COTSknight: Practical Defense against Cache Timing Channel Attacks using Cache Monitoring and Partitioning Technologies

Fan Yao†, Hongyu Fang, Miloš Doroslovački and Guru Venkataramani
Department of Electrical and Computer Engineering,
The George Washington University, Washington, DC, USA
Email: {albertyao, hongyufang_ee, doroslov, guruv}@gwu.edu

***Abstract*—Recent studies have shown how adversaries can exploit hardware cache structures to launch information leakage-based attacks. Among these attacks, timing channels are especially worrisome since adversaries communicate simply by modulating the timing of shared resource accesses, and do not leave any physical trace of the communication. Therefore, guarding the system against such attacks is critical. Unfortunately, most existing mitigation mechanisms either require non-trivial hardware modifications and/or incur high runtime overheads.**

In this paper, we propose *COTSknight*, a new framework that guards the system against several classes of cache timing channel attacks by *making novel use* of Commercial Off-The-Shelf (COTS) architectural support for cache resource monitoring and prioritization. We find that the adversary's attempt to modulate cache access latency during attacks can be captured using cache occupancy patterns. COTSknight leverages efficient signal processing techniques on cache occupancy patterns to determine the potential for timing channel attacks. Once suspicious domains are identified, COTSknight disbands timing channels using dynamic cache partitioning schemes in hardware. We implement a prototype of our COTSknight framework on an Intel Xeon v4 server and evaluate its efficacy extensively using different spatial encoding schemes, as well as serial and parallel implementations of Last Level Cache (LLC) timing channels. Our results show that COTSknight can successfully thwart several classes of timing channel attacks by allocating disjoint LLC ways to malicious processes. Even in benign cache-intensive workloads, we observe a 6% cache partition trigger rate that results in a relatively small 5% worst-case performance degradation. Interestingly, for some benign applications, upon COTSknight's cache partition, we observe an improved performance by up to 9.2% through eliminating cache interference.

I. Introduction

Among the many forms of sensitive information leakage, timing channels are particularly notorious for leaking secrets leaving no physical evidence of a secret communication having taken place [7]. In other words, the spy receiving secrets in these timing channels rely on observing modulations of resource access latencies, and do not explicitly receive any bits. Such timing channels can manifest as either side channels (where a benign victim unknowingly leaks sensitive data to a malicious spy), or as covert channels (where a malicious insider trojan process intentionally colludes with a spy process to manipulate the access timing of a shared resource, and reveals secrets illegitimately). Recently, Spectre and Meltdown attacks [42] have shown that timing channels can easily manifest on mainstream hardware, and stress the need for hardware-based information security to be considered as a first order design constraint in computer architecture.

A number of prior works have studied how to manipulate or leverage cache access timing for information leakage channels [28], [29], [26], [38]. Specifically, caches offer a rich medium for implementing such timing channels due to two major reasons: 1. cache is one of the most commonly shared hardware resources, and hence offers large attack surface for adversaries to exploit; 2. caches access latencies for a cache hit and miss are noticeably different, and as such, these can be manipulated such that a spy can infer secrets relatively easily during timing channel attacks [38]. Recent studies have shown successful high-speed cache-based timing channel attacks on both native and virtualized environments [26], [27].

Existing approaches that detect or defend against cache timing channel attacks fall into one of the two categories: 1. Secure cache designs that prevent the adversaries from manipulating cache latencies [25], [34], 2. Pre-emptive cache locking and/or aggressive cache partitioning to isolate processes among mutually distrusting processes [24]. While pre-emptive cache partitioning methods may cause unnecessary performance degradation on all of the applications running in the system, custom cache designs often result in high implementation cost and hardware complexity. Furthermore, custom caches suffer from lack of flexibility against evolving adversaries. Therefore, we need a more practical (ready to use) and an adaptive solution that minimizes hardware complexity while actively defending against adversaries in a robust manner. Toward this end, recent works have shown the promise of leveraging COTS hardware as powerful and cost-effective security knobs [14].

