A key challenge is to automatically infer the extra attack-inducing inputs. Conventional approaches like symbolic execution (e.g., UC-KLEE [11]) encounter path explosions when inter-procedural or inter-thread propagation analysis is needed.

Our observation is that most attack inducing inputs for concurrency attacks already exist in a program's own or third-party test suites, and new global memory allocated by the extra inputs are often close to corrupted memory infected by concurrency bugs. Leveraging this observation, we built a new Attack Input Fuzzer (§III-C). It intercepts the instructions which may cause inter-thread propagations (e.g., buffer overflows) to dynamically monitor memory layouts at runtime and to record which instruction allocates the overflowed memory. Based on the recorded instructions, the fuzzer runs the test suites and automatically pinpoints the attack-inducing inputs from the test suites.

Another challenge is developing a scalable analysis method that can infer an inter-procedural bug-to-attack propagation. Our study shows that although the consequences of concurrency attacks are implicit, these consequences are triggered by five well-formatted types of vulnerable sites, including memory operations (e.g., `strcpy()`), NULL pointer dereferences, privilege operations (e.g., `setuid()`), file operations (e.g., `access()`), and process-forking operations (e.g., `eval()` in shell scripts). Moreover, a bug triggering instruction and its attack triggering instructions often shared a similar call stack. OWL introduces a scalable inter-procedural analyzer: it starts from bug triggering instructions in a report generated by concurrency bug detectors, propagates corrupted memory using data-flow and control-flow along the bug report's call stack, and finally locates potentially vulnerable sites.

We implemented OWL using LLVM [12] in Linux. OWL incorporates a set of concurrency bug detectors, including TSAN [13], VALGRIND [14] for user space and KTSAN [15], SKI [16] for kernel space. We evaluated OWL on six diverse, widely used programs, including `Apache`, `Chrome`, `Libsafe`, `Linux`, `MySQL`, and `SSDB`. OWL eliminated 94.1% of concurrency bug reports as false positives. With the greatly reduced reports, OWL effectively detected five new severe concurrency attacks, including three new attacks (CVE-2017-7533 [10], CVE-2016-1000324 [17], and CVE-2017-12193 [18]) confirmed and fixed by developers, and two new attacks exploited previously known and well-studied bugs in `Apache`. In addition, OWL detected seven known concurrency attacks without missing any one. The time cost of OWL was reasonable for testing.

The main contribution of this paper is OWL, the first practical, general concurrency attack detection tool. OWL incorporates a general model for understanding concurrency attacks and a new, practical attack input fuzzer. OWL has successfully detected five new concurrency attacks, and three of them have been confirmed and immediately fixed by the developers of RedHat, `Android`, and `SSDB`. The other two new attacks OWL detected were exploited on previously known and well studied concurrency bugs on `Apache`, which shows that existing enormous set of concurrency bugs deserve an extensive re-investigation for their concurrency attack consequences. We envision that OWL will attract further attention not only on detecting concurrency attacks, but also on diagnosing, fixing and defending against them.

The rest of this paper is structured as follows. §II introduces the background of concurrency attacks. §III gives an overview of the concurrency attack model and architecture of OWL. §IV and §V state the implementation of OWL. We evaluate OWL in §VI. We discuss and conclude in §VII.

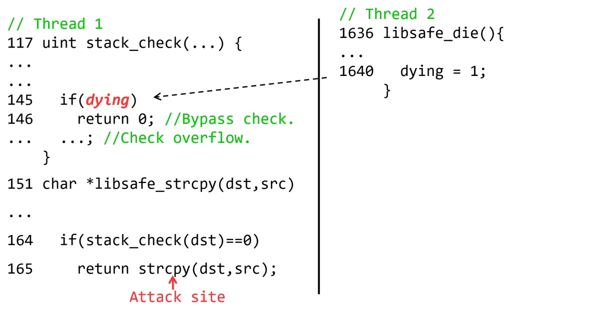
II. BACKGROUND

A. Concurrency Bugs

Multi-threaded programs are prone to concurrency bugs, causing great loss in the real world [19]. Concurrency bugs are usually caused by shared memory access without proper synchronization among threads. Data race is the most common type of concurrency bugs, defined as two threads accessing the same memory byte concurrently and at least one access is write [20], [21]. Data race detecting is already mature in academia and industry [13], [14], [16]. In this paper, we augment current data race detectors to build OWL, the first practical and general concurrency attack detector.

CONMEM [4] focuses on explicit consequences of concurrency bugs. Its evaluation shows that 26 out of 70 real-world concurrency bugs only cause minor function issues, 7 of them cause program hangs and 37 of them cause program crashes. These findings have driven researchers making tremendous progress on detecting severe concurrency bugs. CONMEM [4] has successfully targeted concurrency bugs that result in program crashes and introduced a model explaining how concurrency bugs cause explicit memory crash. CONSEQ [22] augments this model, taking advantage of the explicit crash reports and implementing a backward approach to help improve the accuracy of concurrency bugs detection.

These tools [4], [22] assume that bug inducing instructions and their consequence inducing instruction are within the same function and they provide intra-procedural propagation analysis. However, our study reveals that, concurrency attack inducing instructions and their bug inducing instructions are widely spanned across different functions. For instance, Fig. 1 illustrates a stack overflow attack caused by data race in `Libsafe`. The variable corrupted in `stack_check` afterward causes a violation of stack overflow protection in `libsafe_strcpy`. This makes existing tools [4], [22] ineffective to pinpoint concurrency attacks.



```
// Thread 1
117 uint stack_check(...) {
...
...
145   if(dying)
146     return 0; //Bypass check.
...     ...; //Check overflow.
    }
151 char *libsafe_strcpy(dst,src)
...
164   if(stack_check(dst)==0)
165     return strcpy(dst,src);
```

```
// Thread 2
1636 libsafe_die(){
...
1640   dying = 1;
     }
```

Figure 1: A concurrency attack in the `Libsafe` security library. Dotted arrows mean the bug-triggering thread interleaving.

B. Concurrency Attacks

A prior study [6] browses the bug databases of 46 real-world concurrency bugs and presents three major findings on concurrency attacks. First, concurrency attacks are severe threats: 35 of the bugs can corrupt critical memory and cause three types of violations, including privilege escalation, malicious code injection, and bypassing security authentication. Second, with crafted bug-inducing inputs, concurrency bugs that lead to attacks can be often triggered with high probability (less than 20 repeated program executions). Third, compared to traditional TOCTOU attacks, which stem from corrupted file accesses, pinpointing concurrency attacks is much more difficult because they stem from corrupted and miscellaneous memory accesses.

These three findings reveal that concurrency attacks can weaken or even bypass existing sequential defense tools, because these tools are mainly designed for sequential attacks. This prior study raises an open research question: what should an effective tool be for detecting concurrency attacks? Specifically, can existing concurrency bugs detection tools effectively detect these bugs and their attacks? The answer is probably NO because literature has overlooked these attacks [6].

A recent work 2AD [5] focused on detecting concurrency attacks in database programs. It introduces a new attack model called ACIDRain attack for databases. However, this tool is designed for special programs.

We conducted the first quantitative study on 31 concurrency attacks across four OS platforms (shown in Table I). To gain a deep understanding, we built scripts for 12 concurrency attacks among six real-world programs. We identified the two aforementioned features (§I).

Table I: A summary of consequences for the 31 concurrency attacks we studied across four platforms.

