Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference

Austin Harris*
austinharris@utexas.edu
The University of Texas at Austin

Shijia Wei*
shijiawei@utexas.edu
The University of Texas at Austin

Prateek Sahu
prateeks@utexas.edu
The University of Texas at Austin

Pranav Kumar
pranavkumar@utexas.edu
The University of Texas at Austin

Todd Austin
austin@umich.edu
University of Michigan

Mohit Tiwari
tiwari@austin.utexas.edu
The University of Texas at Austin

ABSTRACT

Micro-architecture units like caches are notorious for leaking secrets across security domains. An attacker program can contend for on-chip state or bandwidth and can even use speculative execution in processors to drive this contention; and protecting against all contention-driven attacks is exceptionally challenging. Prior works can mitigate contention channels through caches by partitioning the larger, lower-level caches or by looking for anomalous performance or contention behavior. Neither scales to large number of fine-grained domains as required by browsers and web-services that place many domains within the same address space.

We observe that cache contention channels have a unique property – contention leaks information only when it is *cyclic*, i.e., domain A interferes with domain B, followed by interference from B to A. We propose to use this cyclic interference property to detect micro-architectural attacks as *anomalous cyclic interference*. Unlike partitioning, our detection approach scales to many concurrent domains in a single address space; and unlike prior anomaly detectors, cyclic interference is robust to noise from benign interference.

We track cyclic interference using non-intrusive detectors in an out-of-order core and stress test our prototype, Cyclone, with fine-grained isolation in browsers (against speculation-driven attacks) and coarse-grained isolation of cores (against covert-channels embedded in database and machine learning workloads). Full-system simulations on an ARM micro-architecture show close to perfect detection rates and 260 – 1000× lower false positives than using (state-of-the-art) contention alone, with slowdowns of only ~3.6%.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Malware and its mitigation.*

*Austin Harris and Shijia Wei are equal contributors.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358273

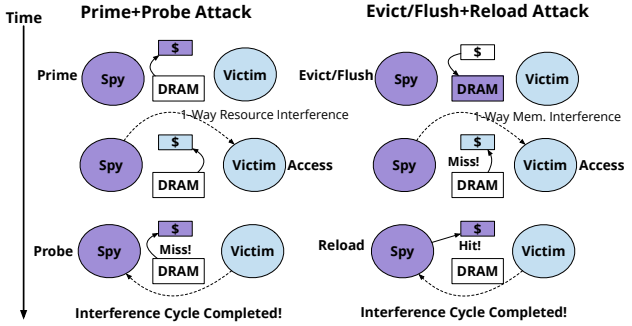


Figure 1: Depiction of cyclic interference on two example side- and covert-channels: Prime+Probe and Evict/Flush+Reload. A micro-architectural information leak through the cache is only completed when a cycle of interference between the spy and the victim is formed.

KEYWORDS

side-channel defenses, secure architectures, anomaly detection

ACM Reference Format:
Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. 2019. Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3352460.3358273

1 INTRODUCTION

Isolation is fundamental to security but surprisingly hard to achieve in practice. Even when encryption and virtual memory prevent an attacker from directly reading secret data, *side-channels* through the micro-architecture leak secrets such as private keys [1, 2, 7, 50, 65, 66, 78, 95], database queries [59], webpage contents [4], kernel memory [49, 54], etc. Side-channel attacks have recently used speculative execution [10–12, 35, 49, 51, 54, 77] as an initial step that greatly amplifies their ability to read secrets across isolation boundaries such as virtual machines, processes, or even browser sandboxes.

Isolation is especially difficult since many security scenarios require information to not leak among *fine-grained* security domains (e.g., sandboxes) that interleave execution at the scale of a few instructions. For example, a browser loading a web-page isolates tens of origins (i.e., website domains) in the same address space, a

web-service isolates data from many distinct users, an application isolates untrusted libraries (including kernel code), etc. Supporting isolation in the same address space is crucial to maintaining performance for many applications (e.g., persistent processes to handle web requests in FastCGI). Unfortunately, speculative attacks bypass all language enforced isolation mechanisms [61] on current systems. Isolating *coarse-grained* domains—such as virtual machines assigned to separate cores—is also challenging and requires customized mechanisms to prevent side-channel leaks through shared caches, memory controllers, or other resources.

Most prior defenses focus on the coarse-grained scenario and use two complementary approaches: (a) *prevent* side-channel leaks by partitioning resources such as caches [22, 33, 44, 48, 55, 56, 81, 83, 84] and memory controller bandwidth [23, 71, 80, 82, 98] among security domains, or (b) *detect* side-channel attacks based on anomalous micro-architecture usage [67], performance counter traces [18], or contention behavior [13, 38] and trigger higher OS-level detectors or virtual machine migrations [62]. While partitioning resources works well when domains are few, partitioning across tens of domains dramatically decreases performance—our SPEC experiments show a 33% slowdown when partitioning an 8-way L1 cache and a 16-way L2 cache into eight domains in the same address space. Detectors, on the other hand, miss attacks and raise false alarms when evasive attacks supply adversarial inputs – we evaluate this in § 3.

In this paper, we identify a new property—*cyclic interference* across security domains—that is common to all known cache contention side-channel attacks, including when they are used deliberately (covert-channels) or driven by transient execution (i.e., Spectre/Meltdown family of attacks). Fig. 1 depicts this property. On the left, an attacker domain contends directly for a cache line ("prime-access-probe") with the victim where the prime and access steps create directional contention from a spy to the victim and the probe step completes the cycle. On the right, the attacker flushes/evicts cache lines first to let the victim bring the cache line in – creating the first step in the cycle – while the cycle is completed when the spy reloads the cache line. Interestingly, in the presence of many security domains, if the cache line is evicted by an unrelated (e.g., non-secret) domain, the cycle is broken and the contention is harmless. While these two attacks look distinct, both require a cycle of directional interference—we describe this intuition in § 4 and examine in § 6 how it applies to speculation-driven attacks (Spectre and Meltdown) without triggering false alarms in benign programs that are specifically created to be similar to both ("SpectreBenign").

We introduce Cyclone, a system that efficiently tracks directional interference in the micro-architecture in order to detect cycles. Cyclone handles coarse-and-fine-grained isolation by enabling software to specify security domains in a security lattice to the micro-architecture. Cyclone then performs *micro-architectural information flow tracking* on these security domains in the registers, caches, and main memory. This tracking defines new propagation rules for accessing micro-architectural resources and accounts for speculative execution. Further, we introduce a non-invasive tracing system, distributed across the micro-architecture, that monitors cyclic interference events and raises alerts when a cache contention attack occurs. Our key idea is to design *local detectors* (LDs) that passively observe requests to each cache, track directional interference conservatively, and forward local anomalous summaries to a *global detector* (GD). LDs count cyclic interference events, sending summaries to the more programmable GD when events occur above a threshold within a time window. The GD uses more sophisticated algorithms to separate out attacks from false positives.

We stress-test Cyclone using fine-grained isolation of origins (i.e., website domains) in a browser (with a Spectre-V1 attack embedded inside an origin), and coarse-grained isolation of cores running privacy-sensitive applications (with cache-based covert channels embedded in each application). We further evaluate the detector using a mix of memory-intensive SPEC workloads. We compare cyclic interference to an improved state-of-the-art system to track contention across domains [13, 38]. For fine-grained isolation against a Spectre browser attack, Cyclone detects 100% of the attack compared to 80% for a contention tracking system [13]. Additionally, Cyclone has a 260× lower false-positive rate. For coarse-grained core-isolation, both Cyclone and contention tracking detect all attacks. In a benign scenario of 4 concurrent SPEC applications, Cyclone generates only 12 false-positives per second, while contention tracking generates 12,000 false-positives per second.

The LDs and GDs in Cyclone do not impact the critical path of our simulated ARM processor while the IPC is lowered by 2.4% due to DRAM tag accesses. The tags which Cyclone relies on to track domains throughout the micro-architecture increase the cache size by 6.25% and the main memory usage by 1.5% when configured to use 8-bit domain-ids. A non-secure processor that uses the extra tag storage in Cyclone to store data instead would see only a 1.2% performance increase. With the extra DRAM accesses and potential opportunity loss, Cyclone sees up to 3.6% performance overhead.

The main contributions of Cyclone are: (i) a new feature for anomaly detectors, cyclic interference, common to all known contention-based cache attacks (ii) a micro-architecture that propagates security domain tags throughout the registers, caches, and memory (iii) software support in the kernel and JavaScript engine to inform the micro-architecture of both fine and coarse-grained security domains as a security lattice (iv) a distributed anomaly detector that tracks cyclic interference with significantly lower false positives (260× for browser and 1000× for a SPEC-mix) and ~ 100% true positives for speculation-driven attacks in browsers, OpenCV, libSVM, and a PostgreSQL database.

2 BACKGROUND

2.1 Isolation Requirements: Coarse- vs. Fine-Grained

Modern systems provide strong isolation between security domains through both hardware and software enforced mechanisms such as virtual memory [5], virtualization [6], and safe languages [60]. Coarse-grained isolation places security domains in separate address spaces (e.g., processes or virtual machines), typically using the hardware memory management unit (MMU) to enforce confidentiality. Many applications also implement fine-grained isolation where multiple security domains run in the same thread of execution and share the virtual address space. Instead of hardware enforced isolation, these applications use language level enforcement such as

type and dynamic bounds checks to enforce confidentiality. This type of fine-grained isolation is often referred to as sandboxing.

In a web browser, the JavaScript engine enforces the same-origin policy: isolation between separate origins (i.e., web domains). For example, a Google ad on a page will be isolated from the parts of the page that don't originate from Google. Fine-grained isolation is useful since a single web page can have tens of origins, e.g., over 16 for `www.cnn.com`.

Efficient isolation also requires allowing legitimate information flows between security domains. A common way of specifying allowed information flows is a *security lattice* [20]. A security lattice describes the level of security that an element in the lattice has access to. Each element in a security lattice is a security domain, such as a virtual machine, process, or sandboxed program. In this model information flows are restricted between low and high elements in the lattice.

