GPU Taint Tracking

Ari B. Hayes, *Rutgers University;* Lingda Li, *Brookhaven National Laboratory;*
Mohammad Hedayati, *University of Rochester;* Jiahuan He and Eddy Z. Zhang,
Rutgers University; Kai Shen, *Google*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/hayes

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

GPU Taint Tracking

Ari B. Hayes
Rutgers University

Lingda Li[*]
Brookhaven National Lab

Mohammad Hedayati
University of Rochester

Jiahuan He
Rutgers University

Eddy Z. Zhang
Rutgers University

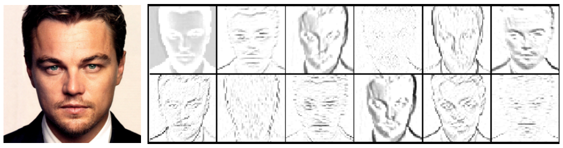
Kai Shen
Google

Abstract

Dynamic tainting tracks the influence of certain inputs (taint sources) through execution and it is a powerful tool for information flow analysis and security. Taint tracking has primarily targeted CPU program executions. Motivated by recent recognition of information leaking in GPU memory and GPU-resident malware, this paper presents the first design and prototype implementation of a taint tracking system on GPUs. Our design combines a static binary instrumentation with dynamic tainting at runtime. We present new performance optimizations by exploiting unique GPU characteristics—a large portion of instructions on GPU runtime parameters and constant memory can be safely eliminated from taint tracking; large GPU register file allows fast maintenance of a hot portion of the taint map. Experiments show that these techniques improved the GPU taint tracking performance by 5 to 20 times for a range of image processing, data encryption, and deep learning applications. We further demonstrate that GPU taint tracking can enable zeroing sensitive data to minimize information leaking as well as identifying and countering GPU-resident malware.

1 Introduction

GPUs have been widely used in many important application domains beyond scientific computing, including machine learning, graph processing, data encryption, computer vision, etc. Sensitive information propagates into GPUs and, while being processed, leaves traces in GPU memory. For example, in a face recognition application, besides the input photo itself, the features extracted at different levels of the deep learning neural networks may also contain part of sensitive or private information. Figure 1 shows the extracted features from the first level of neural networks in a face recognition program, where Figure 1(a) is the original picture and Figure 1(b) are features such as silhouette of a human face. Given a sensitive input user photo, features in deep learning applications may contain much of the sensitive data as well. Other sensitive data in today's GPU applications include encryption keys, digits in personal checks, license plates, location information in virtual reality apps, etc. If not tracked or protected, sensitive information can be inadvertently leaked or stolen by malicious applications on GPUs.



(a) Org. Photo (b) Extracted Features

Figure 1: Neural network information leaking example.

Taint analysis [3, 4, 6, 11, 22, 26, 29, 30] is a powerful tool for information flow tracking and security. It tracks where and how sensitive information flows during program execution. Taint analysis is a form of data flow analysis, wherein an input set of sensitive data is marked as "tainted", and this taint is tracked during runtime as it spreads into different locations in memory via move, arithmetic, and control operations. Taint analysis results can be used to protect data by clearing tainted variables at the end of its life range—for instance, the temporary key schedule at every round of AES algorithm—or by encrypting live but inactive tainted data [27]. Taint analysis can also help identify and counter abnormal behaviors of malicious malware. Existing dynamic taint analysis has primarily been applied to CPU programs though its functions are increasingly desirable for GPUs as well.

This paper presents the first design and implementation of a GPU taint tracking system. Our approach is based on static binary instrumentation that enables dynamic taint tracking of a GPU program. In comparison

[*] This work was done when Lingda Li was a postdoctoral associate at Rutgers University

to dynamic instrumentation that captures and modifies instructions on the fly, our approach does not require a dynamic instrumentation framework or virtual machine emulation that is not readily available on GPUs. We perform static instrumentation on GPU program binaries without source access so that it is easy to apply in practice. We instrument programs on a per-application basis and when the program runs, every thread can dynamically track information flow by itself.

The major challenge for efficient taint tracking is that tracking every dynamic binary instruction is expensive. Our solution exploits the fact that a large portion of a typical GPU program execution operates on un-taintable runtime parameters and constants. Examples include the logical thread indexes, thread block identifiers, dimension configurations, and pointer-type kernel parameters. We use a simple filtering policy that the runtime taint tracking only operates on instructions whose operands can be reached from potential global memory taint sources through dependencies and can reach potential global memory taint sinks. We present an iterative two-pass taint reachability analysis to implement such instruction filtering which significantly reduces runtime taint tracking costs.

Our taint tracking system also exploits the heterogeneous memory architecture on GPUs. A GPU has different types of memory, including either physically partitioned or logically partitioned memory storage. For instance, local memory is private to every thread, shared memory is a software cache visible to a group of threads, and global memory is visible to all threads. Our taint system handles different types of memory storage separately and optimizes the tracking for different types of memory storage. Specifically, we allocate a portion of the register file to store part of the taint map, since GPU contains a much larger register file than CPU—e.g., every streaming multi-processor (SM) has 64K registers on most NVIDIA GPUs. Not all registers are needed [8, 20] nor the maximum occupancy is necessary [10] for best performance. Using fast access registers to maintain the taint map of frequently accessed data will improve the dynamic tainting performance.

GPU taint tracking enables data protection that clears sensitive (tainted) data objects at the end of their life range as well as detects leak of the sensitive data in the midst of program execution. We recognize that data in different GPU memory storage may have different life ranges. For instance, registers and local memory are thread private and can be cleared once a thread finishes its execution; shared memory is only used by a thread block and sensitive data in shared memory can be cleared by that thread block once it releases the SM. Global memory may be accessed at any time of a program run so we cannot clear it at the end of every kernel execution. However, we can detect when and where the sensitive information (in global memory) is sent out by instrumenting memory communication APIs since all communication between GPU, CPU and other network devices require explicit memory API calls. By checking if the sensitive information falls within the region of memory that is transferred, we can identify GPU malware (like Keylogger [17] and Jellyfish [15]) that uses GPU to snoop CPU activities while storing these activities in GPU memory. Such GPU-resident malware would escape detection by a CPU-only taint tracking mechanism.

2 Background

GPU functions, also known as kernel functions, make use of memory which is not directly accessible from the CPU. GPU memory is split into several regions, both on-chip and off-chip. On-chip memory consists of registers, caches, and scratch-pad memory (called *shared memory* in NVIDIA terminology). Note that we use NVIDIA terminology throughout this paper. Off-chip memory is GDDR SGRAM, which is logically distributed into texture memory, constant memory, local memory, and global memory, with texture memory and constant memory mapped to texture and constant caches.

Both texture memory and constant memory are read-only during the GPU kernel execution. Therefore, in this paper we focus on registers, shared memory, local memory, and global memory. Shared memory is available to the programmer, often treated as a software cache. Local memory is thread-private, and is most commonly used for register spilling. Global memory is visible to the entire GPU device, and is typically used as input and output for GPU functions. In all four of these memory types, data persists after deallocation [25].