	Linux	Windows	Darwin	FreeBSD
Privilege Escalation	4	3	1	2
Inject Malicious Code	2	0	0	0
Bypass Security Check	1	0	0	1
Violate Integrity	1	1	1	0
DoS/Crash	9	2	2	1
Total	17	6	4	4

First, among 14 of 31 concurrency attacks, the corrupted memory often resides a long time in the programs, abruptly triggering severe consequences and remaining *implicit* to software developers. For instance, in the `Libsafe` attack we mentioned before (Fig. 1), the raced variable in `stack_check()` affects behaviors of `libsafe_strcpy`, and hence the stack overflow protection is silently bypassed. Moreover, for the five new attacks we detected (§VI-A), four of them incur implicit consequences (`Linux` root privilege escalation, file corruption, and use-after-free) without crashing the programs, and only one is explicit (DoS).

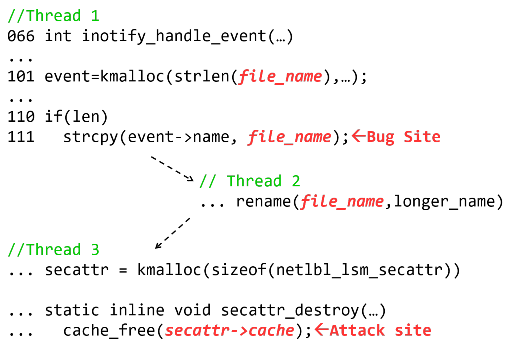
Second, in addition to the inputs for triggering concurrency bugs, 10 out of 12 required *extra subtle inputs* to conduct concurrency attacks. For instance, in the CVE-2017-7533 attack (Fig. 2), the data race caused by `inotify_handle_event()` and `rename()` can cause a kernel heap overflow when the second parameter of rename exceeds the memory allocated before. Since kernel processes share the same kernel heap, a third system call `socket()` allocates a `netlbl_lsm_secattr` structure on the same heap close to the corrupted memory. Hence, the first field of `netlbl_lsm_secattr` will be overwritten once the overflow succeeds. We leveraged this procedure to achieve a privilege escalation. Another known concurrency attack, CVE-2004-1235 [7], also requires extra threads to call extra system calls in addition to the bug triggering system calls.

Overall, no existing tool has incorporated either of these two features and we owe this as the key reason that existing tools are in-effective on detecting concurrency attacks.

C. Related works

Concurrency reliability tools. Various prior systems work on concurrency bug detection [3], [4], [21]–[27], diagnosis [27]–[32], and correction [33], [34]. They focus on concurrency bugs themselves, while OWL focuses on security related consequences of concurrency bugs. Therefore, these systems are complementary to OWL.

ConMem [4] is the most relevant tool to OWL, but it is not open source. ConMem is not able to detect some concurrency attacks (e.g.CVE-2017-7533) because of



```
//Thread 1
066 int inotify_handle_event(…)
...
101 event=kmalloc(strlen(file_name),…);
...
110 if(len)
111   strcpy(event->name, file_name);←Bug Site

                          // Thread 2
                         ... rename(file_name,longer_name)

//Thread 3
... secattr = kmalloc(sizeof(netlbl_lsm_secattr))

... static inline void secattr_destroy(…)
...   cache_free(secattr->cache);←Attack site
```

Figure 2: A new data race bug and privilege escalation attack detected by OWL in `Linux` Kernel. Both the bug and attack have been confirmed by RedHat, Android, and Kernel group, and assigned CVE-2017-7533.

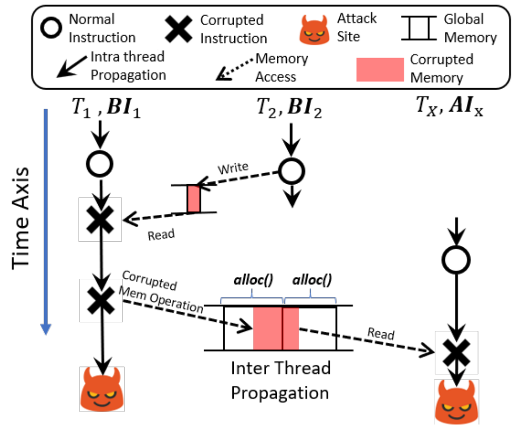


Figure 3: *Concurrency Attack Model*

two main reasons. First, ConMem mainly handles explicit consequences such as program crashes, and it does not describe how to detect buffer overflows. Second, ConMem is not designed to infer the extra attack-inducing inputs.

Static & Dynamic vulnerability detection tools. There are already a variety of static and dynamic vulnerability detection approaches [35]–[38]. Benjamin et al. [39] leverage pointer analysis to detect data flow from unchecked inputs to sensitive sites. This approach ignores control flow and thus it is not suitable to track concurrency attacks like the Libsafe one [40]. Yamaguchi et al. [35] did not incorporate inter-procedural analysis and thus is not suitable to track concurrency attacks either. Moreover, these general approaches are not designed to reason about concurrent behaviors (e.g., [35] cannot detect data races).

Symbolic execution. Scheduler control is a way to exploiting synchronization bugs by using interrupting and scheduling threads. Scheduler control can be utilized by OWL to find thread execution order of triggering concurrency attacks. Symbolic execution is an advanced program analysis technique that can systematically explore a programs execution paths to find bugs. Researchers have built scalable and effective symbolic execution systems to detect software bugs [41]–[44], block malicious inputs [45], preserve privacy in error reports [46], and detect programming rule violations [47]. Specifically, UCKLEE [44] has been shown to effectively detect hundreds of security vulnerabilities in widely used programs. Symbolic execution is orthogonal to OWL; it can augment OWL's input hints by automatically generating concrete vulnerable inputs.

III. Overview

A. Preliminaries

Patterns of concurrency bugs and attacks are diverse and complicated. We design the model to be general so that it can cover all the real-world concurrency attacks we studied. Therefore, we give some preliminaries and simplification first in order to make our discussion clear.

Inputs are defined as the data a program reads from its execution environment, including not only the data read from files and sockets, but also command line arguments, return values of external functions such as `gettimeofday()`, and any external data that can affect program execution. **Bug-inducing Inputs** (*BI*) are the series of inputs that trigger a concurrency bug. **Attack-Inducing Inputs** (*AI*) are the inputs that trigger a concurrency attack. Each **Input** is fed to one **Thread** (*T*), and we express each composition of inputs and threads with (T_i, I_i). Triggering a concurrency bug requires: (T_1, BI_1), (T_2, BI_2), ..., (T_n, BI_n), n≥2. Triggering a concurrency attack requires: (T_1, BI_1), (T_2, BI_2), ..., (T_n, BI_n), (T_{n+1}, AI_1), (T_{n+2}, AI_2), ..., (T_{n+m}, AI_m), n≥2, m≥0. In most concurrency attacks we studied, $n = 2$ *and* $m = 0$ *or* 1. Specifically, when $m = 0$, the attack site is within a bug inducing thread (T_1 or T_2). Therefore, the rest of this paper considers $n = 2$ *and* $m = 0$ *or* 1.