In this paper, we propose *COTSknight*, a novel framework to defend against cache timing channel attacks. COTSknight repurposes COTS hardware to annul timing channels, and does not require modifications to the existing hardware. We observe that cache block replacements by adversaries during their timing channel attacks create distinctly observable patterns on their cache occupancy profiles, which is a stronger indicator than other statistics such as cache misses. Fortunately, recent Intel Xeon v3 and newer processors have incorporated two important performance tuning features, namely, the Cache Monitoring Technology (CMT), that allows for fine-grained

†Fan Yao is currently with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL. This work is done while he was a PhD student at the George Washington University.

monitoring of cache occupancy on individual domains, and the Cache Allocation Technology (CAT), that enables dynamic cache way partitioning for applications [10]. COTSknight makes *smart use* of these two hardware features to boost system security against timing channels. We note that currently, CMT and CAT technologies are made available only in Last Level Cache (LLC). Therefore, we demonstrate our COTSknight on LLCs in this paper. Without loss of generality, we note that COTSknight is applicable to any cache with COTS hardware support for cache monitoring and partitioning.

COTSknight comprises three main components: 1. Cache occupancy monitor, that creates traces of cache occupancy patterns among mutually distrusting domains, 2. Occupancy pattern analyzer, that identifies suspicious domain pairs involved in timing channel-based communication, and 3. Cache way allocation manager, that dynamically partitions cache ways among suspicious application domains to prevent information leakage. We demonstrate the usefulness of our framework on LLCs that currently support cache monitoring and allocation technologies.

The *key* benefits of COTSknight are two-fold: 1. COTSknight prevents information leakage channels in the system by smartly leveraging *off-the-shelf hardware support*. 2. COTSknight effectively identifies suspicious domains *without* indiscriminately separating all domains in a pre-emptive manner.

In summary, the specific contributions of our work are:

1) We demonstrate a practical way to defend caches against timing channel attacks by making *novel use* of COTS performance monitoring and boosting features (namely, CMT+CAT) available in recent commercial processors.

2) We design COTSknight, a novel framework that offers a holistic infrastructure for both detecting and defending against cache-based timing channels. COTSknight monitors cache occupancy traces and quantitatively determines the strength of timing channels before deploying defenses.

3) We implement COTSknight prototype on an Intel Xeon v4 server and evaluate its efficacy extensively using timing channel variants with different spatial encoding schemes, as well as serial and parallel implementations. Our results show that COTSknight can successfully thwart various classes of timing channel attacks.

II. Background

A. Cache Timing Channel Attacks

Cache timing channels typically involve two processes: trojan and spy in the case of covert channels; victim and spy for side channels. Since direct communication between these pairs is explicitly prohibited by the underlying system security policy, the spy process turns to infer secrets by observing the modulated latencies during cache accesses [32].

B. Timing Channel Protocols

Cache timing channel protocols can be categorized along two dimensions: time and space. In the time dimension, 1. *serial protocols* operate by time-interleaving the cache accesses by the victim/trojan and spy in a round-robin fashion (note that such serial protocols are more conducive to covert channels where trojan can explicitly control synchronization [36]); 2. *parallel protocols* do not enforce any strict ordering of cache accesses between the victim/trojan and spy, and let the spy decode the bits in parallel (observed more commonly in side channels [26], [16]). The spy takes multiple measurements to eliminate bit errors due to concurrent accesses.

In space dimension, the attacks can be classified based on the encoding scheme used to communicate secrets [22]. 1. *On-off encoding* works by manipulating the cache access latencies of a single group of cache sets; 2. *pulse position* encoding uses multiple groups of cache sets.