2.2 Side- and Covert-Channels

Modern processors share micro-architectural resources like DRAM, caches, TLBs, functional units, predictors etc. extensively. Unfortunately, an attacker can learn information about a victim based on their usage of these resources, thereby bypassing hardware and software isolation mechanisms. An attacker exfiltrates information either through contention on a resource between security domains (e.g., processes, VMs, sandboxes) or by directly observing the victim's usage through a legitimate external interface (e.g., remote timing attacks on SSL [9]). We refer to these information leaks as contention side-channels and observation side-channels, respectively. A covert-channel is a special case of side-channels where a *trojan* intentionally leaks information to a *spy* sharing the resource, for example a compromised text editor leaking sensitive documents. From the perspective of the micro-architectural resource, covert-channels are indistinguishable from side-channels. However a covert-channel attack can achieve much higher rates of information leakage as the attacker is in full control of how they exercise the resource. A special type of observation channel is the termination channel [28], where the total execution time of the program leaks information.

A particularly powerful mechanism for information leakage is speculation-driven attacks [10, 12, 35, 49, 51, 54, 77] where *transient* instructions operate on sensitive information. Transient instructions are instructions executed speculatively (i.e., before they are known to be valid) and ultimately thrown away, or *squashed*, when it is determined they should not have been executed. Because these transient instructions may alter micro-architectural state before finishing all permission checks, an attacker can learn sensitive information using these transient instructions much more easily. These speculation-driven side-channels allow an attacker to bypass all language-enforced sandboxing mechanisms [61].

A very common and high bandwidth type of side-channel attack is through the caches such as Prime+Probe [57, 66], Evict/Flush+Reload [29, 91], and other variants [21, 40, 53, 94]. Cache attacks occur either through contention for particular cache sets (e.g., Prime+Probe) or through contention for particular memory addresses (e.g., Flush+Reload). For example, in a Prime+Probe cache attack, the attacker first primes the cache by bringing in a line, lets the victim execute the information leak, and then measures the timing of a cache probe access. If the victim causes the attacker's line to be evicted, they will observe a slower execution time than if the access was a cache hit; thus revealing information about the victim's access. Cache side-channel attacks have been shown on the first level caches [1, 95], as well as shared last level (LLC) caches [30, 43, 57].

2.3 Threat Model

The focus of this work is defending against cache channels that occur through contention, both when a victim and attacker share the same address space (e.g., browser sandboxing) and when they are isolated by the kernel or hypervisor using virtual memory. Observation channels (e.g., remote timing attacks on SSL [9]), including the termination channel, are not in scope.

We identify five important requirements defenses must satisfy to mitigate cache contention side-channels: (**R1**) support settings (e.g., a web browser) with fine-grained isolation where tens of domains share the same address space and thread of execution, (**R2**) scale well with many concurrent domains, (**R3**) handle speculation-driven attacks, (**R4**) prevent all cache contention attacks or (for anomaly detection solutions) detect most attacks with a low false-positive rate, and (**R5**) have a low performance overhead.

3 MOTIVATION

3.1 Potential Defenses and Limitations

3.1.1 Partitioning and Flushing. One natural solution to micro-architectural side-channels is to prevent the sharing of resources between security domains. Thus many approaches aim to partition resources in space (e.g., cache partitioning [22, 48, 55, 81, 83]) or in time (e.g., one domain per core, memory controller scheduling [23, 71, 80]). To prevent sharing of the virtual address space, Chrome implements Site Isolation where separate origins (i.e., website domains) are placed in separate processes. Rather than partition, a simple defense is to ensure all micro-architectural state is *flushed* when switching between security domains [27].

Unfortunately partitioning and flushing can have significant overheads, especially in settings with fine-grained isolation. Our experiments show 33% IPC overhead for a mix of SPEC2017 apps when using way-partitioning, where each out-of-order core gets only a single way of the cache. In Chrome, Site Isolation incurs a memory overhead of about 10-13% in real workloads [70]. Further, partitioning does not scale to a large number of security domains (e.g., way-partitioning is limited to the number of ways). Thus while these approaches succeed in fully preventing even speculation-driven attacks (**R3**, **4**), they have high overheads (**R5**)—especially in fine-grained isolation settings with many domains (**R1**, **2**).

3.1.2 Randomization and Encryption. Another approach is to introduce randomness or utilize encryption when accessing the resource [37, 56, 69, 83, 86, 96] to make it more difficult for the attacker to infer the information. However, these defenses are still vulnerable to capacity attacks (i.e., contention for the entire capacity of the cache rather than specific sets). For example, Lipp et. al. [53] show that cache attacks are still possible on an ARM system that uses a random replacement policy. Additionally, approaches

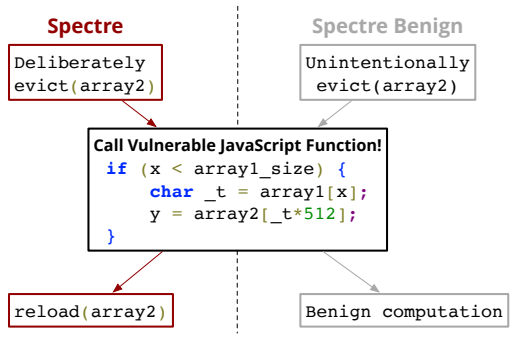


Figure 2: Comparison of Spectre and SpectreBenign executions. Both pages repeatedly call a browser JavaScript function with an input used for a potentially speculative bounds check. However, SpectreBenign does not access out of bounds classified memory speculatively, and does not reload `array2`**.**

such as CEASER [69] only attempt to make it more difficult to find cache conflict sets, thus they do not defend against attacks through memory address contention (e.g., Flush+Reload). Table-based randomization approaches [83, 84] require significant storage overheads: either 1.25MB or 8.5MB for an 8MB LLC [69] depending on whether the OS implements sharing of mapping tables. Thus these approaches scale well, work in fine-grained isolation settings, and handle speculation-driven attacks (**R1**, **2**, **3**), but do not fully prevent attacks (**R4**) and can have high overheads (**R5**).

3.1.3 Anomaly Detection. Rather than prevent the information leakage, another approach is to detect an active attack, called *anomaly detection*. Anomaly detectors monitor system behavior rather than modify the system to eliminate the possibility of information leakage, thus typically leading to low performance overheads and easier deployment. In addition, they provide a mechanism to protect against future attacks. While well-designed anomaly detectors can detect most real attacks, the key challenge is reducing false alerts. Advanced attacks circumvent signature-based approaches and static analysis [63, 93]. Thus dynamic detection systems have been proposed to detect traditional malware such as worms and viruses [18, 46, 47, 67, 73, 90] as well as micro-architectural cache side-channel attacks [13, 14, 18, 79, 93]. Anomaly detectors typically train a machine-learning model on the system's execution characteristics such as OS level signals (e.g., system call traces) [15, 89], instruction traces [3], or micro-architectural signals (e.g., statistics like cache misses from the performance monitoring unit (PMU)). These anomaly detectors can be grouped under three classes:

D1: Generic, unsupervised detectors, which are trained on a set of representative benign applications. These detect deviations from the trained benign behavior, potentially capturing future unknown attacks and protecting a diverse set of applications.

D2: Application-Specific, unsupervised detectors, which are trained on a representative set of benign inputs for one *specific* application. These detectors can achieve better detection rates, but fail to protect systems with diverse workloads.

D3: Application-Specific, supervised detectors, which are trained on a representative set of *both* benign and malicious inputs. Such detectors excel when the attacks on the system are similar to what they were trained on. In practice, such detectors are difficult to deploy since they require knowledge of all possible malicious behavior and each application configuration to be effective.

3.2 Anomaly Detection with PMUs

One anomaly detection approach for existing systems is to utilize performance monitoring units (PMUs) to measure statistics (e.g., cache misses) to detect when cache contention channels are being exploited. PMU-based anomaly detection systems can meet all the requirements in § 2.3 if they have low false-positive rates (**R4**). We evaluate the ability of the three types of detectors, described in § 3.1.3, to detect a JavaScript Spectre attack on both an ARM and x86 system using the provided PMUs.

Detector Testing and Training. We test each detector by running a browser that visits a series of benign websites as well as a malicious site with an embedded Spectre attack. The Spectre attack abuses a vulnerable browser JavaScript function that has a speculative bounds check based on the passed input. To construct "evasive" attacks that look benign, an attacker can modify existing applications that also invoke the same vulnerable function [46]. Thus, we additionally test the detectors using a benign website, SpectreBenign, that calls the same vulnerable function, but does not provide inputs to access out-of-bounds, classified memory and does not probe the cache. Fig. 2 shows the executions of these two pages. SpectreBenign stress tests the robustness of these detectors (i.e. false positive rates against adversarial test cases) to help understand their limitations. In order to train a *generic, unsupervised* detector, we collect traces on a set of six SPEC2017 programs to represent a diverse set of applications. The *app-specific* detectors are trained specifically on the browser, visiting only benign websites for unsupervised and websites with both benign and Spectre for supervised.

Experimental Setup. We collect PMU traces on a 2GHz, 64bit ARM Cortex A57 and a 3.5GHz, 64bit x86 Intel i7-4770K. Since our Spectre implementation in JavaScript abuses branch misprediction to create an Evict+Reload channel on data cache, we select performance counters related to branch predictions and cache misses throughout the cache hierarchy. We also include performance counters like cycle count and instructions retired that were reported to work well in detecting cache side-channels [14, 79, 93]. On ARM we evaluate cycle counts, instructions retired, branch predicts and mispredicts, together with L1 data cache misses and L2 write backs. On x86, we evaluate all cycles, unhalted cycles, instructions retired, all executed branch instructions, all executed mispredicted branch instructions, L2 demand data reads, L2 demand data read misses, L3 read hits, and L3 read misses. Single counter selection is done first and then combinations of the best candidates are evaluated for multi-counter based detection. From this analysis we use L2 data writebacks for the ARM detector, while the x86 detector uses a combination of instructions retired, number of branches, and L2 data read misses.