Global memory contains two kinds of memory space, the shared memory among threads within a program and the global kernel space. **Corrupted memory** is the memory corrupted by a concurrency bug, and the following memory pieces infected by the firstly corrupted memory. Corrupted memory propagates along data-flow and control-flow with **Corrupted Instructions** (instructions that operate on corrupted memory). We refer **Attack Sites** to corrupted instructions that may cause attacks, including memory operations (e.g., strcpy()), NULL pointer deferences, privilege operations (e.g., setuid()), file operations (e.g., `access()`), and process-forking operations (e.g., `eval()` in shell scripts). Moreover, we refer **Corrupted Memory Operations** to memory operations (e.g., `memcpy()` or direct writes to memory) that manipulate corrupted memory or are infected by corrupted memory. **`alloc()`** represents all the memory allocation functions across programs (e.g., `malloc()` in C program) and platforms (e.g., `kmalloc()` in `Linux` Kernel).

B. Concurrency Attack Model

We present a general concurrency attack model (Fig. 3) with three phases: bug triggering, bug-to-attack propagation,

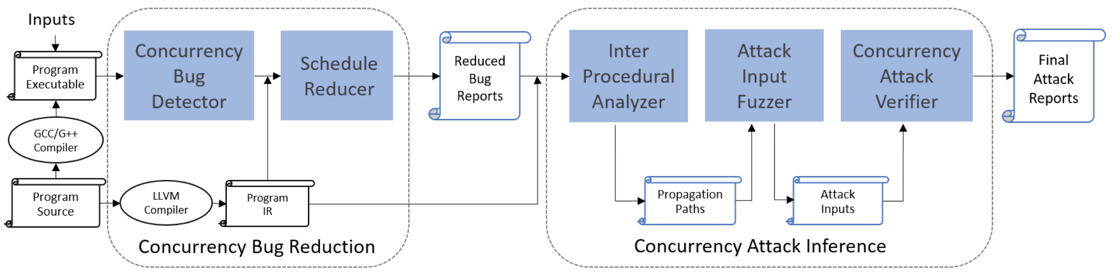


Figure 4: OWL's Architecture with two phases

and attack triggering.

Bug Triggering. In this model, a concurrency bug is triggered by $(T_1, BI_1), (T_2, BI_2)$ with certain thread interleaving. For instance, in Fig. 3, BI_1 and BI_2 trigger a data race on the same byte in global memory.

Bug-to-attack Propagation. Corrupted memory propagates along control-flow and data-flow through program execution. When corrupted memory propagates to an attack site (e.g., `setuid()`), a concurrency attack succeeds. For instance, if corrupted memory affects any parameter in `memcpy()`, an overflow can happen and infect other threads (T_x). If T_x is fed with AI_x which drives T_x to read some overflowed memory, T_x will become vulnerable and trigger a concurrency attack.

Attack Triggering. When the bug triggering thread (T_1) or the vulnerable thread (T_x) executes an attack site and the parameters of the attack site refer to corrupted memory, a concurrency attack can succeed. Currently our model contains five types of attack sites (§III-A), and more types can be added in the future.

C. OWL*'s Architecture*

Fig. 4 shows OWL's architecture, which contains two major phases. The first phase is to reduce false positive reports, including a concurrency bug detector and a schedule reducer. The second phase is to infer actual attacks, including an inter-procedural static analyzer, an attack input fuzzer, and an attack verifier. The inputs of OWL are a program's source code and test suites.

OWL takes a bug report from the concurrency bug detectors and searches for a thread interleaving that can actually trigger the bug, greatly reducing false positive reports. OWL then passes the actual bug reports to its inter-procedural analyzer, which generates bug-to-attack propagation reports. Finally, OWL's attack input fuzzer takes the propagation reports, runs the test suites, and pinpoints attack-inducing inputs which can trigger attack sites.

Concurrency bug detector (§IV-A) uses existing concurrency bug detection tools. It receives program executables and test suites and produces concurrency bug reports at runtime.

Schedule Reducer (§IV-C) identifies actual concurrency bugs from the bug reports and eliminates false positive reports. It receives concurrency bug reports and a program's LLVM bitcode, and tries to search for a thread interleaving for T_1 and T_2 so that they can actually trigger a data race on the same memory byte.

Inter-procedural Static Analyzer (§V-A) does inter-procedural static analysis to see whether corrupted memory may propagate to any attack sites through data-flow or control-flow. Also, it gives hints for potential inter-thread propagation (e.g., heap overflows). It receives the leftover concurrency bug reports from our Schedule Reducer and the LLVM bitcode, and produces vulnerability reports.

Attack Input Fuzzer (§V-B) takes the bug-to-attack propagation reports and runs a program's self-carrying or third-party test suites. It pinpoints attack-inducing inputs (e.g., AI_x for T_x) which can trigger potentially vulnerable instructions, if so, it records T_x.

Concurrency Attack Verifier (§V-C) takes the inserted breakpoints from the Schedule Reducer, the bug-inducing inputs from concurrency bug reports, and the attack-inducing inputs from the Fuzzer. It replays these taken inputs and sees whether an attack site can be executed. If so, it generates a concurrency attack report.

D. *Detecting Example*

We take CVE-2017-7533 (Fig. 2), a new concurrency attack detected by OWL as an example. When we ran Trinity [48], a `Linux` system call benchmark tool, SKI (a kernel data race detector) generated 24.6K data race reports in total. One of them was a data race between the two system calls `inotify_handle_event()` and `rename()` on the same file. The former system call was invoked by a thread to monitor file modifications, while the latter was invoked by another thread to rename the file to a longer name. Because `Linux` did not properly synchronize the two system calls, a data race occurred and triggered this attack.

OWL's Schedule Reducer found a thread interleaving that triggered a race on a global `file_name` variable shared by both threads. Then, OWL's inter-procedural static analyzer reported that a propagation from the variable to the `src` parameter of the attack site `strcpy()` in `inotify_handle_event()`. OWL viewed this as a potential buffer overflow and invoked attack input fuzzer to search for the extra inputs that can lead to attack.

Firstly, the fuzzer ran the system call `inotify_handle_event()` again and dynamically collected the parameters of `strcpy()`. Meanwhile, memory allocation information was collected and the fuzzer located the `dst` parameter, which was pointed to a memory piece allocated by `kmalloc32()`. Secondly, the fuzzer ran Trinity to generate system calls and recorded the system calls which also used `kmalloc32()` to manage its memory. Leveraging the recorded system calls, we found that only when the `socket()` system call allocated a `netlbl_lsm_secattr` structure repeatedly and resided next to the corrupted `dst` memory address, an overflow happened to overwrite a function pointer in the `netlbl_lsm_secattr` structure. Leveraging the corrupted function pointer, we injected malicious code and successfully got the OS root privilege.

IV. Reducing Schedules

This section presents OWL's schedule reducer component, including automatically annotating adhoc synchronizations (§IV-B) and pruning benign schedules (§IV-C). This component in total greatly reduced 94.1% of the total reports (see §VI-B). Moreover, this section presents OWL's integration with extant concurrency bug detectors (§IV-A).

A. Integration with Concurrency Bug Detectors

OWL has integrated four popular race detectors: Ski, KTSan for Linux kernels and TSan, Valgrind for application programs. To integrate OWL's algorithm (§V-A) with concurrency bug detectors, two elements are necessary for the detectors: the `load` instruction that reads the bug's corrupted memory and the instruction's call stack. We built parsers for each detector to provide uniformed bug reports for OWL. An issue for OWL to work with kernels is that Ski lacks call stack information. We configure Linux kernel with the CONFIG_FRAME_POINTER option enabled. Given a dump of the kernel stack and the values of the program counter and frame pointer, we were able to iterate the stack frames and constructed call stacks.