C. Cache Occupancy Monitoring and Way Allocation

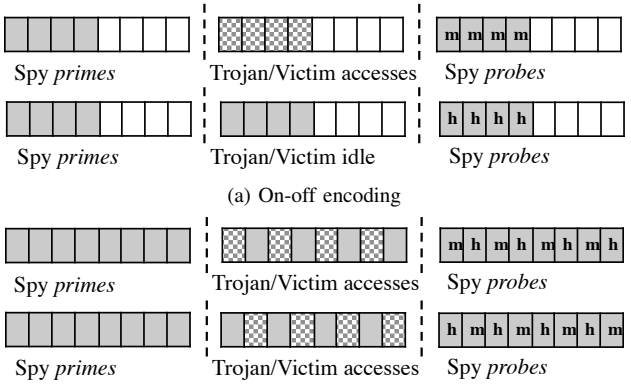
The CMT allows for uniquely identifying each logical core, i.e., hardware thread with a specific Resource Monitoring ID (RMID) [10]. Each unique RMID can be used to track the corresponding LLC usage by periodically reading from its Model Specific Register (MSR). It is possible for multiple threads to share the same RMID allowing for their LLC usage to be tracked together. Such a capability enables flexible monitoring at user-desired *domain* granularity such as a core, a multi-threaded application or a virtual machine.

Additionally, the CAT technology enables an agile way for partitioning the LLC ways. With CAT, caches can be configured to have several different partitions on cache ways, called Classes of Service (CLOS) [2], [10]. A hardware context, that is restricted to certain ways, can still *read* the data from other ways where the data resides, however, it can only *allocate* new cache lines in its designated ways, which means evicting cache lines from other CLOS is not possible. The default for all applications is CLOS0, where all cache ways are accessible. It is worth noting that the current version of CAT supports arbitrary *runtime reconfigurations of CLOSes* transparently, which essentially makes *dynamic response* for cache timing channels possible.

III. Threat Model

Our threat model assumes information exfiltration due to *timing channels*, a class of attacks that rely on timing modulation using a shared resource (LLC in our paper). As noted in Section II-A, we observe that, in order to decipher secrets, the spy relies on the same pattern of interaction with trojan or victim in terms of observing modulated cache latencies created by cache block replacements.

In this paper, we demonstrate a sophisticated form of attacker that does not rely on any prior memory sharing, and launch attacks on caches simply by creating conflict misses (replacement) on cache sets. Flush+Reload, another popular attack strategy, requires the shared memory blocks that is hard to achieve in settings where code or data sharing is prohibited. Furthermore, cache block flushing instructions, such as *clflush*, have been shown to be vulnerable to other types of attacks (including Rowhammer attacks [20]). Hence, some recent system implementations have designed mechanisms to restrict *clflush* usage [41]. In light of this, we do not consider flush+reload



(a) On-off encoding

(b) Pulse-position encoding using odd/even sets

Fig. 1: Cache timing attacks with on-off and pulse-position encoding. '*m*' and '*h*' denote cache miss and hit respectively.

attack. However, for evict+time and evict+reload attacks [15], [28] where cache replacements are used to alter latencies, our solution design would still be applicable (See our design in Section V to track cache occupancy change patterns).

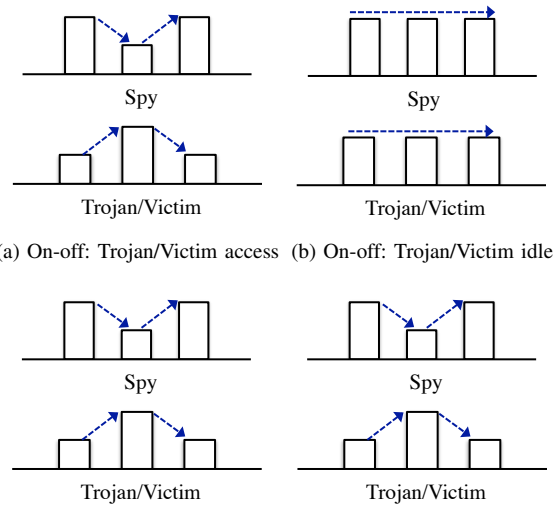
IV. Tracking Cache Occupancy Changes

Cache access latencies are modulated by creating cache conflict misses with their (private) data blocks [36], [28], [26]. There are two possible ways to achieve this by targeting different numbers of cache block groups that encode bits.