Global memory can be set and cleared through API functions, with overhead similar to that of running a GPU kernel, but local memory, shared memory, and registers are only accessible from within a kernel function, and allocated and deallocated by the driver. These three memory types can only be reliably cleared through instrumentation. Moreover, local memory and registers are managed by compilers and they can only be cleared by compile-time instrumentation.

Sensitive information can also propagate to different data storage locations on GPU: memory, software caches, and registers. An example is the advanced encryption standard (AES), in which the key and the plain text to be encrypted may reside in different types of memory [25]. They can be stored in global memory as allocated data objects and in registers as program execution operands.

Currently, there is less memory protection on GPUs as compared with CPUs. When two applications run simultaneously on the same GPU with the Multi-process Service (MPS), one application can peek into the memory of another application, documented in NVIDIA's MPS manual at Section 2.3.3.1, "*An out-of-range read in a CUDA Kernel can access CUDA-accessible memory modified by another process, and will not trigger an error, leading to undefined behavior*." When two applications do not run simultaneously, in which case every application will get a serially scheduled time-slice on the whole GPU, information leaking is still possible. The second running application can read data left by the first running application if its allocated memory locations happen to overlap with those of the first one. This vulnerability has been detailed in several recent works [18, 25, 28].

Future hardware trends such as the fine-grained memory protection in AMD APUs suggest potentially better process isolation. Hardware-level memory protection may exhibit superior performance, but its realization must take into account the hardware implementation complexity. And more importantly, process memory protection does not distinguish sensitive data and its propagation within one program or process. Such protection would be critical for securing sensitive information flows between CPU, GPU and their memories.

3 Efficient GPU Taint Tracking

A typical information flow tracking system on CPUs monitors instructions and operands to maintain proper taint propagations. For example, in a binary operation $v = binop\ v_1, v_2$, assuming $T(v_1)$, $T(v_2)$, and $T(v)$ represent the taint status for operands v, v_1, and v_2 respectively: true means tainted and false means untainted. The taint tracking rule for this instruction is $T(v) = T(v_1)$ $||$ $T(v_2)$. Taint statuses for all data storage locations (program memory, registers, conditional flags, etc.) are maintained in a taint map in memory. A baseline GPU taint tracking system would operate in a similar way.

Dynamic taint tracking [3, 4, 6, 11, 22, 26, 29, 30] is known to incur high runtime costs. Fortunately, GPU executions exhibit some unique characteristics that enable optimization. We present an optimization that recognizes and identifies the large portion of GPU instructions that cannot be involved in taint propagation from sources to sinks. Furthermore, given the large register file on GPU and frequent register accesses, we maintain register taint map in registers to accelerate their taint tracking. These optimizations are performed through binary-level static analysis.

3.1 Taint Reachability

On GPUs, we discover that programs frequently operate on a set of critical runtime un-taintable values, and that not all operands need to be tracked. We exploit this fact and only track the operands that potentially carry taints or may have an impact on the state transition of the un-taintable objects. In the earlier example, if v_1 does not carry any taint, the taint maintenance only needs to track v_2 and v such that $T(v) = T(v_2)$. If neither v_1 or v_2 can be tainted, or if v does not propagate to memory, no taint maintenance is necessary for the variables v, v_1, and v_2.

A frequently used GPU runtime un-taintable is the logical thread index. A thread index is used to help identify the task that is assigned to every thread. It is a built-in variable, and does not come from global memory that is managed by a programmer, and thus the instruction operand as a thread index or an expression of thread index can never be tainted. Similarly, other built-in thread identification variables, including thread block id and dimension configuration, are also un-taintable.

Another frequently used GPU runtime un-taintable value are the non-scalar pointer-type kernel parameters. A GPU kernel function does not allow call-by-reference. To reference a memory data object that can be modified at runtime, it can only use pointers. Moreover, these kernel parameters are kept in a memory region named as "constant memory" in GPUs and are read-only in kernel execution. The memory region pointed to by the kernel parameter must be tracked, but the pointer or the address expression computed using the pointer and thread index (or part of the expression) does not need to be tracked. Other examples include compile-time un-taintable values, such as loop induction variables and stack framework pointers, programmer-specified constants, and combinations of GPU-specific runtime constants with these constants. We analyze and categorize these un-taintable values in Section 5 and Table 1.

To avoid tracking un-taintable values in GPU programs, we take the following approaches.

1. We classify an instruction operand into two types: taintable and un-taintable. The *taintable* state indicates that the operand might be tainted at runtime—whether it will be really tainted depends on the exact dynamic analysis done by tracking instructions. The *un-taintable* state indicates that the operand cannot be tainted at runtime. Any operand that cannot be reached from the taintable source is un-taintable. The taintable sources are program inputs given by the users and reside in the global memory on GPUs. Examples include face recognition photos, a plain-text message, and encryption key.

2. A variable can be overwritten with taintable or un-taintable values at different program execution points. We check for potential state transition of a variable: from un-taintable to taintable, or from taintable to un-taintable. The latter arises in a situa-

tion called *taint removal*—e.g., assigning a constant to a register who might be in a tainted state before the assignment but must now transition to the un-tainted state.

3. We statically check the memory reachability: whether an operand might reach memory (potential taint sinks). Even if an operand is taintable, as long as it does not flow into memory, it will not affect any taint sink. We do not need to add tracking instruction for this type of operands. Common examples include loop trip counters, predicate registers, and stack frame pointers.

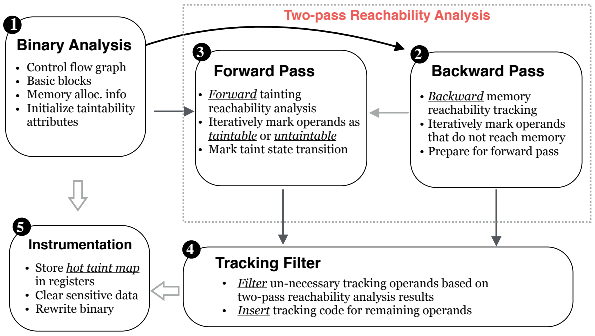

Two-pass Reachability Analysis
1
Binary Analysis
• Control flow graph
• Basic blocks
• Memory alloc. info
• Initialize taintability attributes
3
Forward Pass
• Forward tainting reachability analysis
• Iteratively mark operands as taintable or untaintable
• Mark taint state transition
2
Backward Pass
• Backward memory reachability tracking
• Iteratively mark operands that do not reach memory
• Prepare for forward pass
5
Instrumentation
• Store hot taint map in registers
• Clear sensitive data
• Rewrite binary
4
Tracking Filter
• Filter un-necessary tracking operands based on two-pass reachability analysis results
• Insert tracking code for remaining operands

Figure 2: Overview of our taint tracking system.

```
Original Code                       Tracking Code
1:block0:                           1:
2:    R0 = 0x1234;                  2:
3:    R0 = 0x0;                     3: T(R0) = false;
4:    if (some_condition)           4:
5:       R0 = [R1];                 5: T(R0) = T([R1]) || T(R1);
6:    [R2] = R0;                    6: T([R2]) = T(R0);
7:    some_condition = random();    7:
8:    GOTO block0;                  8:
```

We show an example in the code snippet above. The code describes a loop. Register R0 is overwritten with different types of values. Initially R0 is written with an un-taintable value (lines 2-3). Later in the *if* statement, it is written by a taintable value [R1]; note that here the [R1] notation indicates a memory operation and the address of the memory location is R1. We need a tracking instruction within the *if* statement since [R1] comes from global memory and every operand from memory needs to be tracked. We do not need a tracking instruction for line 2 since $0x1234$ is a constant and the assignment target R0 at line 2 cannot reach memory. However, we do need a tracking instruction for line 3 since the assignment target may reach memory and taint removal applies here (R0 may be tainted from an earlier iteration of the loop and if so, taint must be removed here).