While there are many choices of feature extraction and classification algorithm, we follow best-practices from prior work on

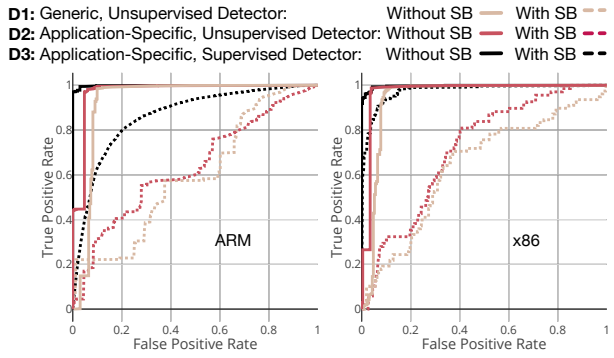


Figure 3: ROC of Three PMU-based Detectors. When SpectreBenign (SB) is included for testing, PMU-based detectors suffer from high false positives. AUC scores for the above figure with and without SB are given as follows:
ARM: D1 {Without: 0.92, With: 0.59}, D2 {Without: 0.97, With: 0.64}, D3 {Without: 1.00, With: 0.83}
x86: D1 {Without: 0.93, With: 0.72}, D2 {Without: 0.98, With: 0.76}, D3 {Without: 0.99, With: 0.97}

anomaly detection with PMUs. Specifically, we run context-switch-aware, per-process data collection multiple times to address the non-determinism issue pointed out in [17]. We use a temporal bag-of-words model [45] to extract features. We use one-class SVM (Support Vector Machine) with the radial basis function kernel [45, 73] for unsupervised detectors (**D1** and **D2**), and XGBoost (Extreme Gradient Boosted Trees) [17, 18, 97] for supervised detectors (**D3**), both with ten-fold cross validation [17, 45, 97].

Analysis. Fig. 3 shows the detection performance of each detector with a ROC (receiver operating characteristic) curve to depict the true positive and false positive rates. Each detector is tested with and without SpectreBenign. The results show that when the detector specifically looks out for Spectre (supervised), it performs well and only sees slight degradation when faced with SpectreBenign. However, for unsupervised detectors, when faced with SpectreBenign (or similar-looking benign examples), otherwise well-designed anomaly detectors suffer from high false positives.

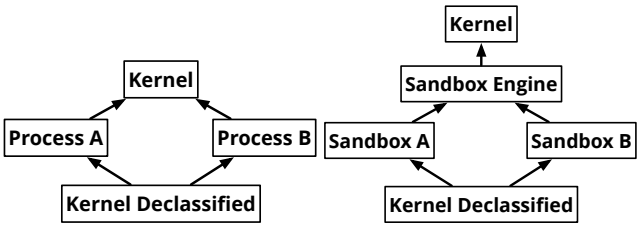
4 PROPERTY: CYCLIC INTERFERENCE

Cyclone uses cyclic interference for detecting contention-based cache side-channel attacks. In this section, we define cyclic interference and introduce the security lattices used for coarse and fine-grained isolation. To reduce false-positives (**R4**), we introduce domain propagation rules and a declassification process. As a driving example, we explain how cyclic interference appears in both Prime+Probe and Flush+Reload attacks. Later in § 6 we present detailed analysis on cyclic interference for speculative attacks (**R3**).

4.1 Cyclic Interference

Direction of Interference. Interference on micro-architectural resources is directional. It occurs from a source to a destination, in which the source of interference, the previously issued instructions, affect the micro-architectural behavior of the interference destination, i.e., the current instruction that is attempting to use the resource. Moreover, a successful side- or covert-channel attack consists of two purposefully controlled directional interference events. To reliably recover information from a micro-architectural channel, it is vital for the attacker to know the state of the channel resource. The attacker accomplishes this by accessing the resource *prior* to the victim operation. Therefore, interference from the attacker is inflicted on the victim during the victim's access. In order to retrieve the transmitted information, the attacker must then probe the resource after the victim's access. This time, interference occurs from the victim to the attacker. Two such consecutive directional interference events on the same resource construct a successful information leak, and will inevitably create a cyclic interference event. However, if the resource is accessed by a third party in between attacker and victim, there will be neither information leak nor cyclic interference.

Notation of (cyclic) interference. We use the symbol $\rightsquigarrow$ to describe the direction of interference on shared resources and physical memory. $\{a \rightsquigarrow b\}$ depicts interference from domain a to b, in which operations of domain a happen before ones of b. For example, $\{a \rightsquigarrow b \rightsquigarrow c\}$ shows interference first happened from a to b, then from b to c. Therefore, the cyclic interference is noted as $\{a \rightsquigarrow b \rightsquigarrow a\}$, where interference $\{b \rightsquigarrow a\}$ follows $\{a \rightsquigarrow b\}$. When this event occurs due to interference on a resource we call it *cyclic resource interference* (CRI), while when it occurs on a memory address we call it *cyclic memory interference* (CMI).



(a) Coarse-Grained isolation (b) Fine-Grained isolation

Figure 4: Security lattice. Fine-Grained isolation restricts infomation flow even within the same address space. However parts of kernel need to be declassified to allow legitimate information flows (§ 5.2).

4.2 Security Lattice and Domain Propagation

Fig. 4 depicts the security lattice used for coarse- and fine-grained isolation. Fine-grained isolation, shown in Fig. 4b, needed in sandboxing environments like browsers and web servers, further restricts information flow from the engine to the sandboxes and across sandboxes, even within the same address space. Given attacks like Spectre/Meltdown that leak kernel memory, labeling all kernel memory as secret seems intuitive. However, this is inaccurate and can lead to high false positives. Several kernel functions legitimately write to user memory—such as setting up a new user-space page—and these writes should be explicitly declassified, allowing information flow to the user program without creating cycles of interference. We place such declassified parts of the kernel lower in the security lattice described in § 5.2.

Operation	Example	Security Tag Propagation
Arithmetic	`XOR x1, x2, x3`	`T[x1] ← max(T[x2],T[x3])`
Store	`STR x1, [x2, #Imm]`	`T[Mem[x2+Imm]] ← max(T[x1],T[CPU])`
Load	`LDR x1 ,[x2, #Imm]`	`T[x1] ← T[Mem[x2+Imm]]` `T[Mem[x2+Imm]] ← propagate(T[Mem[x2+Imm]],T[x2])`

Table 1: Security tag propagation rules. `Mem[ADDR]` **represents the content stored at** `ADDR`. `T[]` **represents the security tag of a register or a memory location.** `T[CPU]` **represents the security tag of the currently executing domain. Algorithm** `propagate` **is depicted in Fig. 5.**

```
1  def propagate(Tag[mem], Tag[op]):
2    if lattice_comparable(Tag[mem], Tag[op]):
3      return max(Tag[mem], Tag[op])
4    elif mem is shared:
5      return Tag[op]
6    else:
7      return Tag[mem]
```

Figure 5: Domain propagation function. `max` **returns the tag higher in the security lattice.**

Security domain tags are propagated throughout architectural and micro-architectural state to enable tracking of cyclic interference. We determine the security tag of each hardware unit based on the security domain that currently occupies it. We also determine the tag of each memory location based on the security domains of both the register and the memory operands of the instruction that most recently accessed this location.

Table 1 summarizes how a new security tag is computed for different operations. For arithmetic operations, the security domain of the destination register is the `max` of the operands (i.e., higher in the lattice). When operand domains are incomparable, `max` escalates to a lattice domain that is higher than all operands. For memory loads, depending on the security tags of both operands and the loaded data from memory, the memory security domain of the destination is updated using the *propagate* function depicted in Fig. 5: for comparable domains (i.e., there is a hierarchical relationship), it returns the one higher in the lattice; for incomparable domains it returns the security level of the addressing register used if the memory is shared, otherwise it returns the security level of the memory.

It is important to note that the propagation rules are extended for memory reads. Besides the destination register, memory reads update the security tag in the memory hierarchy depending on *whether micro-architectural state changes will be visible to other domains*. For example, for a memory location shared between incomparable domains a and b, a LLC miss for a load of a that uses data from domain b as the index will update the memory security tag in LLC to b. However, a hit retains the current memory security tag in the cache line.

4.3 Attack Examples

Next we describe how Prime+Probe and Flush+Reload exhibit cyclic resource and memory interference, respectively. Speculative attacks are analyzed in § 6. For each cache line, we save both the current and previous resource and memory domains, noted as $Blk(res\_dom, res\_dom', mem\_dom, mem\_dom')$ where $'$ represents the previous domain. For example, $Blk(a, b, a, b)$ depicts a cache line whose current domains on resource and memory are a, while the previous domains on resource and memory are b. X is used for don't-care or unknown domains. We assume no knowledge of the previous state of the cache. We refer to the attacker as security domain a and the victim as domain b. Fig. 6 shows how the state of domains is updated for a cache line under attack. Every iteration of Prime+Probe creates a cyclic resource interference (CRI) event, while Flush+Reload exhibits cyclic memory interference (CMI) whenever there is a `0-1` bit pattern.

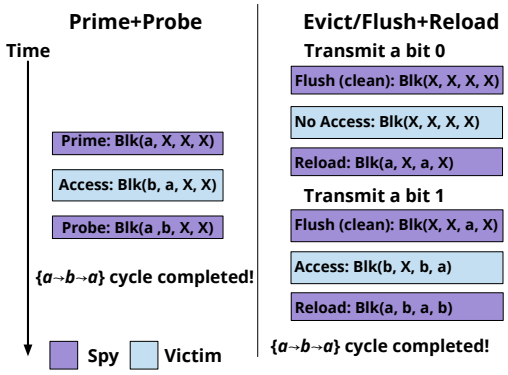


Figure 6: Description of how resource and memory domains are updated to discover cyclic interference in Prime+Probe and Flush+Reload. The metadata state of a cache line is notated as $Blk(res\_dom, res\_dom', mem\_dom, mem\_dom')$ **where** $'$ **represents the previous domain.** X **refers to a don't care or unknown domain. An occurence of cyclic interference is denoted as** $\{a \rightsquigarrow b \rightsquigarrow a\}$**.**

4.4 Challenges in Tracking Cyclic Interference

Labeling. To support both lattices in Fig. 4, we need to label different sandboxes as different domains but must allow some information flows, such as legitimate kernel writes to user memory, or explicit communication via shared memory or objects (**R1**, **2**, **4**).