On-off encoding [35], [27]: In this class of attacks, the trojan/victim and spy contend on a single group of cache sets (first 4 blocks in Figure 1a). During prime phase, the spy fills cache sets with its own data (gray blocks). The trojan/victim either 1. accesses the same group of cache sets to fill them with its own data (dotted blocks), or 2. remains idle and spy's contents are left intact (gray blocks). The spy probes these cache blocks and measures access latencies. Longer latency values indicate cache conflict misses (marked as *m* in Figure 1a), while shorter latencies indicate cache hits (marked as *h*). Secret bits are deciphered based on cache latencies.

Pulse-position encoding [30], [36], [26]: In Figure 1b, the trojan/victim and spy exploit two distinct groups of cache sets to communicate the bits. Initially, the spy primes both groups of cache sets by filling all of the ways with its own data. The trojan/victim may either replace contents in the first (odd) or second (even) group of cache sets. The spy probes both groups of cache sets, and depending on the group with higher cache access latency, the secret bits are decoded. This encoding scheme can be generalized to multi-bit symbols when multiple groups of cache sets are chosen for communication.

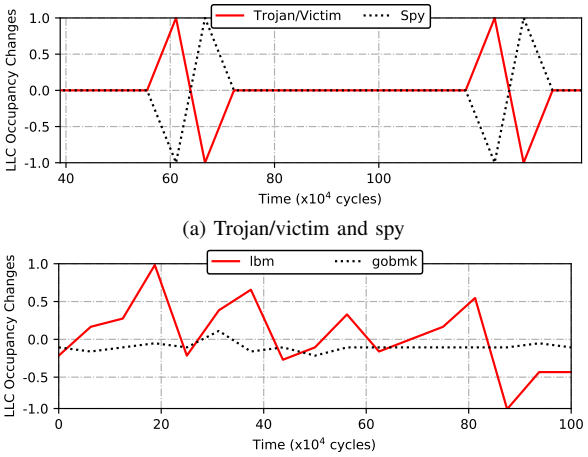
Figure 2 illustrates the changes in LLC occupancy under the two encoding methods. In on-off encoding, when trojan/victim accesses cache, the trojan's cache occupancy should first *increase* (due to trojan/victim fetching its cache blocks) and then *decrease* (during spy's probe phase when trojan/victim-owned blocks are replaced). Similarly, the spy's cache footprint would first *decrease* (due to trojan/victim's



(a) On-off: Trojan/Victim access (b) On-off: Trojan/Victim idle

(c) Pulse-position: Odd sets (d) Pulse-position: Even sets

Fig. 2: LLC occupancy changes for trojan/victim and spy.



(a) Trojan/victim and spy

(b) SPEC2006: lbm and gobmk

Fig. 3: LLC occupancy rate of change for (trojan/victim, spy) and (lbm, gobmk) pairs.

filling in the cache blocks) and then *increase* (when spy probes and fills the cache with its own data). When trojan/victim does not access the cache, neither of the processes change their respective LLC occupancies. Under pulse-position encoding, regardless of trojan/victim's activity, we observe a seesaw (swing) pattern in their LLC occupancies.

To demonstrate our observation, we implement a timing channel with on-off encoding (shown in Figure 1a), and study cache occupancy changes. Figure 3 shows a representative window capturing rate of change in LLC occupancy over time. In Figure 3a, the trojan/victim's cache occupancy gain in proportion to spy's loss and vice versa.