3.2 Iterative Two-pass Taint Analysis

To mark the taintability and reachability attributes for every operand and to detect potential taint state transition, we perform an iterative dataflow analysis.

There are two passes in our iterative dataflow analysis component. The forward pass marks the taintable operands and the un-taintable operands only at the program points where a potential taint state transition occurs. The backward pass marks an operand that potentially reaches memory (taint sinks). In the end, when adding code to track the original program, we only track the operands that are marked in both forward and backward passes.

Figure 2 provides an overview of our taint tracking system. First, we analyze the binary code to obtain the control flow graph and a list of basic blocks. A basic block is the maximum length single-entrance and single-exit code segment. We also mark the operands that are known to be un-taintable before the program starts. They include built-in thread identification variables, non-scalar pointer type kernel parameters, and other programmer-specified constants.

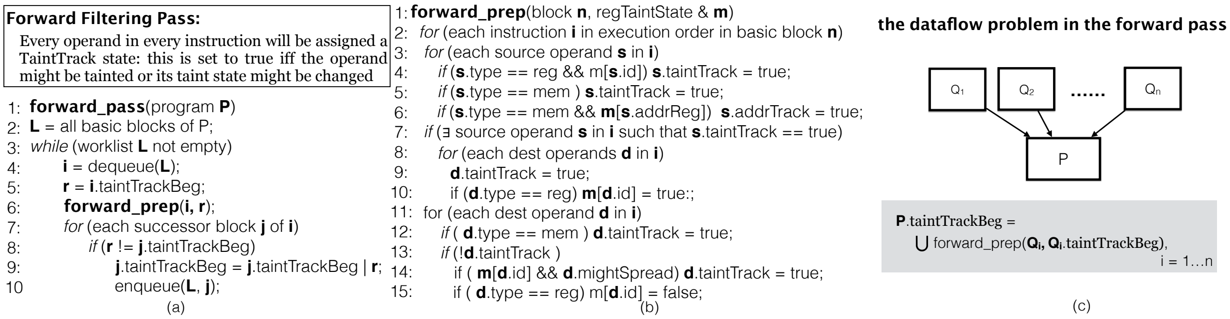
We perform the backward pass first to analyze each operand and set its memory reachability attribute. We name it the *mightSpread* attribute, indicating whether there exists an execution path through which the value of this operand might spread into memory.

We then perform the forward pass to mark all operands as taintable or un-taintable, and for every un-taintable operand, we also analyze if its last immediate state is taintable in one of the potential execution paths. If an operand is taintable or its last immediate state is taintable, we set the *taintTrack* attribute to be true. The taintTrack attribute indicates that the operand may be tainted at runtime. For an indirect memory operand, we also need an attribute on the taintability of the addressing register. We call this *addrTrack* attribute.

Finally, in the *Tracking Filter* component, we scan all instructions and review the taintability and reachability attributes each operand. For the destination operand, if its taintTrack and mightSpread attributes are both true, we add tracking code for this destination operand, otherwise we don't. Similarly, for source operands, if both of its taintTrack and mightSpread attributes are true, we add tracking code for the source operand before the tracking code for the destination operand. For an indirect memory source operand, if its addrTrack and mightSpread attributes are both true, we add taint tracking code for the source operand addressing register.

We describe the detailed algorithms for forward and backward passes below.

Forward Taint Reachability Analysis The input is a control flow graph and a set of basic blocks for the GPU program. The output is the taintTrack property value for every operand in every instruction. We show the forward



Forward Filtering Pass:

Every operand in every instruction will be assigned a TaintTrack state: this is set to true iff the operand might be tainted or its taint state might be changed

```
1:  forward_pass(program P)
2:  L = all basic blocks of P;
3:  while (worklist L not empty)
4:      i = dequeue(L);
5:      r = i.taintTrackBeg;
6:      forward_prep(i, r);
7:      for (each successor block j of i)
8:          if (r != j.taintTrackBeg)
9:              j.taintTrackBeg = j.taintTrackBeg | r;
10              enqueue(L, j);
```

(a)

```
1:  forward_prep(block n, regTaintState & m)
2:   for (each instruction i in execution order in basic block n)
3:    for (each source operand s in i)
4:      if (s.type == reg && m[s.id]) s.taintTrack = true;
5:      if (s.type == mem ) s.taintTrack = true;
6:      if (s.type == mem && m[s.addrReg])  s.addrTrack = true;
7:    if (∃ source operand s in i such that s.taintTrack == true)
8:      for (each dest operands d in i)
9:        d.taintTrack = true;
10:       if (d.type == reg) m[d.id] = true:;
11:   for (each dest operand d in i)
12:     if ( d.type == mem ) d.taintTrack = true;
13:     if (!d.taintTrack )
14:       if ( m[d.id] && d.mightSpread) d.taintTrack = true;
15:       if ( d.type == reg) m[d.id] = false;
```

(b)

the dataflow problem in the forward pass
Q1
Q2
......
Qn
P
P.taintTrackBeg =
⋃ forward_prep(Qi, Qi.taintTrackBeg),
i = 1…n

(c)

Figure 3: Forward taint reachability analysis.

pass algorithm in Figure 3(a).

We adopt the fixed-point computation algorithm that is used in standard dataflow analysis (DFA) framework. Function *forward_pass* in Figure 3*(a)* scans the basic blocks one by one, sets the taintTrack attribute for every operand, and updates the *taintTrackBeg* attribute for every basic block. Our forward analysis pass checks if one basic block's taintability updates affect another basic block's taintability results, and if so, adds the affected basic block to the worklist. Initially, all basic blocks are added to the list. The analysis pass finishes only when all basic block's taintability results do not change.

A DFA problem is formulated using (a set of) dataflow equation(s). We describe the dataflow equation as follows. The taintTrackBeg attribute describes the taint tracking state of every register at the beginning of a basic block, which is a bit array. Every bit in the bit array corresponds to one physical register. If a register's taintTrack attribute is true at the beginning of the basic block of interest, this bit is set to 1, otherwise 0. Assume a basic block P and it has n predecessor basic blocks Q_i, $i = 1...n$, the dataflow relation is

$$\text{P.taintTrackBeg} = \cup$$
$$\text{forward\_prep}(Q_i, Q_i.\text{taintTrackBeg}).$$

The *forward_prep* function in Figure 3(b) updates the taintability state for all instructions in a basic block based on the taintability state at the beginning of the basic block. It scans the first instruction to the last instruction.