B. Annotating Adhoc Synchronization

Developers use semaphore-like adhoc synchronizations, where one thread is waiting for a shared variable until another thread sets this variable to be "true". This type of adhoc synchronizations couldn't be recognized by TSan or Ski and caused many false positives.

OWL uses static analysis to detect these synchronizations in two steps. First, by taking the race reports from detectors, it sees if the "read" instruction is in a loop. Then, it conducts an intra-procedural forward data and control dependency analysis to find the propagation of the corrupted variable. If OWL encounters a branch instruction in the propagation chain, it checks if this branch instruction can break the loop. Last, it checks if the "write" instruction assigns a constant to the variable. If so, OWL tags this report as an "adhoc sync".

Compared to the prior static adhoc sync identification method SyncFinder [49], which finds the matching "read" and "write" instruction by statically searching program code, our approach leverages the actual runtime information from the race reports, so ours is much simpler and more precise.

C. Verifying Real Data Races

OWL's schedule reducer also contains a dynamic race verifier to check whether the reduced race reports are indeed real races. The verifier is lightweight because it is built on top of the LLDB debugger. We found that a good way to trigger a data race is to catch it "in the racing moment". The verifier sets thread-specific breakpoints indicated by TSan race reports. "Thread specific" means when the breakpoint is triggered, we only halt that specific thread instead of the whole program. The rest of the threads are still able to run. In this way, we can catch the race when both of the racing instructions are reached by different threads and are accessing the same address.

For each run, OWL verifies one race. Once a data race is verified, the verifier goes one step further. It prints the following dynamic information as security hints including, the racing instructions from source code, the value they're about to read and write and the type of the variable that these instructions are about to read or write. These hints show whether a NULL pointer difference can be triggered or an uninitialized data can be read because of the race.

RaceFuzzer [28] adopts the same core idea of "thread specific breakpoints" and data race verification. OWL's dynamic race verifier provides a lightweight, general, easy to use way (integrated with the existing debuggers) in verifying potentially harmful data races and their consequences.

There are two cases that could cause OWL's race verifier to miss real races. First, if the race detector does not detect the race upfront, the verifier will not report the race either. Second, depending on runtime effects (e.g., schedules), some races can't be reliably reproduced with 100% success rate [50]. Because all implementations of OWL's dynamic verifiers are based on LLDB. For the Linux kernel, our dynamic verifiers can be in practice implemented in QEMU [51].

V. Concurrency Attack Inference

A. Inter-procedural Analysis

Algorithm 1 shows OWL's static inter-procedural analyzer's algorithm. It takes a program's LLVM bitcode in SSA form, an LLVM `load` instruction that reads from the corrupted memory of a concurrency bug report, and the call stack of this instruction. The algorithm then does inter-procedural static analysis to see whether corrupted memory may propagate to any attack site (§III-B) through data or control flows. If so, the algorithm outputs the propagation chain in LLVM IR format as the bug-to-attack hint for developers.

Algorithm 1: Scalable inter-procedural analysis

```
Input : program $prog$, start instruction $si$, $si$ call stack $cs$
Global: corrupted instruction set $crptIns$, vulnerability set $vuls$
DetectAttack($prog$, $si$, $cs$)
    $crptIns$.add $si$
    while $cs$ is not empty do
        $function \leftarrow cs$.pop
        $ctrlDep \leftarrow$ false
        DoDetect($prog$, $si$, $function$, $ctrlDep$)
DoDetect($prog$, $si$, $function$, $ctrlDep$)
    set $localCrptBrs \leftarrow$ empty
    foreach succeeded instruction $i$ do
        bool $ctrlDepFlag \leftarrow$ false
        foreach branch instruction $cbr$ in $localCrptBrs$ do
            if $i$ is control dependent on $cbr$ then
                $ctrlDepFlag \leftarrow$ true
        if $ctrlDep$ or $ctrlDepFlag$ then
            if $i.type() \in vuls$ then
                ReportExploit($i$, CTRL_DEP)
            if $i.type() \in corruptedMemOp$ then
                ReportOverflow($i$, DATA_DEP)
        if $i.isCall()$ then
            foreach actual argument $arg$ in $i$ do
                if $arg \in crptIns$ then
                    $crptIns$.add $i$
                    if $i.type() \in vuls$ then
                        ReportExploit($i$, DATA_DEP)
                    if $i.type() \in corruptedMemOp$ then
                        ReportOverflow($i$, DATA_DEP)
            if $f.isInternal()$ then
                $cs$.push $f$
                DoDetect($prog$, $f$.first(), $f$, $ctrlDep$ or $ctrlDepFlag$)
                $cs$.pop
        else
            foreach $operand\ op\ in\ i$ do
                if $op \in crptIns$ then
                    if $i.type() \in vuls$ then
                        ReportExploit($i$, DATA_DEP)
                    if $i.type() \in corruptedMemOp$ then
                        ReportOverflow($i$, DATA_DEP)
                    $crptIns$.add $i$
                    if $i.isBranch()$ then
                        $localCrptBrs$.add $i$
ReportExploit($i$, $type$)
    if $i$ is never reported on $type$ then
        ReportToDeveloper()
```

The algorithm works as follows. It first adds the corrupted read instruction into a global corrupted instruction set, it then traverses all following instructions in the current function and if any instruction is affected by this corrupted instruction set ("affected" means any operand of current instruction is in this set), it adds the instruction into this corrupted set. The algorithm looks into all successors of branch instructions as well as callees to propagate this set. It reports a potential concurrency attack when an attack site (§III-B) is affected by this set. Moreover, it reports a potential buffer overflow when a memory operation is affected by this set.

To achieve reasonable accuracy and scalability, we made three design decisions. First, based on our finding that bugs and attacks often share similar call stack prefixes, the algorithm traverses the bug's call stack (§II-B). If the algorithm does not find an attack site on the current call stack and its callees, it pops the latest caller in the current call stack and checks the propagation through the return value of this call, until the call stack becomes empty and the traversal of current function finishes. This targeted traversal makes the algorithm scale to large programs with greatly reduced false reports (Table IV).

Second, the algorithm tracks propagation through LLVM virtual registers [12]. Similar to relevant systems [22], [32], our design did not incorporate pointer analysis [52], [53] because one main issue of such analysis is that it typically reports too many false positives on shared memory access in large programs.

Our analyzer compensates the lack of pointer analysis by: (1) tracking read instructions in the detectors at runtime (§IV-A), and (2) leveraging the call stacks to precisely resolve the actually invoked function pointers (another main issue in pointer analysis).

Third, some detectors do not have read instructions in the reports (e.g., write-write races), and we modified the detectors to add the first `load` instruction for these reports during the detection runs (§IV-A).

All five types of vulnerability sites we found (§III-B) have been incorporated in this algorithm. The generated attack site reaching branches from this algorithm serves as bug-to-attack propagation hints and helped us identify subtle inputs to detect five new attacks and seven known ones (§VI-A).