Speculative Execution. Interference can result from speculative execution, where traditional MMU permission checks no longer guarantee confidentiality in the presence of micro-architectural side-channels. Thus each micro-architectural operation, speculative or not, needs to be tracked (**R3**).

Domain Propagation. Interference can occur on both hardware resources and memory addresses. Hence, besides information flow tracking on architectural state like registers and memory locations,

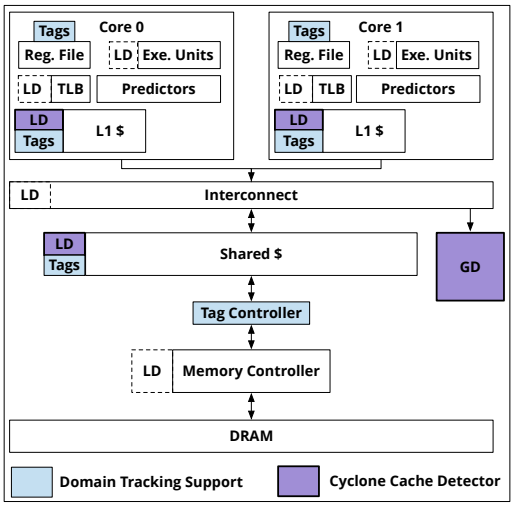


Figure 7: Processor Architecture with Cyclone. Our prototype places Local Detectors (LDs) on the caches for cyclic resource and memory interference, and modifies structures like the register file to store domains. Potential future work for Cyclone could place LDs on other units shown with dashed lines.

micro-architectural resources need to be tagged and tracked. Further, the domains needs to be propagated from architectural state, e.g. registers and memory, to micro-architectural state (**R3**, **4**).

Hardware Efficiency. Inteference is not explicitly expressed in the software of the victim and attacker. Thus, cyclic interference must be tracked in hardware efficiently (**R5**).

To overcome these challenges, we propose an architecture that properly propagates and tracks fine-grained domains throughout cores, caches, and memory.

5 CYCLONE ARCHITECTURE

Cyclone tracks both CRI and CMI in local detectors, and aggregates and filters alerts in a global detector. We first discuss hardware and software support in Cyclone to track directional interference among fine-grained security domains. Then we introduce our local detector prototype for contention-based cache information leaks which tracks cyclic interference. Importantly, Cyclone can still track cyclic interference in speculation-based attacks (**R3**), which manifest when a victim accesses a location based on attacker controlled inputs. Finally, we show how a global detector can be used to combine alerts and time series features from local detectors to create a robust, efficient anomaly detection system with low false-positives (**R4**, **5**).

5.1 Hardware Support

Fig. 7 shows an overview of the Cyclone architecture. The cores are modified to include a domain-id control register, add domain-ids to the register file, and add logic for propagating domains. The domain-id control register defines the security domain of the code currently executing. The register domain-ids are updated based on the lattice described in § 4. Memory requests carry a domain-id which corresponds to the security domain associated with the operands used to calculate the memory address. This mechanism enables Cyclone to track whether memory instructions are influenced by untrusted, attacker-controlled inputs or classified secrets.

To track cyclic interference, a history of the security domains on shared resources needs to be kept. Specifically, Cyclone implements tracking for current and previous domain-ids. We associate a current and previous **resource** domain as well as a current and previous **memory** domain with each cache line to track both *cyclic resource interference (CRI)* and *cyclic memory interference (CMI)*. Whenever a cache line is accessed, if the domain-id of the request differs from the current domain-id, the current is moved to previous and then updated based on our security lattice. Note that a particular deployment of Cyclone may choose to only track interference in the last-level cache, or only on the caches in certain cores depending on the threat model and desired performance overheads. Every cache line of main memory also has a tag associated with it that holds the security domain of the last access. Memory tags are implemented by reserving space at the end of DRAM and a simple mapping from cache line to tag. There are no new instructions for managing tags, memory requests have domains associated with them based on the domain register set by software and our propagation rules. This memory domain tracking enables detection of attacks that contend for specific memory *addresses* such as Flush+Reload and Evict+Reload in addition to resource contention attacks (**R4**). Note that when a cache hit for a load occurs we do not update the memory domain, just the resource domain.

Cyclone reduces the false positives in hardware (**R4**). First, Cyclone provides a shared bit in the domain-id for software to mark memory regions as shared. This allows explicit sharing between different domains by not propagating shared memory domains (Fig. 5 line 5). Second, Cyclone lowers the source register and memory security domains upon committed writes. Committing write instructions satisfy both hardware permission and software bounds checks, indicating that the domain writing memory has valid ownership of the data. This helps prevent over-tainting due to the lattice escalation policy. Finally, Cyclone provides support for domain *masking* used to explicitly allow contention between security domains. For example, it can be used to allow contention created by explicit communication in (e.g.) producer-consumer applications.

5.2 Software Support

Cyclone's software support enables defining fine-grained security domains to meet application-level security requirements. It provides system and application software with support to assign and manage domains. Although software compartmentalization [31, 85] is not the focus of this paper, we discuss our efforts to leverage such support.

Domain-ids are associated with particular *users* (e.g., tied to cloud account), and must be managed by the software that enforces isolation. For example, within the kernel, new processes are initialized with a user's security domain, and the domain-id control register is managed on context and mode switches. By default, the kernel sits higher in the security lattice than any user processes. For example, page tables are marked with kernel security domains and are confidential to user processes. However, not all data accessed by the kernel should be marked as classified since various kernel functions

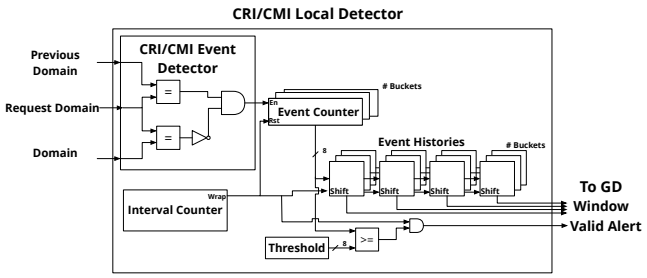


Figure 8: Cyclone cache local detector (LD) design. Each cache is augmented with two local detectors, one for cyclic memory interference and one for cyclic resource interference. The LD consists of one 32-bit interval counter, one 8-bit event counter, four 8-bit event history registers per-bucket, and logic to check thresholds and send alerts to the global detector.

operate on user memory or inputs. Cyclone supports a *declassification* operation for those functions. For example in `__copy_to_user`, store operations to user memory set the domain to unclassified. Specifically, we declassify the kernel's operations in the following scenarios: (1) process creation (e.g. fork), (2) memory management (e.g., zero-init), and (3) syscall (e.g., socket read/write).

Applications themselves must also be modified in a similar manner to the OS kernel if they implement intra-address space fine-grained isolation. In our evaluation (§ 8), we modify the implementation of the PhantomJS [34] browser to leverage this isolation. We assign all JavaScript sandboxes in the same frame the same security domain, while JavaScript sandboxes in different frames get distinct security domains (**R1**, **2**). This complies with the same origin policy, where JavaScript instances only have access to information within the same origin. The browser engine (WebKit) also has its own security domain such that it is isolated from every JavaScript sandbox. In addition, during transition from the WebKit engine to the JavaScript executor who executes the sandbox, common data structures and global objects used by both the WebKit engine and the JS executor have to be properly declassified [*]. We modified both the kernel and the JavaScript engine to ensure domains are properly labeled and declassified to reduce false-positives (**R4**).

5.3 Local and Global Detectors

Cyclone employs a combination of *local detectors* (LDs) and a *global detector* (GD) to allow trade-offs between an efficient design and detector performance.

Local detectors function similar to traditional performance counters, but instead are programmed to track cyclic interference with fine-grained domains. LDs are placed physically next to each cache and snoop on the traffic. Such snoopy detectors do not require modifications to the cache as the tracking logic is outside the monitored resource. They are responsible for storing the history of security domains and tracking cyclic interference. Each LD has counters that increment when cyclic interference events occur. An *interval counter* parameter determines the number of cycles between counter resets. Whenever a sample is collected, it is first investigated in the LD and possibly sent off to a GD which has abundant temporal information of the system for anomaly classification.

The GD in Cyclone is ultimately responsible for determining if the LD alerts are malicious. The GD consists of a classifier trained specifically for the user's deployed system and set of workloads. As samples are collected from the LDs, the GD aggregates and extracts features before performing inference using the classifier. The GD is intended to be programmable to enable the user to choose their classifier (SVM, CNN, LSTM, etc.) and feature extraction process to achieve their desired security requirements. We envision the GD similar to the power-management unit in a modern SoC; these units are often even implemented as full-fledged micro-controllers. Depending on the deployment scenario of a system utilizing Cyclone, a user may or may not require this level of programmability and could instead implement a fixed-function GD.

5.4 Cache Detector Prototype

Our LD cache detector prototype implements a set of counters (up to 32 in our prototype) that are shared by all the cache lines. Fig. 8 depicts the LD design for one event counter. *Bucketing* maps each line of the cache to a counter instead of keeping a counter for each cache set. When a cyclic interference event occurs, a simple hash function maps the address to a bucket and increments the counter. The LD functions by recording samples of the event counters on every interval counter reset into a N-sample Event History buffer. When a sample over a specified threshold is observed, an alert is raised, and that sample along with the previous **N** samples (referred to as a *window*) are sent to the GD for classification. Windowing is carried out to give the GD more context in predicting an anomaly, while trading-off LD storage and alert frequency sent to the GD. For our prototype we determine the threshold using the 99^{th} percentile of the distribution of counter values in the training set of benign applications (five in our experiments), and we choose a window size of four samples. Our GD prototype then computes statistics on the window (max and mean) and uses a one-class SVM model to classify the window using these features. The choice of number of buckets and sampling interval balances implementation overhead and desired accuracy. Sensitivity studies for these parameters are shown in § 8.5. Contention-only tracking uses the same local detector shown in Fig. 8, but only increments the event counter when the domain stored for that line differs from the current request domain. Contention without buckets has just one event counter for the whole cache instead of a set of counters.