Given an instruction, the *forward_prep* function checks its source operands first (lines 3−6 in Figure 3(b)). If a source operand is register and the taintTrack attribute is true, this source operand needs to be tracked. If a source operand is of memory type, it has to be tracked. Note that if the address register of an indirect memory operand is taintable, we need to track the register as well—setting addrTrack attribute at line 6 in Figure 3(b).

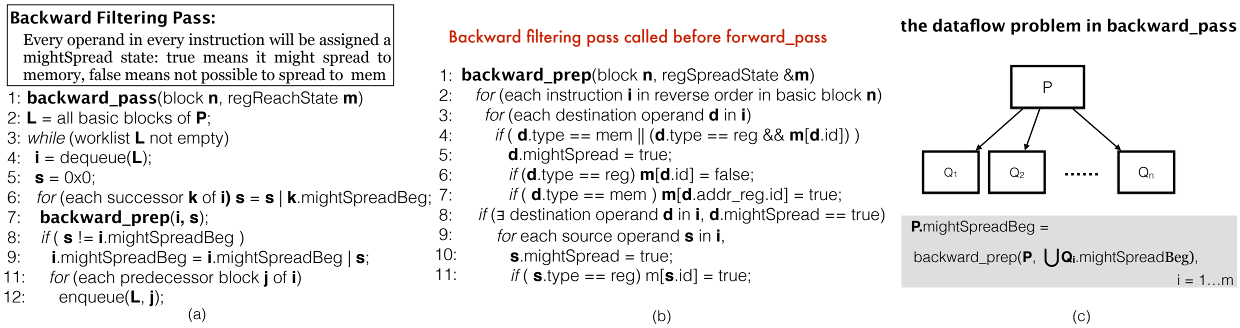
Next, the *forward_prep* function checks every destination operand. If any source operand needs to be tracked based on the above analysis, destination operand needs to be tracked as well. In the meantime, we update the register tracking state for the corresponding destination operand (line 10 in Figure 3(b)). If the destination operand is of memory type, it needs to be tracked. If the destination operand is un-taintable (lines 13-15 in Figure 3), and its prior tracking state is taintable, and the destination operand might spread to memory, the destination operand needs to be tracked as well. Further we update the register tracking state for the corresponding destination operand.

code	regTaintState	taintTrack
block 3:		
R0 = **R1** + R2;	→ [1,1,0,0];	→ taintTrack(R0, R1, R2) = {*true, true, false*};
R1 = 0x5000;	→ [1,0,0,0];	→ taintTrack(R1) = *true*;
R2 = **[**R1**]**;	→ [1,0,1,0];	→ taintTrack(R2, R1) = {*true, false*};
R3 = **R0** + 0x1;	→ [1,0,1,1];	→ taintTrack(R3, R0) = {*true, true*};
BRA block5;		

We use the above example to illustrate the **forward_prep** step for updating the register tracking state. Let the initial regTaintState be [0, 1, 0, 0], meaning that only register R1 is found to be taintable on entry to this basic block. Since the first instruction has R1 as a source and R0 as a destination, we set the operand's taintTrack flag and regTaintState[0] to true.

Since the second instruction writes an immediate value to R1, but since regTaintState[1] was previously true, we have to set the operand's taintTrack flag to true, if its result can spread to memory. This instruction potentially changes the taint value of R1 at runtime from true to false, so if it can reach memory, then we need to instrument it, or else we will suffer from over-tainting as a result of incorrectly treating the data as still being tainted. We flip regTaintState[1] to false since at compile-time and at the second instruction, register R1 is untaintable.

The next instruction loads from memory into R2, so we set the operands' taintTrack flags and regTaintState[2] to true, because memory is a possible taint source. The final instruction before the branch carries potential taint from R0 to R3; since regTaintState[0] is true, regTaintState[3] is set to true along with the operand's taintTrack flag.



Backward Filtering Pass:

Every operand in every instruction will be assigned a mightSpread state: true means it might spread to memory, false means not possible to spread to mem

```
1: backward_pass(block n, regReachState m)
2: L = all basic blocks of P;
3: while (worklist L not empty)
4:   i = dequeue(L);
5:   s = 0x0;
6:   for (each successor k of i) s = s | k.mightSpreadBeg;
7:    backward_prep(i, s);
8:    if ( s != i.mightSpreadBeg )
9:      i.mightSpreadBeg = i.mightSpreadBeg | s;
11:     for (each predecessor block j of i)
12:       enqueue(L, j);
```

(a)

Backward filtering pass called before forward_pass

```
1: backward_prep(block n, regSpreadState &m)
2:   for (each instruction i in reverse order in basic block n)
3:     for (each destination operand d in i)
4:       if ( d.type == mem || (d.type == reg && m[d.id]) )
5:         d.mightSpread = true;
6:         if (d.type == reg) m[d.id] = false;
7:         if ( d.type == mem ) m[d.addr_reg.id] = true;
8:     if (∃ destination operand d in i, d.mightSpread == true)
9:       for each source operand s in i,
10:        s.mightSpread = true;
11:        if ( s.type == reg) m[s.id] = true;
```

(b)

the dataflow problem in backward_pass

P.mightSpreadBeg =

backward_prep(P, $\bigcup Q_i$.mightSpreadBeg),

i = 1…m

(c)

Figure 4: Backward memory reachability analysis.

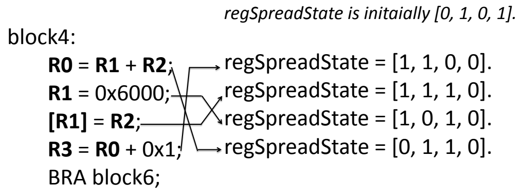
Backward Memory Reachability Analysis Similar to the forward pass, the backward pass uses the program as input. The output is the memory reachability property of every operand. The backward reachability analysis also uses a dataflow analysis framework, which solves the *mightSpreadBeg* bit array for every individual basic block, representing the memory reachability state of the registers at the beginning of basic block. In this bit array, each bit corresponds to one physical register. A value of 1 for the bit at index n of basic block b means that the value of register Rn at the beginning of basic block b might reach memory.

The relationship between one basic block P and its successor basic blocks $Q_i, i = 1..m$, where m is the total number of immediate successor basic blocks, is described using the following equation:

$$P.\text{mightSpreadBeg} = \text{backward\_prep}(\cup\ Q_i.\text{mightSpreadBeg}).$$

The initial mightSpreadBeg bit array is set to 0 for every basic block. Our backward pass keeps updating the mightSpreadBeg bit arrays until they do not change any further (Figure 4(a)). In the meantime, the attribute mightSpread is updated for every operand, as described in Figure 4(b).