Besides timing channel variants in space dimension, note that this phenomenon exists along time dimension as well. In a parallel protocol, since the spy decodes a single bit with multiple measurements, there will be a cluster of such swing

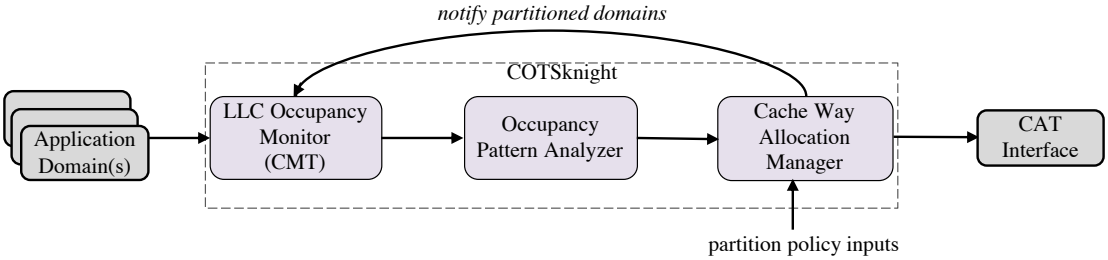


Fig. 4: Overview of COTSknight Design.

patterns during every bit transmission, whereas serial protocols will likely show a single swing pattern.

To contrast with regular applications that have no known timing channels, we also show a representative benign application pair from SPEC2006 benchmarks [17] with relatively high LLC activity, namely *lbm* and *gobmk*. We observe that these application pairs do not usually show any repetitive pulses or *negative* correlation in their occupancy rates. As we can see from Figure 3b, the occupancy patterns are rarely correlated (no obvious swing pattern), e.g., there are time periods when both applications have unaligned negative dips, or one application's LLC occupancy fluctuates while the other remains unchanged, or the two LLC occupancies almost change in the same direction.

Based on the discussion above, we make the following key observation: *Timing channels in caches fundamentally rely on cache block replacements (that influence spy's timing) and create repetitive swing patterns in cache occupancy regardless of the specific timing channel protocols. By analyzing these correlated swing patterns, there is a potential to uncover the communication strength in such attacks*. We note that merely tracking cache misses on an adversary will not be sufficient as an attacker may inflate cache misses (through issuing additional cache loads that create self-conflicts) on purpose in order to evade detection.

V. System Design

In this section, we first discuss CMT-based LLC occupancy monitoring and trace analysis to detect timing channels. We then outline our cache way allocation mechanism that dynamically partitions LLC to prevent potential information leakage. Our design overview is shown in Figure 4.

A. LLC Occupancy Monitor

From the architecture perspective, the finest granularity for LLC occupancy monitor (referred to as *monitor*) is at the level of logical cores that can be readily setup by configuring a per-thread architectural MSR (i.e., `IA32_PQR_ASSOC`) [3]. However, this requires every thread migration between cores to be manually bookmarked. To counter this problem, application-level and Virtual Machine (VM) level monitoring are available that can automatically manage remapping of RMIDs when applications or VM guests swap in or out of logical cores [1], [3]. Also, CMT integrates a query-based model where any core in a processor package can query the LLC occupancy of other cores. COTSknight capitalizes this capability and uses a separate, non-intrusive thread to collect LLC occupancy traces for all of the currently running domains.

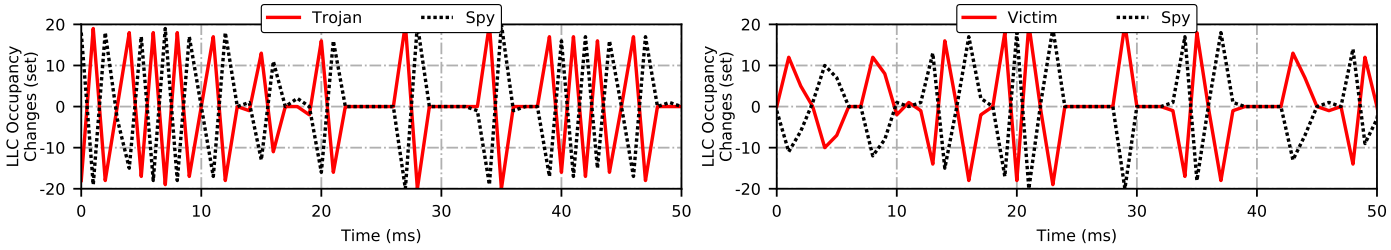
B. Occupancy Pattern Analyzer

Once LLC traces are gathered, the LLC occupancy analyzer (abbreviated as *analyzer*) checks for any potential timing channel activity. Note that the timing channel attacks can happen within a certain period during the span of entire program execution, and hence, we adopt a window-based analysis of LLC occupancy traces. The window size can be chosen by the system administrator based on her needs: swiftness of defense vs. runtime overhead trade-offs.