6 CYCLIC INTERFERENCE IN ADVANCED ATTACKS

As Cyclone tracks information flow at the micro-architecture level, the domain propagation rules described in § 4.2 still apply to speculative instructions as long as they leave a trace in the micro-architecture. In this section, we discuss how cyclic interference enables Cyclone to meet **R3** in presence of Spectre and Meltdown as well as how cyclic interference discriminates the SpectreBenign example in § 3.2 (**R4**). We use the notation described in § 4.3 to

[*]Note that PhantomJS is controlled by a JavaScript controller that specifies which actions to take (e.g., the pages to visit). Since this script can access many internal data structures, we assign it the same security domain as the browser engine.

depict the state of the current and previous resource and memory domains of each cache line. In the following examples, we refer to the attacker as security domain a and the victim as domain b.

Spectre: Bounds Check Bypass. We use a Spectre variant with a Evict+Reload cache channel shown in Fig. 2 as an example. This variant uses 256 (2^8) different shared cache lines to transmit 1-byte of leaked information per iteration. The shared cache lines map one-to-one to the byte values. We show that cyclic interference on shared physical memory exists in Spectre when non-repeating bytes are leaked and transmitted.

Consider two iterations of an example Spectre attack leaking a sequence of two distinct bytes: $0x80$, $0x64$. At T_1, attacker domain a evicts all shared cache lines out of the cache, and subsequently supplies a malicious input x to the victim code, executed by domain b. At T_2, the victim mispredicts the branch and executes the wrong path speculatively. The first load, indexed by the malicious input x, brings in a secret byte, $0x80$ from memory. Then the second memory read, using the secret byte as an index, subsequently loads in a shared cache line mapped to the byte $0x80$. At T_3, the attacker scans through the shared memory region, testing each byte value, including $0x64$ and $0x80$. The read to the shared cached line mapped to $0x80$ hits in cache. The access to the memory block mapped to $0x64$ misses in cache. At T_4, attacker again evicts the shared cache lines, and feeds the victim code with a malicious input x', pointing to the next secret byte, $0x64$. At T_5, the victim speculatively brings in this byte and then loads a shared cache line mapped to the byte $0x64$. Finally, at T_6, attacker probes the 256 shared cache lines. This time, access to the shared line mapped to byte $0x64$ hits in cache.

From T_1 to T_6, the Spectre attack transmits two byte values $0x80$ and $0x64$. Interference on shared physical memory occurs as follows: at T_3, due to the miss on the cache line mapped to $0x64$, domain a reads from main memory, updating the in-main-memory security tag of this memory location to domain a as it modifies the micro-architectural state of the shared memory location. When domain b accesses this memory block from main memory at T_5, the state of the cache line brought in is $Blk(b,\ X,\ b,\ a)$. Then at T_6, domain a hits on this cache line and the state becomes $Blk(a,\ b,\ a,\ b)$, creating cyclic interference on shared memory as $\{a \rightsquigarrow b \rightsquigarrow a\}$.

Meltdown. Meltdown reads classified kernel memory from a user program. Similarly to Spectre, Meltdown uses a Flush+Reload channel. However, the FLUSH, ACCESS, and RELOAD phases are all performed by the attacker whose domain a is lower than the kernel domain b in the lattice, i.e., $a \subset b$.

During the ACCESS phase, the first load to classified kernel memory propagates the kernel domain b to the destination register. In the second load, the attacker uses this register to speculatively access a memory location. Based on the propagation rule (Fig. 5 line 3 since $a \subset b$), the loaded cache line will also have the tag b. Thus when attacker performs Flush+Reload on this cache line, cyclic interference exists $\{a \rightsquigarrow b \rightsquigarrow a\}$.

SpectreBenign: In-Bound Array Accesses. SpectreBenign in Fig. 2, without the RELOAD phase, does not infer any secrets from the victim. Interference may still exist but does not form a cycle. Further, if the `array1` is unclassified or shared, in-bound accesses do not leak information. Consider a benign situation where every access of the victim is in-bound to a shared location. During the ACCESS phase: the victim's first load, addressed by inputs from the attacker (domain a), accesses an in-bound shared memory location. Because the micro-architectural state of this shared cache line is affected by domain a, the loaded cache line and the destination register will both be tagged as a (Fig. 5 line 5). The second load, using this register (domain a) as address, reads a shared memory `array2` and the loaded line is again tagged as a. Thus, cyclic interference will not take place on the shared `array2`.

7 EXPERIMENTAL METHODOLOGY

7.1 System Configuration

Arch	ISA	ARM v8
Core	Frequency	3GHz
	BPred	2048-Entry BiModal Predictor
	Fetch	8 wide, 64-entry IQ
	Issue	8 wide, 192-entry ROB
	Writeback	8 wide, 32-entry LQ, 32-entry SQ
Memory	L1-I, L1-D	64kB, 8-way, 64B line, 2cycles, LRU, 4 20-entry MSHR, no prefetch
	L2 (LLC)	2MB, 16-way, 64B line, 20 cycles, stride prefetch, LRU, 8-entry write-buffer, 4 20-entry MSHR
	DRAM	16GB Micron-MT40A2G4 DDR4

Table 2: Gem5 System Configurations

We implement Cyclone's design using full-system gem5 [8] configured to simulate an out of order (OoO) processor running the ARMv8 ISA. Table 2 shows the specific configuration used in our experiments. We choose parameters to be similar to a modern ARM OoO core such as the Cortex-A73. Note that our design could be ported to any ISA. We simulate ARM due to superior support for full-system mode in gem5 which enables us to perform end-to-end real-world security evaluations with complex applications such as SQL databases and web browsers.

7.2 Workloads

Our workloads are divided into two sets: one that evaluates Cyclone's detection ability and another that evaluates overheads.

Our first set of security-centric workloads are applications that operate on potentially sensitive information. These include classification (libSVM: classifies sensitive inputs), object recognition (OpenCV: operates on sensitive images), and a medical database (PostgreSQL: operates on sensitive patient information). Each of these applications is modified to embed a per-set Prime+Probe covert-channel on the LLC that leaks bits across isolation boundaries to a co-resident *receiver*. The receiver process can then reconstruct the sensitive information. These bit leaks are interleaved with normal benign application behavior. In our experiments, we use inter-process communication between the covert channel gadgets to synchronize the leak phases. This assumes a more powerful attacker than one who must synchronize by exercising the channel.

Our next security-centric workload is PhantomJS [34], a headless browser that allows automating page loads using JavaScript. As discussed in § 2.1, a browser is an application that implements fine-grained isolation between potentially many domains. Thus PhantomJS enables us to exercise the ability of our detector to scale to many domains while still maintaining low false positives

(**R1**, **4**). To this end, we implement a hardware attack vector that violates fine-grained isolation in PhantomJS—*Spectre-V1 bounds check bypass*. In addition to the attacks, we simulate benign executions consisting of PhantomJS visiting various websites collected by Gutierrez *et al.* [32].

To better assess the value of Cyclone, in addition to tracking cyclic interference, we implement contention-tracking similar to prior work [13, 38]. Rather than counting cyclic interference events, contention-tracking only discovers when there is directional interference with *any other* domain, and does not discover cycles. Further, we apply our *bucketing* scheme to the contention-based detectors to track at a spatially finer-granularity. In all security evaluations, we compare tracking cyclic interference to contention and bucketed contention.

Our performance workloads include a representative set of five SPECspeed 2017 applications (cactuBSSN, mcf, leela, xz, and deepsjeng) taken from [68]. The performance workloads enable us to evaluate the overhead of Cyclone and the presence of false-positives when there is no attack. For the mixed SPEC workloads, we fast-forward all benchmarks past the initialization point of the application that takes the longest to initialize (cactuBSSN). We then warm-up for 50 million cycles and simulate detailed statistics for 1 billion cycles. For non-SPEC workloads, we fast-forward past initialization and then run to completion. All experiments are performed in gem5 full-system mode, running Ubuntu 16.04 with Linux kernel v4.15 (the most recent kernel supported by gem5 with ARM at time of writing).

8 RESULTS AND ANALYSIS

In this section, we analyze our detector's performance on both resource and memory based (e.g., Flush+Reload) micro-architectural information leak attacks on cache. We compare tracking of cyclic interference with contention and bucketed contention, as described in § 5.2 and 7.2. We report classification metrics such as precision, false positive rate, and F1 score. Recall describes the accuracy of a detector in identifying all attacks instances, while false positive rate depicts its practicality. F1 score is the harmonic mean of precision and recall. We omit ROC curves since there are very few malicious samples compared to benign. The covert-channel experiments group results into leak frequencies which are based on how many cycles an attacker chooses to transmit a single bit. In addition to detector effectiveness, we analyze the performance trade-off of our detector prototype due to extra DRAM traffic and if the detector tracking area were instead devoted to storing data in the cache.