The *backward_prep* function calculates mightSpreadBeg for every individual basic block. In Figure 4(b), we scan the instructions in reverse order in a basic block. First, we check the destination operand, if it is register type and the register's memory reachability state is true, the destination operand's mightSpread attribute is set to true. In the meantime, we update the register's memory reachability state for the destination register to false since the value to spread into memory is defined at this point and for any instruction that happens before this instruction, they don't see the same value as defined here. If it is memory type, the mightSpread attribute is set to true and the address register's reachability state is set to true (line 7 in Figure 4(b)). Next, we check the source operands. If any destination operand can spread into memory, then all source operands' mightSpread property is set to true (line 10). Correspondingly, we will set the register reachability state to true (line 11).



regSpreadState is initaially [0, 1, 0, 1].

```
block4:
    R0 = R1 + R2;        regSpreadState = [1, 1, 0, 0].
    R1 = 0x6000;         regSpreadState = [1, 1, 1, 0].
    [R1] = R2;           regSpreadState = [1, 0, 1, 0].
    R3 = R0 + 0x1;       regSpreadState = [0, 1, 1, 0].
    BRA block6;
```

We use the above example to illustrate the process for updating the mightSpreadBeg bit arrays in the backward pass. The backward pass is mechanically similar to the forward pass, aside from the direction in which instructions are processed. In this example, we assume that registers R1 and R3 have been determined to spread to memory in later blocks, hence the initial regSpreadState value of [0, 1, 0, 1]. We skip over the branch instruction since it has no operands except for a jump offset.

The last instruction has data flow into R3 from R0, and regSpreadState[3] is true, so we mark the R0 operand's mightSpread flag as true and set regSpreadState[0] to true. We also flip regSpreadState[3] to false since this instruction is overwriting R3.

The instruction before the last stores register R2 to memory, so we simply mark the R2 operand's mightSpread flag as true and set regSpreadState[2] to true.

The third instruction counting from the last puts an immediate value into R1, so we set regSpreadState[1] to false. Finally, the fourth instruction counting from the last has data flow into R0 from R1 and R2, and regSpreadState[0] is true, so we mark both source operands' mightSpread as true, set regSpreadState[1] and regSpreadState[2] to true, and set regSpreadState[0] to false since R0 has been overwritten.

3.3 Register Taint Map in Registers

A GPU contains a much larger register file than CPU does—e.g., every streaming multi-processor has 64K registers on most NVIDIA GPUs. Registers are naturally accessed frequently and maintaining their taint statuses require frequent reads and writes from/to their taint map locations. At the same time, the large GPU register file presents the opportunity to maintain a portion of the taint map in registers. These facts motivate us to place the register taint map in registers.

We use multiple 32-bit general purpose registers to store the taint map, in which one bit corresponds to one register that is tracked. Using register-stored taint map increases the number of registers used per-thread, and might decrease occupancy, determined as the number of active threads running at the same time. Fortunately in many GPU programs, not all the register file is needed [8, 20] nor the maximum occupancy is necessary [10] for the best program performance. Therefore the overall taint tracking cost is significantly reduced by our use of register-stored taint map, as demonstrated later in evaluation.

4 Tainting-Enabled Data Protection

Taint tracking results can be used to help protect sensitive data and prevent information leaking on GPUs. We describe two major use cases of taint tracking analysis and present our prototype implementation of tainting-enabled data protection.

4.1 Sensitive Data Removal

Lee et al. [18] and Pietro et al. [25] have recently demonstrated that information leaking from one program to another may occur in GPU local memory between GPU kernel executions, and in GPU global memory between program runs. Our taint tracking results may help a program understand the propagation of certain sensitive information and clear all taints before relevant points of vulnerability (e.g., clearing local memory taints at the end of each kernel and clearing global memory taints at the end of program run).

We make a prototype implementation of this use case. For registers, we let every thread clear its own tainted registers. It is possible that some threads exit earlier than others. However since register taint map is thread-private, we can insert the clearing code right before every *EXIT* point and thus early-exiting threads can also clear their tainted registers early. For local memory, since it is thread-private, we treat it the same way as registers. Note that registers and local memory cannot be cleared by programmers themselves (unlike shared memory and global memory) and thus a trustworthy binary instrumentation tool is necessary to prevent sensitive data from leaving taints on GPUs.

For shared memory, since shared memory is visible to all threads in the same basic block, we need to make sure the sensitive shared memory data is cleared *after* all threads in the same thread block finish their work. Therefore, our design is to create a control flow reconvergence point for all threads since different threads might take different execution paths. We then insert a thread block level barrier at the reconvergence point before clearing the tainted shared memory data.

Pietro et al. [25] proposed a register-spilling based attack, which makes use of compiler to force spilling the registers so that the encryption key (or reversibly transformed encryption key) in the AES encryption module in the *SSLShader* program can be moved from registers to local memory. Then a second running application can steal the leaked information in local memory. Our taint clearing approach prevents such attacks by clearing the registers, local memory, and shared memory right before every thread in the GPU application completes.

Experimental results in Section 5 will show that the data clearing cost is low—worst-case slowdown of 13% and in most cases no more than 5% slowdown.

4.2 GPU Malware Countermeasure

GPU taint analysis identifies where and when sensitive data is sent from GPU device to CPU or other network devices. This is especially important for integrated CPU-GPU whole system taint tracking. A dynamic taint tracking system that only monitors data dependences during CPU execution may miss the influence propagation of untrusted inputs or execution results through GPU computation. For example, GPU malware Keylogger [17] and Jellyfish [15] exploited direct memory access (DMA) at mapped CPU memory to snoop the CPU system activities and steal host information. GPU may obtain the leaked CPU information, process it, and send it through a network or other output device while evading countermeasures that only monitor CPU executions.

Our GPU data protection system can not only clear sensitive data, but also capture possible attempts of stealing and emitting sensitive information. We prevent this type of attacks by dynamically monitoring the data transfer between CPU and GPU. If the GPU-mapped CPU memory contains sensitive information (i.e., keystroke buffer in the Keylogger attack [17]), the mapped data region is marked as taint sources. We track the dependency propagation of tainted data in GPU executions. Further we statically instrument memory transfer APIs so that before any data is sent from GPU through cudaMemcpy APIs in CUDA or clEnqueueReadBuffer APIs in OpenCL, the memory address range is checked. If the transmitted data falls within the sensitive tainted memory range, we either alert the system that tainted data is transmitted, or mark the corresponding CPU destination memory region (if data is transferred back to CPU) as tainted. Since all communication between GPU and other devices rely on explicit memory transfer API, we can check and protect information flow by instrumenting these memory transfer APIs.

Our taint tracking and data protection system helps protect applications that utilize both CPU and GPU, if combined with CPU taint tracking. Our work ensures full system taint tracking that is essential to whole system security. We are not aware of any other work that provides the same degree of protection. Besides the Keylogger case, other potential whole-system tracking examples include a web site that relies on taint tracking to prevent untrusted user inputs with malicious database queries (e.g., through SQL injections) or invoking dangerous system calls (e.g., through buffer overflow attacks).

Finally, when GPU tainting is securely applied to untrusted programs, it can also identify malicious programs that attempt to scan uninitialized data which may have been left by previous kernel and program runs from other users. We have not implemented this type of data protection. However, our tool can be readily extended to help detect uninitialized memory region containing sensitive data left by prior GPU program execution and clear / zero them if appropriate.

5 Evaluation

We perform evaluation on a machine configured with an NVIDIA GTX 745. This is a "Maxwell" generation GPU with compute capability 5.0. Since NVIDIA's compiler and binary ISA are closed-source, we modify the GPU binaries using tools inspired by the asfermi [12] and MaxAs [9] projects, allowing for binary instructions be directly inserted into the executable.