B. Attack Input Fuzzing

A main challenge for OWL's attack input fuzzer is how to check whether the memory corrupted by the bug triggering threads can infect other threads and lead to attacks. For instance, in our model (§III-B), given that T_1 and T_2 have corrupted global memory with a concurrency bug, our fuzzer's goal is to find (T_x, AI_x), which can leverage the corrupted memory to hit an attack site. However, checking whether two threads can violate a certain property is NP-hard [54] and unpractical to academia and industry attack detection. To the best of our knowledge, no existing fuzzer is designed to tackle this challenge due to two reasons (§II-C): (1) no existing fuzzer (e.g., Driller [55] and UC-KLEE [11]) considers a memory corruption model with concurrent threads, and (2) their symbolic execution techniques often scale poorly on the inter-procedural bug-to-attack propagation paths.

To mitigate this challenge, OWL's attack input fuzzer takes the bug-to-attack propagation hints from its analyzer (§V-A), and infers potential memory areas that may be overflowed by memory corruption operations. Then, the fuzzer runs a program's own or third-party test suites to pinpoint extra threads and their attack inputs, which can allocate memory blocks near the corrupted memory blocks.

In an implementation level, OWL's fuzzer consists of three steps for each program. First, given the bug-to-attack reports (LLVM instructions) generated from OWL's inter-procedural analyzer, the fuzzer collects the list of static corrupted memory instructions that manipulate corrupted memory or are infected by corrupted memory.

The second step is a replay. The fuzzer re-runs the test suites, including the inputs that trigger the concurrency bug.

It records the lifecycles of all memory blocks managed by `alloc()` and `de-alloc()` operations for all threads of the program. During the replay, for the memory blocks which are infected (written) by the corrupted instructions collected from the first step, the fuzzer marks these blocks as "corrupted".

Third, the fuzzer traverses all the memory blocks B without the "corrupted" mark. If B is next to a "corrupted" block and a thread T_x reads from B, the fuzzer reports this thread T_x and its input AI_x as vulnerable. Finally, (T_1, BI_1), (T_2, BI_2) and (T_x, AI_x) will be fed to OWL's attack verifier (§V-C) to search for a thread interleaving to conduct a concurrency attack.

OWL's fuzzer supports both userspace programs and `Linux` kernel. We leveraged two `Linux` tools, `Kprobe` for kernel and `Uprobe` for userspace, to achieve dynamic tracing of functions. Our evaluation (§VI-C) shows that the fuzzer is effective in finding potential attack-inducing inputs from a large number of test cases generated by test suites.

C. Concurrency Attack Verifier

OWL's verifier is built on LLDB so it is lightweight. It takes the bug-inducing inputs and attack-inducing inputs from the fuzzer (§V-B), replays the inserted breakpoints from the schedule reducers (§IV-C), and then checks whether an attack site is hit. If so, it outputs a concurrency attack report.

VI. Evaluation

We evaluated OWL on six widely used C/C++ programs and used common test suites of these programs as workloads (Table II). Our evaluation was done on a machine with a 2.60 GHz 24 hyper-threading cores Intel Xeon CPU, 64 GB memory, and 1TB SSD, running `Linux`-4.10.0-35-generic.

Table II: Test suites for the six programs. The test suites we used for generating race reports and testing efficiency and performance of OWL. In addition to test suites, we also involve all the 12 exploitation scripts as test cases.

Name	Test Suites
`Linux`	Trinity (Syscall bench)
`SSDB`	SSDB-bench
`Libsafe`	Attack exploit script
`MySQL`	DBT2 Benchmark Tool
`Chrome`	Octane 2.0
`Apache`	Ab (Apache bench)

We focused our evaluation on four key questions:

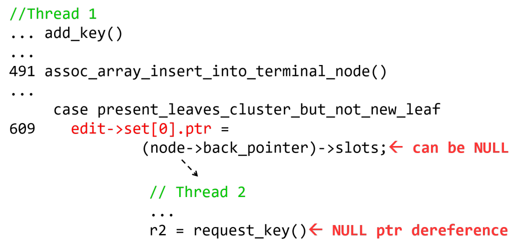
- Can OWL detect new and known concurrency attacks in real-world programs? (§VI-A)
- How many false-positive reports from concurrency bug detection tools can OWL reduce? (§VI-B)
- Can OWL infer the extra attack inputs? (§VI-C)
- How much time does OWL cost? (§VI-D)

A. Detecting New and Known Concurrency Attacks

OWL detected five new concurrency attacks listed in Table III. For three out of five new attacks, both the concurrency bugs and the concurrency attack consequences are new. One attack is CVE-2017-7533 [10], which caused a `Linux` OS root privilege escalation. This attack has been confirmed and fixed by RedHat developers immediately after we reported it. `Android` also gave us a $2500 reward for reporting this severe attack. The other two attacks (CVE-2017-12193 [18] and CVE-2016-1000324 [17]) have been confirmed and fixed by RedHat and `SSDB` developers respectively. Surprisingly, although the two old concurrency bugs `Apache`-25520 and `Apache`-46215 have been reported over years and well studied, OWL still detected two new concurrency attack consequences on these bugs: one attack is HTML integrity violation and the other is an integer overflow with a DoS attack.

Currently, OWL focuses on practical and scalable detection for concurrency attacks, and its inter-procedural analysis (§1) may miss real attacks. To evaluate whether OWL may miss attacks, we applied OWL on 7 known concurrency attacks listed in Table III. OWL detected all these known attacks.

The first new concurrency attack (CVE-2017-7533 [10], shown in Fig. 2) detected by OWL is caused by a new data race between `inotify_handle_event()` and `rename()`. Our static analyzer (§V-A) reported that the data race may cause a buffer overflow. We used Attack Input Fuzzer, which gave nine potential victim system calls and their allocated memory could be overflowed. After manually analyzing the nine victim inputs, we found a specific `bind()` system call can generate a SELinux structure `netlbl_lsm_secattr`, which has a function pointer `void (*free)` in the front. Destroying the socket leads to dereference of the function pointer `void (*free)`. We overwrote this function pointer to point to a piece of malicious code `mmap()` in kernel space and successfully got an OS root privilege.



```
//Thread 1
... add_key()
...
491 assoc_array_insert_into_terminal_node()
...
    case present_leaves_cluster_but_not_new_leaf
609   edit->set[0].ptr =
              (node->back_pointer)->slots;← can be NULL

              // Thread 2
              ...
              r2 = request_key()← NULL ptr dereference
```

Figure 5: A NULL pointer dereference and DoS attack detected by OWL in `Linux` kernel, confirmed as CVE-2017-12193 [18].

The second new concurrency attack (shown in Fig.5) was reported by OWL as a potential NULL pointer dereference attack, caused by two threads both running `request_key()` system calls. OWL first received a data race report on the `node->back_pointer`

Table III: OWL*'s detection results on concurrency attacks.* With the listed subtle inputs, all these attacks were often triggered within 20 repeated queries or loops except for the `Apache` one and `Linux` 4.11.9 one. The three new attacks detected by OWL in `Linux` 4.11.9, `Linux` 4.12.1 and `SSDB` 1.9.2 have been confirmed as CVE-2017-7533, CVE-2017-12193, and CVE-2016-1000324.