8.1 Case Study 1: False Positives Stress Test

In this case study, we run several benign applications concurrently in order to quantify false positives in Cyclone. Specifically, we simulate a mix of 1 to 5 SPEC2017 programs (§ 7.2) on a shared machine with Cyclone. Each application is assigned a unique domain ID from the kernel and we run a max of 6 concurrent applications, including the kernel, on our four-core simulation. Fig. 9 (top) shows a heat map of cyclic interference observed in the L2 cache over 80 million cycles simulated in region of interest. The cyclic interference is aggregated into 32 cache buckets, shown on the y-axis. A darker color indicates higher intensity of the interference. The

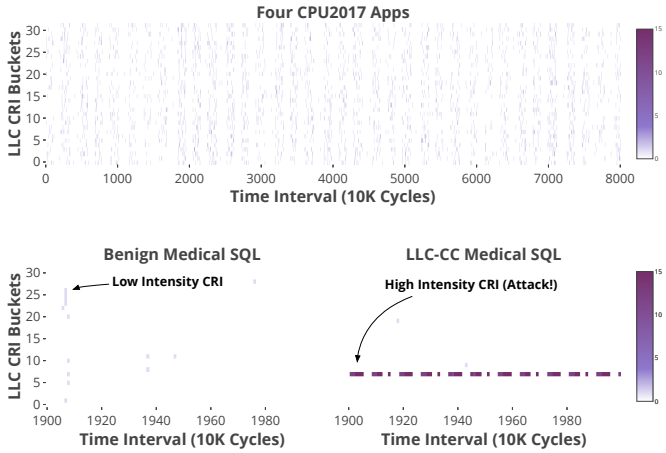


Figure 9: HeatMap of cyclic resource interference (CRI) in L2 Cache sets on a mix of four SPEC2017 programs and on a medical SQL queries. Benign SPEC and medical show only low intensity CRI, but the medical with Prime+Probe covert-channel show clear high intensity CRI during the leak phases.

benign SPEC mixed workload exhibits only low intensity cyclic interference events (less than 6 events per 10k cycles).

8.2 Case Study 2: Information Leaks in Privacy-Sensitive Applications

We embed a cross-process LLC Prime+Probe covert-channel into three privacy-sensitive applications: SVM classification, facial recognition, and a medical SQL database. Note that a high-bandwidth, cross-process Prime+Probe attack can also be performed on the private L1 cache in systems which support simultaneous multithreading (SMT), however our gem5 simulation is limited to one hardware thread per core. Fig. 9 (bottom) shows the frequency of cyclic interference on each of the 32 LLC cache buckets throughout the program run on our medical database, sampled at a granularity of 10k cycles. There is very low intensity of cyclic resource interference (CRI) on the benign application run shown on the left, while the leak phases of the covert-channel on the right are clearly visible with high intensity CRI. For brevity we only show the heatmap for the medical database. SVM classification and face recognition show similar cyclic interference behavior.

As described in § 5.2, our LD+GD setup consists of simple thresholding and windowing at the LD combined with a one-class SVM classifier that uses `max` and `mean` features on the windows at the GD. The interval size between counter samples used in this case-study is 10k and the window size is four.

Table 3 shows detector results for the Prime+Probe covert-channel on three privacy-sensitive applications for generic detectors. The table also shows the average performance of them on benign SPEC2017 workloads (3 different mixes of 4 SPEC apps running on 4 cores). Fig. 10 plots the F1 Scores for the generic and application-specific detectors. Testing per malicious app yields similar results with bucketed contention and CRI. The trade-off between

Attack Scenarios			Contention Detector			Contention Buckets Detector			Cyclone		
App	Attack Bandwidth (bps)	Cyles Per Bit Leak	Alerts/s	Precision	Missed Attacks/s	Alerts/s	Precision	Missed Attacks/s	Alerts/s	Precision	Missed Attacks/s
LibSVM	2391k	1000	253k	0.9334	23.6	235k	0.9998	0	235k	0.9998	0
	239.1k	10000	246k	0.9689	550.1	199k	1	0	199k	1	0
Medical DB	90k	1000	159k	0.0574	4.6	9k	0.9987	0	9k	0.9999	1.8
	9k	10000	158k	0.0538	575.8	4k	0.9985	0	4k	1	0.9
Face Detection	99k	1000	16k	0.6029	0	10k	0.9992	0	10k	0.9999	0
	9.9k	10000	16k	0.6165	0	9k	0.9991	0	9k	1	0
Benign SPEC2017	0	0	300k	0	0	12k	0	0	12	0	0

Table 3: Generic, unsupervised detection performance of Cyclone with cyclic resource interference (CRI) against Prime+Probe covert-channel attacks. Each row in this table describe how Cyclone performs given a specific attack scenario. For example, the second row in the table reports that when Prime+Probe covert-channel in LibSVM uses 1000 cycles to leak each single bit, achieving a bandwidth at 2391kbps, contention based detectors report 253k alerts per second, 93% of which are true alarms, missing 23.6 attacks per second. However, both our Contention Buckets Detector and Cyclone report around 235k alerts per second, 99.98% of which are true alerts, missing no attacks. Since, missed attacks captures false negatives, we omit recall here for brevity.

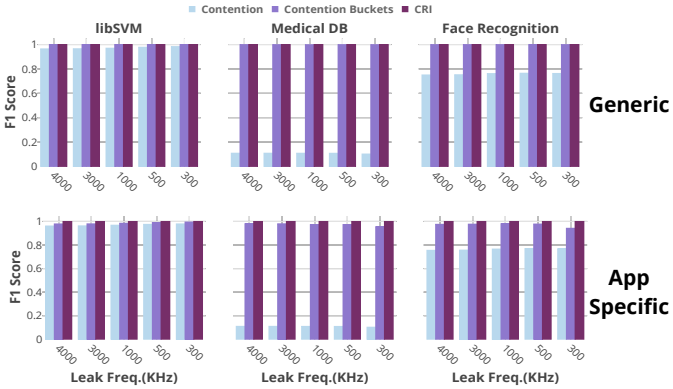


Figure 10: Generic (top) and application-specific (bottom), unsupervised detector F1 scores. Tracking cyclic resource interference improves detection results compared to both contention and bucketed contention tracking. Leak frequency refers to the amount of cycles an attacker chooses to transmit a single bit.

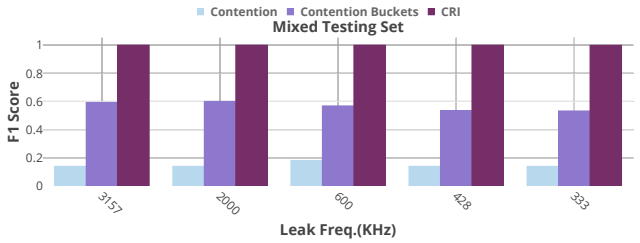


Figure 11: Generic, unsupervised detector F1 scores. We report results for a sample of mixed benign and malicious workloads. While contention tracking with buckets improves significantly upon normal contention tracking, cyclic resource interference achieves near perfect F1 scores.

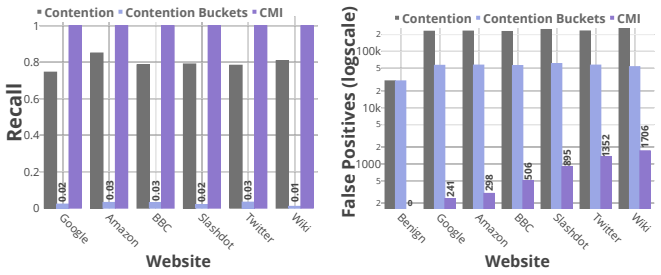
the bucketed contention detector and Cyclone lies in between false positives (reduced precision) and false negatives (missed attacks). Cyclone has better precision and recall, but incurs a few false negatives. Bucketed contention, on the other hand, tolerates reduced precision in lieu of eliminating all false negatives. However, the real advantage of tracking cyclic interference shines in benign workloads which exhibit significant contention (12,000 alerts/second compared to only 12).

We simulate a setting of diverse benign and malicious workloads by running a mix of concurrent SPEC applications, time-interleaved with all three privacy-sensitive applications compromised with a Prime+Probe covert-channel. Overall, we create five mixed testing sets each of which contain executions of four concurrent SPEC apps (on all 4 cores). The testing sets also consist of SVM classification, face detection, and a medical DB, all running Prime+Probe at a particular leak frequency. The four SPEC workloads used for each mixed testing set are chosen out of the five SPECspeed 2017 applications mentioned in § 7.2. Global detectors based on all three designs (contention, bucketed contention, and CRI) are trained to cover a range of benign behaviors: a SPEC workload mix of 1-5 concurrently running applications and benign executions of the privacy-sensitive applications. Fig. 11 plots the F1 scores for these mixed testing workloads. Cyclic interference is not affected by the presence of diverse benign samples in the workload whereas both types of contention tracking exhibit a large number of false positives. We conclude that one-way directional interference occurs much more frequently than cyclic interference in benign applications, making cyclic interference a more robust signal for detecting cache contention leaks.

8.3 Case Study 3: Remote Attacks in Sandboxes with Fine-Grained Isolation

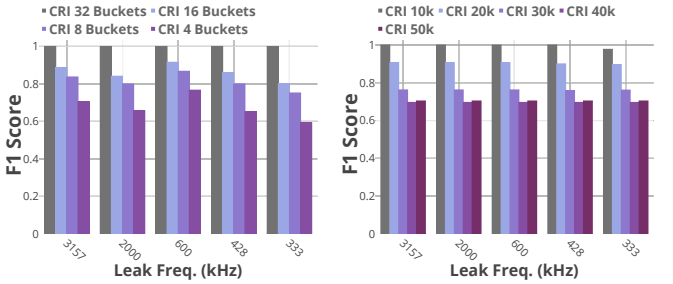
To evaluate Cyclone in a fine-grained isolation setting, we use PhantomJS with 9 different benign websites[†] [32], Spectre, and SpectreBenign (described in § 3.2) implemented in JavaScript. After modifying PhantomJS to assign distinct domains for each website

[†] Amazon, BBC, CNN, CraigslistNY, eBay, Google, Slashdot, Twitter, and Wikipedia



(a) Recall (Higher is better) (b) False Positives (Lower is better)

Figure 12: Generic, unsupervised detector recall and number of false positives for Spectre in a web browser. Compared to pure contention tracking, Cyclone achieves higher recall and lower false positives by tracking cyclic memory interference(DMI). Notably, Cyclone detects all byte leaks (recall is always 1), and creates zero false positives when there's no attack.