5.1 Benchmarks

Our evaluation employs a variety of GPU kernels in deep learning, image processing, and data encryption. First, Caffe [16] is a deep learning framework in which a user writes a "prototxt" file describing the input and layers of the deep learning network (e.g., convolutional layers, inner-product layers, etc.), which can be fed into the Caffe executable to create, train, and test the network. Newer versions of Caffe allow various layers to be executed on the GPU via CUDA. A common use of Caffe is image classification. We use three Caffe kernels in our evaluation: im2col, ReLUForward, and MaxPoolForward. These three kernels consume the majority of the execution time for image classification.

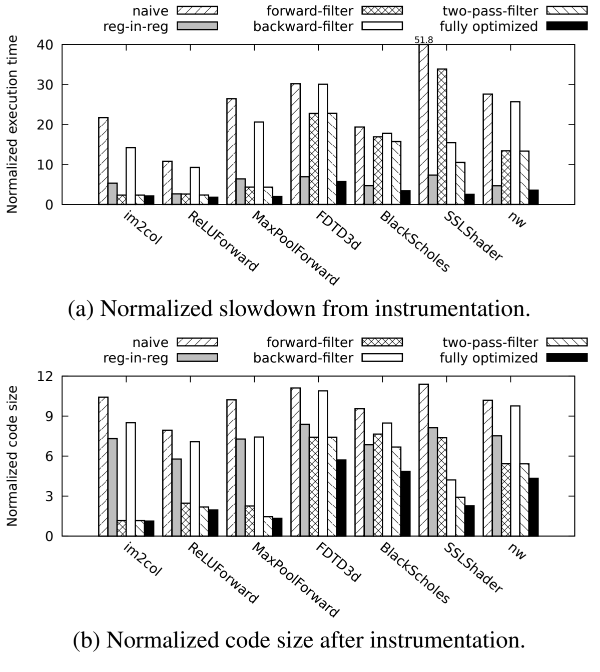
We additionally use kernel functions from the CUDA SDK [23], the Rodinia benchmark suite [2], and SSLShader [13]. From the CUDA SDK we include BlackScholes, a program for financial analysis, and FDTD3d, a 3D Finite Difference Time Domain solver. As a numerical analysis program, FDTD3d is unlikely to have sensitive data to protect, but serves as an additional data point for testing our performance. From Rodinia, we include Needleman-Wunsch, a bioinformatics benchmark used for DNA sequencing. From SSLShader, we include an AES encryption program.

5.2 Taint Analysis & Optimizations

We evaluate the effectiveness of the two performance enhancing techniques in Section 3—taint reachability filtering and taint map in registers.

Since we modify the executable directly, we measure the cost of taint analysis in terms of both slowdown and code size. There are a few factors which exacerbate these costs. Whenever we insert an instruction to get or set a location's taintedness in memory, we first have to calculate its address. Since addresses for global memory are 64-bit on this architecture, but registers and integer operations are 32-bit, this requires multiple instructions with immediate dependencies.

Additionally, each thread has access to only one carry flag, so if it is already in use where we need to get the taint address, extra instructions are needed to spill it into a register or to memory. Furthermore, the singular carry flag makes it difficult to interleave instructions effectively, since they may overwrite each other's result. As the GPU is incapable of out-of-order execution within a thread, the latency for accessing the taint-map is costly.



(a) Normalized slowdown from instrumentation.

(b) Normalized code size after instrumentation.

Figure 5: Overhead of tainting instrumentation.

Figure 5(a) illustrates the GPU tainting slowdown with each of our optimizations, compared to native execution. The 'naive' bar shows slowdown without any optimizations, the 'reg-in-reg' bar shows the results of placing part of the taint map into registers, the 'forward-filter' and 'backward-filter' bars show the results of each filter pass, the 'two-pass-filter' bar shows results when using both filter passes, and the 'fully optimized' bar shows results when using all of these optimizations. Figure 5(b) shows normalized code sizes (static instruction counts) for the same cases.

Figure 5 shows that both two-pass filtering and hot register taint map can reduce the tainting cost significantly. For the filter passes, there is a high correlation between relative slowdown and code size after instrumentation. Saving taint mapping into registers does not shrink as much of the code size as two-pass filtering, but it still improves the tracking performance significantly. The tracking cost saving comes more from the reduced memory latency than from reduced instruction count.

Taint Map in Registers Even on its own, saving part of the taint map in registers reduces significant time during taint analysis. The main alternatives, local memory and global memory, are both off-chip memories that may take hundreds of cycles to access. Even the cache to which such memory is saved is off-chip, because the on-chip L1-cache is typically only used for read-only data on newer architectures [24]. Since most GPU programs have numerous threads running at once, some of this latency is hidden by some threads continuing to execute while others wait for memory accesses to complete, but even so, saving register taint information into registers reduces slowdown compared to naive taint tracking in our benchmarks by 78% on average.

Filtering The forward pass filtering also saves significant time, though it has more variance across different benchmarks. Its effectiveness stems from the properties of GPU kernels. Most kernels make use of non-taintable, read-only data such as thread ID and grid size to perform many calculations. Additionally, function parameters are read-only in GPU functions, making it impossible for them to become tainted in most programs. On its own, the forward pass reduces slowdown in our benchmarks by an average of 53%.

kernel	para- meter	imme- diate	const mem.	thread block id
im2col	85%	85%	29%	64%
ReLUForward	20%	40%	47%	57%
MaxPoolForward	70%	72%	57%	58%
FDTD3d	17%	17%	11%	12%

Table 1: Percentage of filtered-out instructions for various reasons.

We also analyze the reason why we are able to filter out a significant number of instructions for some applications in the forward pass. Table 1 shows the percentage of filtered out instructions under different categories. *Parameter* means one or more source registers are from the (constant-memory) kernel parameters. *Immediate* means one or more source operands are immediate numbers. *Const memory* means at least one source is from constant memory. Finally, *thread/block id* means the influence is from the identifier of the current thread or thread block. The identifiers are stored in special registers private to each thread or constant memory depending on the architecture, but in either case they are known at static-time. While it might be surprising that the sum of percentages due to multiple reasons may exceed 100%, note that an instruction may be filtered out due to multiple reasons.

We discover that most instructions are filtered out because of these four categories. The reason is that GPU programs distribute workload among threads based on their ids. To get the assigned workload, each thread must perform a lot of computation using ids, immediate, and constant memory values (e.g., thread block & grid dimensions). The computation results, together with parameters (e.g., the start address of an array), are used to fetch assigned data. Then the real computation starts as well as the taint tracking. For most GPU programs, the real computation is short with several instructions, and the preprocessing including address calculation consumes most of the time. That is why we can filter out most instructions in our forward pass: most instructions do preparation work and are not related to the potentially tainted input data. For FDTD3d, the computation is more complex and fewer instructions are filtered out. It also explains why FDTD3d does not benefit from two-pass filtering as much as compared with other benchmarks, as shown in Figure 5(a).

The backward pass is usually less effective than the other optimizations. While a lot of the inputs to a kernel function are effectively constants, the only means of returning anything is through global memory. As such, we can expect that most operations will produce values which influence memory. Regardless, the backward pass does provide some benefit in most cases, and in the SSLShader benchmark it reduces slowdown compared to the naive approach by 22%.