Assume that we have n windows (indexed by i) of raw LLC occupancy traces for a pair of application domains (D_1, D_2). x_i and y_i ($0 \leq i \leq n-1$) are the LLC occupancy sample vectors obtained by reading LLC occupancy MSRs periodically within the i^{th} window for domains D_1 and D_2, respectively. We can then get the time-differentiated cache occupancy traces for each domain, denoted as $\Delta x_{i,j}$ and $\Delta y_{i,j}$ (i.e., the LLC occupancy difference between two consecutive samples). Figure 5 shows time-differentiated LLC occupancy traces for covert and side channels that implement serial protocol with on-off encoding and parallel protocol with pulse-position encoding respectively.

As the second step, the analyzer focuses on finding mirror images of pulses in the two time-differentiated cache occupancy traces. Recall from Section IV where the spy and trojan/victim communicate by growing their own cache space through taking away the corresponding cache space from each other to create conflict misses that alter cache access timing for the spy. To capture such unique patterns, we take the product of Δx_i and Δy_i as z_i. Additionally, to filter the noise effects from surrounding cache activity, we zero-out all *non-negative* $z_{i,j}$ values that do not correspond to gain-loss swing patterns in LLC occupancy.

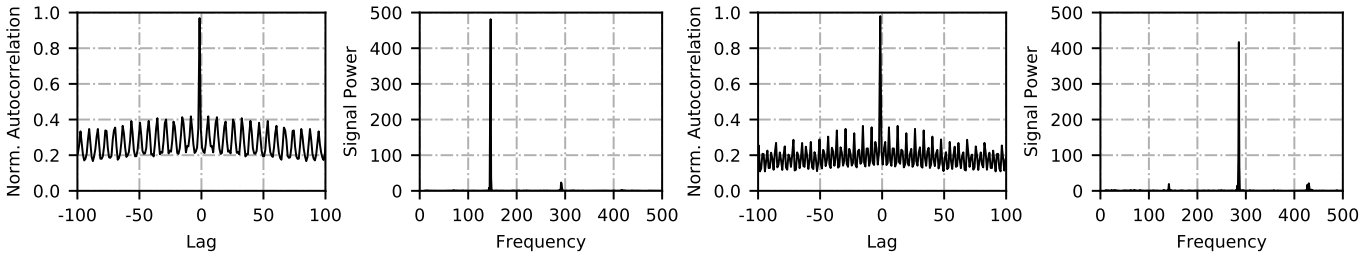
We note that z_i *elegantly characterizes the swing pattern and cancels noise from other background processes*: When cache occupancy of one process changes while the other one remains stationary, the product at that point would be zero. When two processes are both influenced by a third-party process, their cache occupancy might change in the same direction, so that the product of two time-differentiated occupancy trace points would be positive. Negative values



(a) Serial protocol with on-off encoding

(b) Parallel protocol with Pulse-position encoding

Fig. 5: LLC occupancy traces for covert channel (with serial, on-off) and side channel (with parallel, pulse-position).



(a) Serial protocol with on-off encoding (covert channel)

(b) Parallel protocol with pulse-position encoding (side channel)

Fig. 6: LLC trace analysis through computing normalized autocorrelation and power spectrum on covert and side channels.

occur when the cache occupancy patterns of the two processes move in opposite directions due to mutual cache conflicts.