(a) Bucket sensitivity (b) Sampling Sensitivity

Figure 13: Sensitivity studies for Cyclone. A larger number of buckets and finer-granularity interval sizes improve the F1 scores of our detector. However, these improvements come at a cost of more area for counters and increased traffic to the global detector.

frame, loading each website involves on average 7 security domains. The rendering of websites experiences on average 93.2 security domain switches per second.

We implement a cross-site password stealing attack using Spectre. Rather than searching for a usable Spectre gadget in PhantomJS, we extend the JavaScript API to include a function that is vulnerable to a Spectre variant similar to Fig. 2, i.e. bounds-check-bypass [49]. In addition, we assume that the victim's browser does not support strict Site Isolation (one process per origin even in the same page) and that the attacker has prior knowledge about the memory locations of the passwords. A remote attacker exploiting JavaScript does not have direct access to cache flush instructions, and thus performs an Evict+Reload attack. Our Spectre implementation enables the attacker to read arbitrary memory contents within the browser process's address space, bypassing the isolation boundaries of the JavaScript sandbox. We apply the attack to 9 different benign web sites[†]. It fails on 3 of them due to unintentional cache evictions created by the browser. On the remaining 6 websites, Spectre successfully steals a total of 672 bytes at varying attack bandwidths ranging from 110.1 to 115.2 bits/second.

We configure our cache detector LD to sample every 100k cycles, while the window size is kept at four. To detect every byte leak, we set no thresholds on the CMI at the local detector. Fig. 12 depicts the detection performance of Cyclone with CMI-based LDs in a generic, unsupervised setup, against the remote Spectre variants on 6 different websites. Results are compared to contention tracking without bucketing and with 32 buckets. Unlike case study 2, where a contention bucket detector detects almost all covert-channel attacks but with higher false positives, contention-based detectors fail to detect all spectre attacks. On the y-axis we plot recall and false positives to better visualize this difference. On the x-axis, we list the 6 different websites being attacked. Compared to contention tracking with and without buckets, Cyclone achieves higher recall and lower false positives.

Fig. 12a shows that Cyclone detects all byte leaks (recall is always 1). However, the recall for tracking contention with buckets degrades compared to without buckets. This degradation is due to contention being prevalent even in benign applications. Bucketing disaggregates the attacker's contention across all sets, reducing the interference to a benign intensity. Thus the bucketed contention detector fails to classify the attacks as anomalous. Meanwhile, Fig. 12b shows the number of false positives plotted on a logarithmic scale. The first group, at 0 bps effective bandwidth, shows the detector false positives on SpectreBenign. In this benign setting, both types of contention tracking report large number of false alarms, while CMI introduces no false positives. Cyclone with CMI detects 100% of the attacks, compared to 80% detection using contention tracking, while reducing false positives by 260×. Further, we verify that on the 3 failed attacks, CMI creates no additional false positives.

8.4 Case Study 4: Sensitivity Studies

We perform sensitivity studies in order to arrive at the best parameters for Cyclone. Fig. 13 summarizes our system's sensitivity to the number of counter buckets and the sampling granularity. We experiment with 4 to 32 buckets and a sampling granularity between 10k to 100k cycles. We show our results on the same mixed testing sets from Fig. 11. We observe that the F1 score of a CRI-based detector increases with increasing number of buckets. While having one bucket per set would be ideal, the area overheads would be too large for a realistic implementation (e.g., 2048 counters for the LLC only in our prototype). We also note that there is no need to strive for such fine-grained counters since we already achieve near perfect F1 scores with 32 buckets. As expected, finer-grained time-sampling increases accuracy. However, our experiments sampling at a finer granularity than 10k cycles fail to improve the scores dramatically, thus we perform all of our experiments with a 10k sampling frequency and using 32 buckets. Additionally, finer-grained interval sizes will increase the amount of traffic sent to the GD.

8.5 Performance Evaluation

We use the SPEC2017 performance workloads to evaluate the performance impact of Cyclone when no channel is being created. First, we evaluate the overhead of extra DRAM transactions to read and update memory domains which are stored at the end of the address space. Next, we compared IPC to a system where the

detector area used for domain tracking is instead used for data in the cache. Finally, we compare the performance to a system that does way-partitioning for each domain. We did not explicitly model interconnect traffic due to alerts sent to the GD. However, our alert message is 4 bytes leading to only 48 bytes/second in the benign case for the 12 alerts/second shown in Table 3. Context-switching the domain-id control register requires only four instructions to save and switch to the kernel domain-id and two instructions to restore; regardless of the number of concurrent domains. We evaluated single SPEC benchmarks without our kernel changes and saw negligible (<0.001 IPC) performance change. As discussed in § 8.3, the JavaScript engine switches sandboxes only 93.2 times per second (much less frequent than the kernel), thus has less overhead than the kernel.

Cache Overheads. As discussed in § 5, each 64B cache line is tagged with four 8-bit wide domain-ids. This choice adds an overhead of 32-bits per cache line, leading to a worst-case space overhead of 1 way for the L1 and 2 ways for the LLC based on the sizes in Table 2. Thus the area overhead for Cyclone is estimated at 6.25% of the total cache data-store, 32 bits of meta-data for every 64 bytes of cache data. Our experiments on SPEC benchmarks show that the IPC speedup over our system achieved by allocating the extra area to data instead of tags is about 1.2% on average without any partitioning and remains approximately the same for a per-core partitioned cache. However for fine-grained isolation, we would need many cache partitions to prevent information leakage between sandboxes. The same workloads running on an 8-way partitioned L1 and 16-way partitioned L2 see a slowdown of up to 33% on average with a max slowdown of 40%. Further increasing partitioning will cause more performance degradation, whereas Cyclone is only limited by the choice of domain-id width.

DRAM Overheads. The DRAM overhead to store 8-bit domain tags for every 64 bytes is 1.5% of the total size. Our DRAM tag transactions result in a 2.4% IPC decrease on SPEC. Note that the only optimization we implement is eliminating writes that don't change the domain (i.e., silent writes). We do not currently implement tag caching or tag compression which have been shown to significantly reduce this overhead [42, 74].

Detector Overheads. Since the local detector works in parallel with the corresponding domain-id, the entire monitoring system is off the critical path of the processor. The global detector contains a pre-trained SVM model, the maximum size for which is 311 bytes in our simulations. Due to this small size, we did not directly model any potential system effects such as memory bandwidth for the initial (e.g., at boot-time) GD setup. The GD can be implemented as a programmable micro-controller or as custom hardware. One possible implementation is a systolic array architecture computing the dot product between the support and test vectors with a 2-dimensional array of MAC units that uses a scratchpad and FIFO buffers for storage [52].

9 DISCUSSION & RELATED WORK

Discussion. To evade Cyclone, an attacker can spread the cycles over time-windows to keep the values closer to benign interference, which greatly reduces bandwidth (down to single bits per second in our experiments). Alternatively, an attacker can eliminate cycles of interference by colluding with another security domain, attempting to mislead a defense when it initializes the security label lattice. To avoid collusion, security domains must be set up to correctly reflect trust assumptions (e.g., each identity/origin gets a single label that is assigned to all VMs spawned by that origin). Identities thus rely on real-world constraints such as credit card numbers used to purchase cloud services to throttle unconstrained collusion among labels. Alternatively, the system must be configured to isolate each domain from the union of all other domains. Cyclone provides alerts that can be combined with OS-level signals (e.g., from osquery [39]) and handled in software [62]. We expect an end-to-end attack will exhibit behavior such as filesystem and network activity that can be used to drive down our false-positive rate. We note that processing a few alerts per second will have a negligible impact on overall system performance.

Related Work. Cyclone improves upon contention-tracking detectors [13, 38] by exploring the opportunity to track cyclic interference and introducing an efficient distributed detection architecture. OS techniques like timing protection [27] can be used in addition to Cyclone to trade-off overheads in various settings.

Rather than detect when a side-channel is exploited at run-time, some approaches aim to verify there is no information leakage in the micro-architecture [24, 25, 41, 75, 76, 92]. Deng et. al. [19] introduce a three-step model that verifies the existence of side-channels in secure-cache designs by searching through possible execution paths; this work identifies a similar flow of interference.

Tagged architectures have a long history dating back to the 60s and 70s [26], with use cases such as capabilities [36, 88], memory safety [64, 87], and taint tracking [16, 72]. Compared to traditional page-sized metadata (e.g., RWX permissions), some of these systems propose efficient mechanisms for finer-granularity metadata [42, 87]. Cyclone leverages these principles to enable cache-line granularity tagging of memory, as well as tagging of resources in the micro-architecture. Other systems [58] utilize similar micro-architectural tagging for quality-of-service instead of domain tracking. Cyclone could repurpose the tagging support provided by these architectures to implement interference tracking. For example [16] provides 4 bits per memory word which could be used to track domains per cache line.

10 CONCLUSION

We introduce Cyclone, a distributed anomaly detection system that tracks a new property called cyclic interference which is common to all known contention-based cache information leaks. Our extensive full-system evaluation with browser- and core-level isolation shows that cyclic interference is extremely resilient to false positives while remaining highly accurate. Cyclone provides a new conceptual direction for achieving protection of micro-architectural structures, especially in fine-grained isolation scenarios where partitioning across tens of security domains does not scale.

11 ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by DARPA under Contract HR0011-18-C-0019 and by NSF award #1453806.

REFERENCES

[1] Onur Aciiçmez. 2007. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/1314466.1314469

[2] Onur Aciiçmez and Werner Schindler. 2008. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'08)*. Springer-Verlag, Berlin, Heidelberg, 256–273.