Combined Optimizations Compared to the forward pass, the two-pass filter reduces slowdown by 12% on average, and compared to the backward pass, it reduces slowdown by 50%. Full optimization reduces slowdown by 56% compared to the two-pass filter, and 42% compared to only keeping part of the taint map in registers. This demonstrates the merit of combining our different optimization techniques, which together reduce slowdown by an average of 87%.

With full optimizations, our benchmarks' kernel functions experience an average normalized runtime of 3.0× after instrumentation. The FDTD3d benchmark suffers the worst slowdown at 5.7× runtime, due to frequent use of shared memory making the filter less effective. The

Needleman-Wunsch benchmark, which also has shared memory usage, is the next slowest with a 3.6× runtime. Although the SSLShader benchmark also makes use of shared memory, it only uses shared memory to store compile-time constants for faster retrieval, allowing us to filter out all shared memory instructions for less runtime slowdown of 2.5×.

One special consideration when modifying GPU programs is occupancy—the number of threads that can be live at once. A high occupancy means that latency is less costly, as the GPU can switch to different groups of threads every cycle. Since our instrumentation results in additional use of registers, and the register file is evenly split among all live threads, there is potential for occupancy to be decreased, hurting performance more drastically. In such a case, it may be more beneficial not to store any part of the taint map into registers. However, in practice, we use few enough additional registers that reducing occupancy is unlikely, since for every 32 registers in the original program, we only need 1 extra register to store their taintedness. We find that GPU programs typically use less than 64 registers per-thread, and so none of our benchmarks require more than two extra registers per-thread for storing register taintedness.

5.3 Memory Protection

We next evaluate the incorporation of memory protection into our dynamic analysis framework. As discussed in Section 4, the GPU does not clear memory before deallocation. This includes all types of memory, both on-chip and off-chip. [25] demonstrates that data left behind even in local memory and shared memory can be stolen, such as the encryption key and plaintext in the SSLShader benchmark. We have found that this data can also be stolen directly from registers by preparing a kernel function with the same thread block size and occupancy as the victim kernel function—thereby ensuring the register file will be partitioned in the same, predictable manner—and then manually coding the eavesdropping kernel's binary to read the desired registers.

Programmers can manually erase global memory before program exit, but registers and local memory are allocated by the compiler and cannot be as easily cleared. Sensitive data in registers, local memory, and also shared memory must be cleared before the kernel function exits, or else a malicious kernel function may be invoked and acquire these resources for itself. We leverage our instrumentation framework to clear sensitive data in these regions, via additional modification to the binary code. This can be used to prevent attacks such as the one in [25], which stole encryption key data through such resources. The results are summarized in Table 2.

GPU kernel	*Memory*	*Slowdown*
im2col	N/A	0.26%
ReLUForward	N/A	0.33%
MaxPoolForward	N/A	0.59%
FDTD3d	Shared	5.10%
BlackScholes	N/A	0.40%
SSLShader	Local	0.41%
needle	Shared	13.05%

Table 2: Slowdown from memory erasure during kernel execution, measured as a fraction of the original kernel time. ”Memory” column indicates which memory types need to be cleared (besides registers).

Since registers and local memory are thread-private, they can be safely cleared by each thread prior to exit. We insert instructions to clear this data before the EXIT instruction, using the results of our forward-filter pass to avoid unnecessary work. But shared memory is shared by every thread in a thread block, and therefore may not be safe to erase until all of its threads finish execution. Before the EXIT instruction we insert a synchronization barrier, which causes threads to halt until all other threads in the block reach the same point, and then add a loop which has every thread zero out a separate portion of shared memory. In benchmarks with less regular control flow, where threads exit at different points in the code, we can instead have shared memory cleared by a subset of its threads.

We find that the cost to clear tainted registers is trivial, adding only a fraction of a percent to runtime. Each register takes only one cycle of amortized time to erase for every 32 threads, and the GPU is likely able to overlap most of these cycles with memory stalls from other threads. None of our benchmarks use local memory by default, since it is usually used for register spilling. In order to evaluate the slowdown of clearing local memory, we recompile SSLShader, which uses 40 registers, to instead use 20 registers. Clearing local memory and registers in this benchmark adds 0.41% time overhead.

Shared memory is slower to clear. In FDTD3d, clearing taints in shared memory adds 5.10% runtime compared to the original kernel function, and in Needleman-Wunsch it adds 13.05%. The increased slowdown compared to clearing local memory likely stems from the use of a loop, due to the GPU's inability to perform speculative and out-of-order execution, forcing a thread to wait until each shared memory location is cleared until it can zero the next one. Local memory is simpler to handle, with every thread accessing the same logical addresses despite using different physical locations, allowing for the local memory clearing loop to be fully unrolled.

Using the taint information to erase only sensitive data can help significantly, compared to naively clearing these memories fully. For example, in the SSLShader benchmark the tainted registers and local memory are cleared in 47 mSecs, but this benchmark makes use of shared memory which is never tainted. If its shared memory arrays are erased, in addition to clearing the small amount of registers and local memory in their entirety, then the overhead would jump to 407 mSecs.

6 Related Work

Dynamic taint analysis [3, 4, 6, 11, 22, 26, 29, 30] tracks data (and sometimes control) dependencies of information as a program or system runs. Its purpose is to identify the influence of taint sources on data storage locations (memory, registers, etc.) during execution. Taint tracking is useful for understanding data flows in complex systems, detecting security attacks, protecting sensitive data, and analyzing software bugs. Its implementation usually involves static code transformation, dynamic instrumentation, or instruction emulation using virtual machines to extend the program to maintain tainting metadata. While existing dynamic tainting systems track CPU execution, this paper presents the first design and implementation of a GPU taint tracking system.

A large body of previous work presented techniques to improve the performance of CPU taint tracking. LIFT [26] checks whether unsafe data are involved before a code region is executed, and if not, no taint tracking code is executed for that code region to reduce overhead. Minemu [1] proposes a novel memory layout to reduce the number of taint tracking instructions. It also uses SSE registers for taint tracking to reduce performance overhead. TaintEraser [30] makes use of function summary to reduce the performance overhead of taint tracking. It summarizes taint propagation at the function level so that instruction level taint tracking is reduced. TaintDroid [6] is a taint analysis tool proposed for Android systems. By leveraging Androids virtualized execution environment and coarse-grained taint propagation tracking, it can achieve nearly real time analysis with low performance overhead. Jee et al. [14] proposed to separate taint analysis code from the original program, and dynamic and static analysis was applied on the taint analysis code to optimize its performance. In this paper, we present new performance optimizations by exploiting unique GPU characteristics.

Security vulnerabilities on GPUs have been recognized recently. Dunn et al. [5] showed that sensitive data can be leaked into graphics device driver buffers. They proposed encryption to protect data in transit over the device driver but their approach does not protect data in GPU memory. Lee et al. [18] uncovered several vulnerabilities of leaking sensitive data in GPU memory—leaking global memory data after a program context finishes and releases memory without clearing; leaking local memory data across kernel switches on a CU. They did not present any solution to address these vulnerabilities. More recently, Pietro et al. [25] proposed memory zeroing to prevent information leaking in GPU. However, memory zeroing alone provides limited protection—it cannot track information flow in memory; nor can it counter GPU malware such as Keylogger [17] and Jellyfish [15]. Furthermore, GPU tainting is complementary to memory zeroing—tainting identifies a subset of sensitive memory for zeroing to reduce the costs.