Attack Name	Software Version	Vulnerability Type	Need Subtle Attack Input?	New Bug?	New Attack?
CVE-2017-7533	`Linux`-4.11.9	Privilege escalation	Three syscalls	New	New
CVE-2017-12193	`Linux`-4.12.1	NULL pointer deref	Repeated `add_key()`	New	New
CVE-2016-1000324	`SSDB`-1.9.2	Use after free	No need	New	New
`Apache`-25520	`Apache`-2.0.48	Integrity violation	Loop with two log requests	Known	New
`Apache`-46215	`Apache`-2.2.10	Integer overflow	Decreasing worker threads	Known	New
CVE-2004-1235	`Linux`-2.6.10	NULL pointer deref	Syscall parameters	Known	Known
CVE-2009-1527	`Linux`-2.6.29	Privilege escalation	Syscall parameters	Known	Known
CVE-2010-3412	`Chrome`-6.0.472.58	Use after free	No need	Known	Known
CVE-2015-1125	`Libsafe`-2.0-16	Buffer overflow	Loops with `strcpy()`	Known	Known
`MySQL`-24988	`MySQL`-5.0.27	Access permission	FLUSH_PRIVILEGES	Known	Known
`MySQL`-35589	`MySQL`-5.1.35	Double free	SET_PASSWORD	Known	Known
`Apache`-21287	`Apache`-2.0.48	Double free	PhP queries	Known	Known

and OWL's inter-procedural analyzer reported a potential pointer dereference on the `node->back_pointer`. With the report provided by OWL, we found that the `node->back_pointer` can be set to NULL without checking. Hence we conducted a DoS attack based on this NULL pointer dereference. This attack is confirmed as CVE-2017-12193 [18] by RedHat and immediately fixed by `Linux` Kernel team.

```
// Thread 1
355 log_clean_thread_func(void *arg){
356   BinlogQueue *logs =
...     (BinlogQueue *)arg;
357
358   while(!logs->thread_quit){
359     if(!Logs->db){ - - - - - - - - - ->
360       break;
361     }
...
371     logs->del_range(start, end);
375   }
380 }

341 int del_range(...){
342   while(start <= end){
347     Status s = db->Write(...); ←Attack Site
351   }
```

```
// Thread 2
190 ~BinlogQueue(){
...
200     db = NULL;
201 }
```

Figure 6: A new concurrency bug and use-after-free attack detected by OWL in `SSDB`-1.9.2, confirmed as CVE-2016-1000324 by `SSDB` developers in their emails [17].

The third new concurrency attack (CVE-2016-1000324 [17]) detected by OWL is caused by a new data race and a new use-after-free attack in `SSDB`. Fig. 6 shows the details of this vulnerability. During a server shut-down, `SSDB` uses adhoc synchronization to synchronize among threads. However, it is possible that line 359 is executed before line 200. This race causes `log_clean_thread_fun` failing to break out from the while loop. Moreover, `log_clean_thread_fun` could execute `del_range` which could use `db` and cause a use-after-free. Furthermore, line 347 contains a function pointer dereference which could cause log corruption or program crash if the memory area was shared and reused by other threads.

OWL's static analyzer (§V-A) identified the vulnerability site at line 347 because it contains a pointer dereference. This site is control-dependent on the corrupted branch at line 359. OWL's dynamic vulnerability verifier (§V-C) further verified that another thread would free the memory area and set the pointer to `NULL` before the dereference within this thread. We reported the race and its consequential attack to `SSDB` developers and got confirmed.

```
1327 ap_buffered_log_writer(void *handle, ...)
...
1334 {
1335   char *str;
1336   char *s;
1337   int i;
1338   apr_status_t rv;
1339   buffered_log *buf = (buffered_log*)handle;
...
1342   if (len + buf->outcnt > LOG_BUFSIZE) {
1343       flush_log(buf);
1344   }
...
1357   else {
1358    for(i=0,s=&buf->outbuf[buf->outcnt];i<nelts;++i) {
1359      memcpy(s,strs[i], trl[i]);← vulnerable site
1360      s += strl[i];
1361    }
1362    buf->outcnt += len;
1363    rv = APR_SUCCESS;
1364   }
...
1366 }
```

Figure 7: A new HTML integrity violation attack exploited by OWL based on known bugs in `Apache`-2.0.48. Although the bug has been reported and studied over years, we are the first to exploit an attack based on this bug.

The fourth new attack stems from a known data race in `Apache`. This attack could write `Apache`'s own request logs into other users' HTML files stored in `Apache`, causing a HTML integrity violation and information leak. Fig. 7 shows the code of this vulnerability from the Apache-25520 bug [56]. `buf->outcnt` is shared among threads and serves as an index of a buffer array. Lacking proper synchronization when modifying this variable at line 1362, a data race occurred and caused the server to write wrong contents to `buf->outbuf`.

Worse, the wrong contents could also overflow `buf->outbuf` and cause a buffer overflow. Even worse, `Apache` stores the file descriptor of its HTTP request log next to `buf->outbuf`. We constructed a one-byte overflow of `buf->outbuf`, successfully corrupted this file descriptor, and made `Apache`'s own HTTP request logs written to an HTML file with the corrupted value of this file descriptor.

Although this data race has been well studied by researchers [57], people thought the worst consequence of this bug might just be corrupting `Apache`'s own request log. We are the first to detect this HTML integrity violation attack with OWL and the first to construct the actual exploit scripts.

OWL's vulnerability analysis (§V-A) pinpointed the vulnerable site at line 1359 and inferred that this line is data-dependent on the corrupted variable at line 1358. OWL's dynamic race verifier (§IV-C) triggered the race and showed how many bytes in `buf->outbuf` were overflowed.

```
size_t busy;  /* busyness factor */

// Thread 1
588 static int proxy_balancer_post_request(…)
...
616 if (worker && worker->s->busy)
617   worker->s->busy--;
                              // Thread 2
                              616 if (worker && worker->s->busy)
                              617   worker->s->busy--;

      // Thread 3
      1138 static proxy_worker *find_best_bybusyness(...)
      ...
      1144     proxy_worker *mycandidate = NULL;
      ...
      1192 if (!mycandidate
      1193     || worker->s->busy < mycandidate->s->busy
      1194     || ...
      1195   mycandidate = worker; ← vulnerable site
```

Figure 8: ***A new integer overflow and DoS attack exploited by* OWL *based on a known bug* `Apache`*-46215.*** The bug is reported over years but OWL first found it could be utilized to trigger attacks.

The last new concurrency attack is an integer overflow DoS attack based on a known `Apache`-46215 data race. Fig. 8 shows the `Apache`-46215 bug [58]. Each `Apache` worker thread contains a field `worker->s->busy` indicating its status (busy or not). An `Apache` load balancer component contains threads to concurrently increment or decrement these flags for worker threads when they start or finish serving requests. However, as shown at line 616, this is a data race because developers forgot to use a lock during the counter increment and decrement.

Over years, this status counter has been viewed as statistic information and its data race does not matter much extent. Unfortunately, this counter is an unsigned integer, and an integer overflow could be triggered during the decrement. In some cases, the counter could be overflowed and become the largest unsigned integer (i.e., marking a thread the "busiest" one). The check at line 617 can be easily bypassed because of the race. Since load balancer assigns future requests based on the worker threads' counters, arbitrary worker threads in `Apache` can be viewed as the busiest ones and be completely ignored, causing a DoS attack on these threads and a significant downgrade of `Apache`'s throughput.

OWL detected this concurrency attack as follows. OWL's race detector detected a race between line 617 and line 1192. OWL's dynamic race verifier reported a detailed dynamic race information including the racing instructions, the value they could read or write to the variable, and the types of the variables. We then found `worker->s->busy` in some worker threads had an overflowed value: `018, 446, 744, 073, 709, 551, 614`. OWL's vulnerability analysis (§V-A) reported that a pointer assignment could be control-flow dependent on the corrupted branch at line 1192. OWL's vulnerability verifier verified that the branch was indeed corrupted and line 1195 was reachable.