[3] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-Based Malware Detection Using Dynamic Analysis. *J. Comput. Virol.* 7, 4 (Nov. 2011), 247–258. https://doi.org/10.1007/s11416-011-0152-x

[4] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *2015 IEEE Symposium on Security and Privacy*. 623–639. https://doi.org/10.1109/SP.2015.44

[5] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/106972.106984

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[7] D.J. Bernstein. 2005. Cache-timing Attacks on AES. (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, and Somayeh Sardashti. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[9] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 1–1.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv:1811.05441 [cs]* (Nov. 2018). arXiv:cs/1811.05441

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. *arXiv:1802.09085 [cs]* (Feb. 2018). arXiv:cs/1802.09085

[13] Jie Chen and Guru Venkataramani. 2014. CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 216–228. https://doi.org/10.1109/MICRO.2014.42

[14] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters. *Appl. Soft Comput.* 49, C (Dec. 2016), 1162–1174. https://doi.org/10.1016/j.asoc.2016.09.014

[15] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining Specifications of Malicious Behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1287624.1287628

[16] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 482–493. https://doi.org/10.1145/1250662.1250722

[17] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security*. IEEE, 0.

[18] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 559–570. https://doi.org/10.1145/2485922.2485970

[19] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2018. Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18)*. ACM, New York, NY, USA, 2:1–2:8. https://doi.org/10.1145/3214292.3214294

[20] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

[21] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 51–67.

[22] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Trans. Archit. Code Optim.* 8, 4, Article 35 (Jan. 2012), 35:1–35:21 pages. https://doi.org/10.1145/2086696.2086714

[23] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. Lattice Priority Scheduling: Low-Overhead Timing-Channel Protection for a Shared Memory Controller. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. 382–393. https://doi.org/10.1109/HPCA.2016.7446080

[24] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 555–568. https://doi.org/10.1145/3037697.3037739

[25] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1583–1600. https://doi.org/10.1145/3243734.3243743

[26] E. A. Feustel. 1973. On The Advantages of Tagged Architecture. *IEEE Trans. Comput.* C-22, 7 (July 1973), 644–656. https://doi.org/10.1109/TC.1973.5009130

[27] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, 1:1–1:17. https://doi.org/10.1145/3302424.3303976

[28] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2010. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Proceedings of the 12th International Conference on Information Security and Cryptology (ICISC'09)*. Springer-Verlag, Berlin, Heidelberg, 176–192.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721 (DIMVA 2016)*. Springer-Verlag, Berlin, Heidelberg, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 897–912.

[31] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[32] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. 2011. Full-system analysis and characterization of interactive smartphone applications. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 81–90. https://doi.org/10.1109/IISWC.2011.6114205

[33] Zecheng He and Ruby B. Lee. 2017. How secure is your cache against side-channel attacks?. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. 341–353. https://doi.org/10.1145/3123939.3124546

[34] Ariya Hidayat. 2013. PhantomJS: Headless Webkit with Javascript API. *WSEAS Trans. Commun* (2013), 457–477.

[35] Jann Horn. 2018. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[36] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. 1981. IBM System/38 Support for Capability-Based Addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 341–348.

[37] Wei-Ming Hu. 1991. Reducing Timing Channels with Fuzzy Time. In *IEEE Symposium on Security and Privacy*. 8–20. https://doi.org/10.1109/RISP.1991.130768

[38] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. 2015. Understanding Contention-Based Channels and Using Them for Defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium On*. 639–650. https://doi.org/10.1109/HPCA.2015.7056069

[39] Facebook Inc. 2019. osquery. https://osquery.io.

[40] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP*

'15). IEEE Computer Society, Washington, DC, USA, 591–604. https://doi.org/10.1109/SP.2015.42

[41] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. 2018. High-Level Synthesis with Timing-Sensitive Information Flow Enforcement. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. ACM, New York, NY, USA, 88:1–88:8. https://doi.org/10.1145/3240765.3243415

[42] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*. 641–648. https://doi.org/10.1109/ICCD.2017.112

[43] Mehmet Kayaalp, Nael B. Abu-Ghazaleh, Dmitry V. Ponomarev, and Aamer Jaleel. 2016. A High-Resolution Side-Channel Attack on Last-Level Cache. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. 72:1–72:6. https://doi.org/10.1145/2897937.2897962

[44] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael B. Abu-Ghazaleh, Dmitry V. Ponomarev, and Aamer Jaleel. 2017. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. 7:1–7:6. https://doi.org/10.1145/3061639.3062313

[45] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. 2016. Quantifying and Improving the Efficiency of Hardware-Based Mobile Malware Detectors. In *Proceedings of the 49th International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwain, October 15-19, 2016*.

[46] Khaled N. Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. 2017. RHMD: Evasion-Resilient Hardware Malware Detectors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 315–327. https://doi.org/10.1145/3123939.3123972

[47] Khaled N. Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2015. Ensemble Learning for Low-Level Hardware-Supported Malware Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404 (RAID 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 3–25. https://doi.org/10.1007/978-3-319-26362-5_1

[48] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Computer Society, Fukuoka, Japan.

[49] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203 [cs]* (Jan. 2018). arXiv:cs/1801.01203

[50] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 104–113. https://doi.org/10.1007/3-540-68697-5_9

[51] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD.

[52] C. Kyrkou and T. Theocharides. 2012. A Parallel Hardware Architecture for Real-Time Object Detection with Support Vector Machines. *IEEE Trans. Comput.* 61, 6 (June 2012), 831–842. https://doi.org/10.1109/TC.2011.113

[53] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 549–564.

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 973–990.

[55] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. 406–418. https://doi.org/10.1109/HPCA.2016.7446082

[56] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. 203–215. https://doi.org/10.1109/MICRO.2014.28

[57] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[58] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, Lixin Zhang, and Yungang Bao. 2015. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/2694344.2694382

[59] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 311–324. https://doi.org/10.1145/2508859.2516692

[60] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

[61] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre Is Here to Stay: An Analysis of Side-Channels and Speculative Execution. *arXiv:1902.05178 [cs]* (Feb. 2019). arXiv:cs/1902.05178

[62] Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. 2015. Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1595–1606. https://doi.org/10.1145/2810103.2813706

[63] A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 421–430. https://doi.org/10.1109/ACSAC.2007.21

[64] Santosh Nagarakatte, Milo Martin, and Steve Zdancewic. 2013. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro* 33, 3 (May 2013), 38–47. https://doi.org/10.1109/MM.2013.26

[65] Michael Neve and Jean-Pierre Seifert. 2007. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography (SAC'06)*. Springer-Verlag, Berlin, Heidelberg, 147–162.

[66] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[67] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. 2015. Malware-Aware Processors: A Framework for Efficient Online Malware Detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 651–661. https://doi.org/10.1109/HPCA.2015.7056070

[68] R. Panda, S. Song, J. Dean, and L. K. John. 2018. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 271–282. https://doi.org/10.1109/HPCA.2018.00032

[69] M. K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 775–787. https://doi.org/10.1109/MICRO.2018.00068

[70] Charlie Reis. 2018. Mitigating Spectre with Site Isolation in Chrome. https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html.

[71] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding Information Leakage in the Memory Controller with Fixed Service Policies. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. 89–101. https://doi.org/10.1145/2830772.2830795

[72] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[73] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2014. Unsupervised Anomaly-Based Malware Detection Using Hardware Features. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*. 109–129. https://doi.org/10.1007/978-3-319-11379-1_6

[74] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. 2008. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 94–105. https://doi.org/10.1109/MICRO.2008.4771782

[75] M. Tiwari, X. Li, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. 2010. Gate-Level Information-Flow Tracking for Secure Architectures. *IEEE Micro* 30, 1 (Jan. 2010), 92–100. https://doi.org/10.1109/MM.2010.17

[76] C. Trippel, D. Lustig, and M. Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 947–960. https://doi.org/10.1109/MICRO.2018.00081

[77] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv:1802.03802 [cs]* (Feb. 2018). arXiv:cs/1802.03802

[78] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (Jan. 2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y

[79] Guru Venkataramani, Jie Chen, and Milos Doroslovacki. 2016. Detecting Hardware Covert Timing Channels. *IEEE Micro* 36, 5 (Sept. 2016), 17–27. https://doi.org/10.1109/MM.2016.83

[80] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. 2014. Timing Channel Protection for a Shared Memory Controller. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. 225–236. https://doi.org/10.1109/HPCA.2014.6835934

[81] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. 74:1–74:6. https://doi.org/10.1145/2897937.2898086

[82] Y. Wang, B. Wu, and G. E. Suh. 2017. Secure Dynamic Memory Scheduling Against Timing Channel Attacks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 301–312. https://doi.org/10.1109/HPCA.2017.27

[83] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. 494–505. https://doi.org/10.1145/1250662.1250723

[84] Zhenghong Wang and Ruby B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 83–93. https://doi.org/10.1109/MICRO.2008.4771781

[85] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. 2016. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro* 36, 5 (Sept 2016), 38–49. https://doi.org/10.1109/MM.2016.84

[86] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[87] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 304–316. https://doi.org/10.1145/605397.605429

[88] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468.

[89] J-Y. Xu, A. H. Sung, P. Chavez, and S. Mukkamala. 2004. Polymorphic Malicious Executable Scanner by API Sequence Analysis. In *Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS '04)*. IEEE Computer Society, Washington, DC, USA, 378–383. https://doi.org/10.1109/ICHIS.2004.75

[90] Zhixing Xu, Sayak Ray, Pramod Subramanyan, and Sharad Malik. 2017. Malware Detection Using Machine Learning Based Analysis of Virtual Memory Access Patterns. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '17)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 169–174.

[91] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 719–732.

[92] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 503–516. https://doi.org/10.1145/2694344.2694372

[93] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*. 118–140. https://doi.org/10.1007/978-3-319-45719-2_6

[94] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2976749.2978360

[95] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2382196.2382230

[96] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 827–838. https://doi.org/10.1145/2508859.2516741

[97] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware Performance Counters Can Detect Malware: Myth or Fact?. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. ACM, New York, NY, USA, 457–468. https://doi.org/10.1145/3196494.3196515

[98] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. 2017. Camouflage: Memory Traffic Shaping to Mitigate Timing Attacks. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017 (HPCA '17)*. IEEE Press, Piscataway, NJ, USA.