GPU information flow analysis has been performed in the past. Leung et al. [19] and Li et al. [21] employed static taint analysis to reduce the overhead of GPU program analysis and verification. Static analysis requires memory aliasing analysis of memory accesses that are inherently imprecise. While they are suitable for testing and debugging purposes [19, 21], security data flow analysis in this paper requires more precise dynamic tracking. Farooqui et al. [7] proposed static dependency analysis between thread index and control conditions to identify possible thread divergence in GPU executions (the result of which helps determine whether symbolic execution can be performed on given GPU basic blocks). Their static dependency analysis is narrowly targeted and it is unclear whether it applies to general taint tracking.

7 Conclusion

Recent discoveries of information leaking through GPU memory and GPU-resident malware call for systematic data protection in GPUs. This paper presents the first design and implementation of a dynamic taint tracking system for GPU programs. We exploit unique characteristics of GPU programs and architecture to optimize taint tracking performance. Specifically, we recognize that a large portion of instructions on GPU runtime parameters and constants can be safely eliminated from taint tracking to reduce tainting costs. We also utilize the large GPU register file for fast maintenance of the taint map for registers. These optimizations result in 5 to 20 times tainting speed improvement for a range of image processing, data encryption, and deep learning applications.

Acknowledgement

We thank Adam Bates for his help during the preparation of the final version of the paper, and the anonymous reviewers for their insightful comments. This work is supported by NSF Grant NSF-CCF-1421505, NSF-CCF-1628401, and the Google Faculty Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: The world's fastest taint tracker. In *International Workshop on Recent Advances in Intrusion Detection* (2011), Springer, pp. 1–20.

[2] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE Int'l Symp. on Workload Characterization (IISWC)* (2009), pp. 44–54.

[3] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole system simulation. In *Proc. of the 13th USENIX Security Symp.* (2004), pp. 321–336.

[4] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A generic dynamic taint analysis framework. In *Proc. of the 2007 Int'l Symp. on Software Testing and Analysis* (London, United Kingdom, 2007), pp. 196–206.

[5] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., AND WITCHEL, E. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 61–75.

[6] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst. 32*, 2 (June 2014), 5:1–5:29.

[7] FAROOQUI, N., SCHWAN, K., AND YALAMANCHILI, S. Efficient instrumentation of GPGPU applications using information flow analysis and symbolic execution. In *Proc. of Workshop on General Purpose Processing Using GPUs* (Salt Lake City, UT, Mar. 2014), GPGPU-7, pp. 19:19–19:27.

[8] GEBHART, M., KECKLER, S. W., KHAILANY, B., KRASHINSKY, R., AND DALLY, W. J. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proc. of the 45th Annual IEEE/ACM Int'l Symp. on Microarchitecture* (Vancouver, B.C., CANADA, Dec. 2012), MICRO-45, pp. 96–106.

[9] GRAY, S. Maxas: Assembler for nvidia maxwell architecture. `github.com/NervanaSystems/maxas`, 2014.

[10] HAYES, A. B., AND ZHANG, E. Z. Unified on-chip memory allocation for simt architecture. In *Proc. of the 28th ACM Int'l Conf. on Supercomputing* (Munich, Germany, 2014), ICS'14, pp. 293–302.

[11] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *Proc. of the First EuroSys Conf.* (Leuven, Belgium, Apr. 2006), pp. 29–41.

[12] HOU, Y., LAI, J., AND MIKUSHIN, D. Asfermi: An assembler for the nvidia fermi instruction set. `code.google.com/p/asfermi/`, 2011.

[13] JANG, K., HAN, S., HAN, S., MOON, S. B., AND PARK, K. SSLShader: Cheap SSL acceleration with commodity processors. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Mar. 2011), pp. 1–14.

[14] JEE, K., PORTOKALIDIS, G., KEMERLIS, V. P., GHOSH, S., AUGUST, D. I., AND KEROMYTIS, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proc. of the 19th Annual Network & Distributed System Security Symp. (NDSS)* (San Diego, CA, Feb. 2012).

[15] GPU rootkit PoC by team Jellyfish. `github.com/x0r1/jellyfish`.

[16] JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[17] LADAKIS, E., KOROMILAS, L., VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. You can type, but you cant hide: A stealthy GPU-based keylogger. In *Proc. of the 6th European Workshop on Systems Security (EuroSec)* (Prague, Czech Republic, Apr. 2013).

[18] LEE, S., KIM, Y., KIM, J., AND KIM, J. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *Proc. of the 35th IEEE Symp. on Security and Privacy* (San Jose, CA, May 2014), pp. 19–33.

[19] LEUNG, A., GUPTA, M., AGARWAL, Y., GUPTA, R., JHALA, R., AND LERNER, S. Verifying GPU kernels by test amplification. In *Proc. of the 33rd ACM Conf. on Programming Language Design and Implementation (PLDI)* (Beijing, China, June 2012), pp. 383–394.

[20] LI, C., YANG, Y., LIN, Z., AND ZHOU, H. Automatic data placement into GPU on-chip memory resources. In *proc. of the 13th Int'l Symp. on Code Generation and Optimization (CGO)* (Feb. 2015), pp. 23–33.

[21] LI, P., LI, G., AND GOPALAKRISHNAN, G. Practical symbolic race checking of GPU programs. In *Proc. of SC14: The Int'l Conf. for High Performance Computing, Networking, Storage and Analysis* (New Orleans, LA, Nov. 2014).

[22] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network & Distributed System Security Symp. (NDSS)* (San Diego, CA, 2005).

[23] NVIDIA. GPU computing sdk. `developer.nvidia.com/gpu-computing-sdk`.

[24] NVIDIA. Maxwell tuning guide. `docs.nvidia.com/cuda/maxwell-tuning-guide/`, 2014.

[25] PIETRO, R. D., LOMBARDI, F., AND VILLANI, A. CUDA leaks: A detailed hack for CUDA and a (partial) fix. *ACM Trans. on Embedded Computing Systems (TECS) 15*, 1 (Feb. 2016).

[26] QIN, F., WANG, C., LI, Z., SEOP KIM, H., ZHOU, Y., AND WU, Y. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *39th Int'l Symp. on Microarchitecture* (2006), pp. 135–148.

[27] TANG, Y., AMES, P., BHAMIDIPATI, S., BIJLANI, A., GEAMBASU, R., AND SARDA, N. CleanOS: Limiting mobile data exposure with idle eviction. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 77–91.

[28] VASILIADIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. PixelVault: Using GPUs for securing cryptographic operations. In *Proc. of the 21st ACM Conf. on Computer and Communications Security (CCS)* (Scottsdale, Arizona, USA, Nov. 2014), pp. 1131–1142.

[29] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. of the 15th USENIX Security Symp.* (Vancouver, B.C., Canada, 2006).

[30] ZHU, D. Y., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 142–154.