The above five new concurrency attacks were overlooked by prior reliability and security tools mainly due to three reasons. First, compared to OWL's reduced vulnerable reports, existing concurrency bug detector generate at least 71X more data race reports in `Apache` and 14X more reports in `Linux`. Developers' burden is heavy using existing concurrency bug detection tools, because diagnosing all these reports is just like finding needles in a haystack.

Second, for both the new and known concurrency attacks we evaluated, OWL's inter-procedural analysis (§V-A) precisely pinpoints bug-to-attack propagation across different functions and threads. In contrast, existing concurrency bug consequence analysis tools only provide intra-procedural analysis and they only analyze the explicit consequences, thus they have not reported any concurrency attack we evaluated in this paper.

Third, to the best of our knowledge, none of the existing tools is designed to infer the extra attack-inducing inputs. Therefore, even if a concurrency bug and its concurrency attack is within the same function, it is still hard for existing tools to get the attack inducing inputs.

B. Reducing False-positive Race Reports

Table IV shows OWL's results. The third column indicates the number of raw reports generated by our race detectors. The fourth and fifth columns show the number of the remaining reports after static analysis and further after dynamic verification respectively. Overall, OWL is able to prune 93.0% schedule cases of false positives in Linux kernel and 97.7% for the other applications. This significant reduction will help developers save much diagnostic time. Except for `Linux`, the total number of actual bug reports (the "**# reduced r.**" column in Table IV) is up to 126, which saves much time for OWL's analysis (confirmed in Fig. 9). Two programs `Libsafe` and `SSDB` did not generate many race reports because their multi-threading model is simple.

C. Find Extra Attack Inputs

As shown in Table V, OWL's Attack Input Fuzzer is effective in finding potential attack-inducing inputs from a large number of test cases generated by test suites. Specifically, Attack Input Fuzzer works the best for `Linux` kernel, which greatly reduces the number of test cases from 1153 to 29. The reason is that `Linux` diverges its `kmalloc()` to `kmalloc32()`, `kmalloc64()`, etc, and memory allocated by the same type of `kmalloc()` resides on the same kernel heap slab. Moreover, the concurrency bugs reported by SKI only require `kmalloc32()` to conduct the attack and other memory allocation sites (e.g., `kmalloc64()`) are

Table IV: OWL *race report reduction results.* We selected 12 attacks whose bugs have been triggered on our machine. OWL detects all these attacks. In this table, **LoC** represents the number of lines of code in each program; **# race r.** represents the number of raw race reports generated; **# reduced r.** represents the number of reduced reports after Schedule Reducer (VI-B); **# final r.** represents number of final reports given by OWL; **# atks** stands for the number of actual concurrency attacks we have found so far.

Name	LoC	# race r.	# reduced r.	# final r.	# atks
`Apache`	290K	715	10	10	3
`Linux`	2.8M	24645	1722	36	4
`Chrome`	3.4M	1715	126	115	1
`Libsafe`	3.4K	3	3	3	1
`MySQL`	1.5M	1123	18	16	2
`SSDB`	67K	12	2	2	1
Total	5.36M	31874	1885	182	12

filtered by the fuzzer. Finally, the fuzzer found an socket call with an SELinux label contributed to the new `Linux` OS root privilege escalation attack (§VI-A).

Table V: OWL's Attack Input Fuzzer and its fuzzing results. Type means the type of memory allocation functions which allocate the overflowed global memory. **# Test Cases** is the number of the inputs of Attack Input Fuzzer. **# Extra Inputs** is the number of the outputs of OWL's fuzzer. **# Atks** is the number of actual attacks that requires subtle attack inputs (10 out of 12).

Name	Type	# Test cases	# Attack inputs	# Atks
`Apache`	apr_palloc	243	58	3
`Linux`	kmalloc	1153	29	4
`Chrome`	partalloc	432	123	0
`Libsafe`	malloc	4	4	1
`MySQL`	sql_alloc	814	409	2
`SSDB`	malloc	2	2	0
Total	n/a	2648	625	10

D. OWL*'s Performance*

OWL contains two major phases: concurrency bug report reduction is to reduce the false positive reports and identify the actual bug reports; attack input inference is to infer all the attack inducing inputs. This section evaluates the effectiveness of the first phase on reducing diagnosis time cost and overall time cost of OWL, with two plans: (1) direct feeding all raw data race reports to the concurrency attack inference phase of OWL and measuring its time cost; (2) measuring the end-to-end time cost of OWL (Fig. 9).

Overall, plan 2 spends 24.5X less time cost than plan 1. This implies that the concurrency bug reduction phase in OWL can greatly reduce the false positive reports and save much time for program developers. Moreover, let's consider only the time cost of plan 2, which is OWL's time cost. OWL consumes the largest amount of time for `Linux` (2548 minutes). This time cost is reasonable for in-housing testing because OWL processed a large `Linux` test suite generated by Trinity. For the other five programs, OWL's time cost is even smaller.

VII. Discussion and Conclusion

OWL's main design choice is to find new concurrency attacks with reasonable accuracy and scalability, and it trades

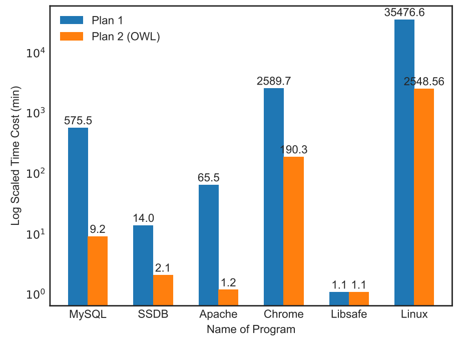


Figure 9: OWL's reduction on race reports saves further diagnosis time. We measured and compared the time cost of feeding all raw data race reports and the reduced reports into OWL's second phase. The y-axis is in *log* scale.

off soundness (i.e., do not miss any attacks). Also, due to the lack of domain knowledge and semantic, as well as the huge analysis efforts needed, it is hard to verify any false negatives in all produced reports. Other concurrency bug detection systems (e.g., ConMem [4], RaceMob [25] and RacerX [23]) also made the same design choice. Typical way to ensure soundness is to plug in a sound alias analysis tool [52], [53] to identify all LLVM `load` and `store` instructions that may access the same memory. However, typical alias analyses are known to be inaccurate (e.g., too many false positives).

OWL's inter procedural analysis tool integrates the call stack of a concurrency bug to direct static analysis toward vulnerable program paths, but OWL's vulnerable propagation path reports (§V-A) may contain false positives (e.g., the reports may contain non-vulnerable instructions). In our evaluation, we found that these propagation reports are quite precise because OWL reported only a small number of final hints (Table IV) and they are informative as they helped us identify subtle inputs for real attacks (§VI-A).

In conclusion, we have presented the first quantitative study on real-world concurrency attacks and OWL, the first detection tool to effectively detect them. OWL accurately detects a number of new concurrency attacks in large, widely used programs. We believe that OWL will attract more attention for detecting, diagnosing, fixing, and defending against concurrency attacks.

Acknowledgment

Thank all anonymous reviewers for their valuable comments. The work is funded in part by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (HK RGC ECS No.27200916, HK RGC GRF No.17207117), the Croucher Innovation Award, NSF CNS-1564055, ONR N00014-16-1-2263, and ONR N00014-17-1-2788 grants.

References

[1] S. Sender and A. Vidergar, "Concurrency attacks in web applications," Blackhat '08, 2008.

[2] Z. Wu, K. Lu, X. Wang, and X. Zhou, “Collaborative technique for concurrency bug detection,” *International Journal of Parallel Programming*, vol. 43, no. 2, pp. 260–285, 2015.

[3] 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 *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP ’07)*, 2007, pp. 103–116.

[4] W. Zhang, C. Sun, and S. Lu, “ConMem: detecting severe concurrency bugs through an effect-oriented approach,” in *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’10)*, Mar. 2010, pp. 179–192.

[5] P. B. Todd Warszawski, “Acidrain: Concurrency-related attacks on database-backed web applications,” in *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. ACM, 2017, pp. 5–20.

[6] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, “Concurrency attacks,” in *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR ’12)*, Jun. 2012.

[7] “Cve-2004-1235,” https://www.exploit-db.com/exploits/895/.

[8] “MSIE javaprxy.dll COM object exploit,” http://www.exploit-db.com/exploits/1079/.

[9] “CVE-2010-0923,” http://www.cvedetails.com/cve/CVE-2010-0923.

[10] “CVE-2017-7533,” http://seclists.org/fulldisclosure/2017/Aug/6.

[11] D. A. Ramos and D. R. Engler, “Practical, low-effort equivalence verification of real code,” in *Proceedings of the 23rd International Conference On Computer Aided Verification (CAV ’11)*, 2011, pp. 669–685.

[12] “The LLVM compiler framework,” http://llvm.org, 2013.

[13] “Threadsanitizer,” https://code.google.com/p/data-race-test/wiki/ThreadSanitizer, 2015.

[14] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI ’07)*, Jun. 2007, pp. 89–100.

[15] “Kernelthreadsanitizer, a fast data race detector for the linux kernel,” https://github.com/google/ktsan.

[16] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, “Ski: Exposing kernel concurrency bugs through systematic schedule exploration.” in *OSDI*, 2014, pp. 415–431.

[17] “CVE-2016-1000324,” https://github.com/dsn18-paper46/owl/cve-2016-1000324.pdf.

[18] “CVE-2017-12193,” http://seclists.org/oss-sec/2017/q4/181.

[19] N. G. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, “Eraser: A dynamic data race detector for multithreaded programming,” *ACM Transactions on Computer Systems*, pp. 391–411, Nov. 1997.

[21] Y. Yu, T. Rodeheffer, and W. Chen, “RaceTrack: efficient detection of data race conditions via adaptive tracking,” in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP ’05)*, Oct. 2005, pp. 221–234.

[22] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, “ConSeq: detecting concurrency bugs through sequential errors,” in *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’11)*, Mar. 2011, pp. 251–264.

[23] D. Engler and K. Ashcraft, “RacerX: effective, static detection of race conditions and deadlocks,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP ’03)*, Oct. 2003, pp. 237–252.

[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou, “AVIO: detecting atomicity violations via access interleaving invariants,” in *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’06)*, Oct. 2006, pp. 37–48.

[25] B. Kasikci, C. Zamfir, and G. Candea, “Racemob: crowdsourced data race detection,” in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM, 2013, pp. 406–422.

[26] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, “Parallelizing data race detection,” in *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’13)*, Mar. 2013, pp. 27–38.

[27] B. C. C. Kasikci, “Techniques for detection, root cause diagnosis, and classification of in-production concurrency bugs,” 2015.

[28] K. Sen, “Race directed random testing of concurrent programs,” in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI ’08)*, Jun. 2008, pp. 11–21.

[29] S. Park, S. Lu, and Y. Zhou, “CTrigger: exposing atomicity violation bugs from their hiding places,” in *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’09)*, Mar. 2009, pp. 25–36.

[30] C.-S. Park and K. Sen, “Randomized active atomicity violation detection in concurrent programs,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT ’08/FSE-16)*, Nov. 2008, pp. 135–145.

[31] M. Attariyan, M. Chow, and J. Flinn, “X-ray: Automating root-cause diagnosis of performance anomalies in production software.” in *OSDI*, vol. 12, 2012, pp. 307–320.

[32] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, “Failure sketching: A technique for automated root cause diagnosis of in-production failures,” in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP ’15)*, Oct. 2015.

[33] J. Wu, H. Cui, and J. Yang, “Bypassing races in live applications with execution filters,” in *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI ’10)*, Oct. 2010.

[34] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, “Automated concurrency-bug fixing.” in *OSDI*, vol. 12, 2012, pp. 221–236.

[35] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 590–604.

[36] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, “A security policy oracle: Detecting security holes using multiple api implementations,” *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 343–354, 2011.

[37] X. Zhang, A. Edwards, and T. Jaeger, “Using CQUAL for static analysis of authorization hook placement,” in *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002, pp. 33–48.

[38] R. Paleari, D. Marrone, D. Bruschi, and M. Monga, “On race vulnerabilities in web applications,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 126–142.

[39] V. B. Livshits and M. S. Lam, “Finding security errors in Java programs with static analysis,” in *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005, pp. 271–286.

[40] “Libsafe - Safety Check Bypass Vulnerability,” http://www.securityfocus.com/archive/1/395999. [Online]. Available: http://www.securityfocus.com/archive/1/395999

[41] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *PLDI ’08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008, pp. 206–215.

[42] C. Cadar, D. Dunbar, and D. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI ’08)*, Dec. 2008, pp. 209–224.

[43] 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 (CCS ’06)*, Oct.–Nov. 2006, pp. 322–335.

[44] D. A. Ramos and D. R. Engler, “Under-constrained symbolic execution: Correctness checking for real code.” in *USENIX Security Symposium*, 2015, pp. 49–64.

[45] C. Sapuntzakis, “Personal communication,” Apr. 2000, bug in OpenBSD where an interrupt context could call blocking memory allocator.

[46] M. Castro, M. Costa, and J.-P. Martin, “Better bug reporting with better privacy,” in *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’08)*, Mar. 2008, pp. 319–328.

[47] H. Cui, G. Hu, J. Wu, and J. Yang, “Verifying systems rules using rule-directed symbolic execution,” in *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’13)*, 2013.

[48] “Trinity: A Linux System Call Fuzzer,” https://github.com/kernelslacker/trinity.

[49] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, “Ad hoc synchronization considered harmful,” in *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI ’10)*, Oct. 2010.

[50] J. E. Gottschlich, G. A. Pokam, C. L. Pereira, and Y. Wu, “Concurrent predicates: A debugging technique for every parallel programmer,” in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 331–340.

[51] http://www.qemu.org.

[52] J. Whaley, “bddbddb Project,” http://bddbddb.sourceforge.net.

[53] C. Lattner, A. Lenharth, and V. Adve, “Making context-sensitive points-to analysis with heap cloning practical for the real world,” in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI ’07)*, 2007.

[54] R. Jhala and R. Majumdar, “Software model checking,” *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[55] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.” in *NDSS*, vol. 16, 2016, pp. 1–16.

[56] “Apache bug 25520,” https://bz.apache.org/bugzilla/show_bug.cgi?id=25520.

[57] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS ’08)*, Mar. 2008, pp. 329–339.

[58] “Apache bug 46215,” https://bz.apache.org/bugzilla/show_bug.cgi?id=46215.