In effect, the series z_i contains information about mutual eviction behavior between the two processes. Our goal now is to check if z series contains repeating patterns that may be caused by intentional eviction over a longer period of time (denoting illegal communication activity). For every window, we compute the normalized autocorrelation function for z_i, denoted as $r'_i(m)$, where m (samples) is the lag of series z_i. According to the Cauchy-Schwarz Inequality [23], if the time-differentiated curves Δx_i and Δy_i are strictly linearly dependent, $r'_i(0)$ would equal to 1. Conversely, the lack of linear dependency between Δx and Δy would be indicated by $r'_i(0)$ being close to 0. Note that benign applications may also exhibit short swing patterns on LLC cache occupancy, but are highly unlikely to repeat over a longer time period. To reduce noise from such short swings, we compute average of all autocorrelation function r'_i over n windows, which we denote using $r'(m)$.

With increase in lag value (m), the eviction pattern would begin to mismatch more heavily. Consequently, normalized autocorrelation at lag m, $r'(m)$, would begin to decrease. When the lag m equals to length of the complete pattern (wavelength, m_w), some of the patterns would rematch and the $r'(m_w)$ would rise back to higher values. Note that there still might exist a small offset in the repetitive pattern, and this may cause $r'(m_w)$ to be not as high as $r'(0)$. *However, $r'(m_w)$ is very likely to be a local maximum in the presence of timing channel activity*. As m increases further, we note that the local maxima caused by rematched patterns would begin to appear repeatedly.

Fourier transform is a powerful tool to extract the repetitive patterns in signals. We compute discrete Fourier transform of the autocorrelation function r' (we assume that there are $p+1$ samples within each window):

$$R(k) = \sum_{m=-p+1}^{p-1} r'(m) W_{2p-1}^{m \cdot k} \tag{1}$$

where $W_{2p-1} = e^{-2\pi i/(2p-1)}$. Here R is the *power spectrum* of z. The presence of a single or equally-spaced multiple spikes with concentrated (very high) signal power outside of frequency 0 in R indicates repetitive pattern in the underlying sequence. Note that this is a typical characteristic of timing channels. Figure 6a illustrates the normalized autocorrelation function of *(trojan, spy)* in covert channels [7]. $r'(0)$ is very close to one, so the two time-differentiated LLC occupancies are linearly dependent. We can also visually observe repeated occurrence of local maxima and a sharp peak around frequency of 150 in the power spectrum, which indicates timing channel activity. Figure 6b shows results for *(victim, spy)* in LLC side channel [26], where a sharp peak in power spectrum around 290 is observed.

C. Cache Way Allocation Manager

After the Cache way allocation manager (*allocator*) receives RMIDs of identified suspicious domains from the analyzer, it will configure LLC ways to fully or partially isolate the suspicious pairs. Note that all of the newly created domains (i.e., newly spawned processes) are initially set to a default CLOS (e.g., CLOS0) with access to all LLC ways.

Consider a newly discovered suspicious pair (D_1, D_2). The allocator can simply create two non-overlapping CLOS1 and

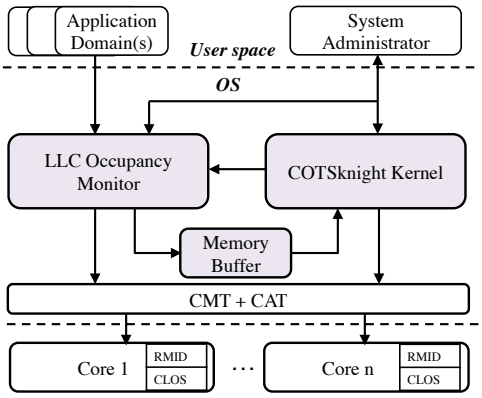


Fig. 7: COTSknight with other system components.

CLOS2 for assignment to D_1 and D_2. COTSknight heuristically assigns ways to each domain based on their ratio of LLC occupancy sizes during the last observation period. To avoid starvation, our policy sets the minimum number of ways for any domain to be at least 4, which works reasonably well (See our experimental results in Section VIII).

The allocator can apply different policies to manage the partitioned domains at runtime. We list two candidate policies: 1. ***Aggressive Policy***, that partitions the two suspicious domains and keeps them separated until one of them finishes execution. This policy guarantees the highest level of security, and removes the need to track already separated domain pairs. 2. ***Jail Policy***, that partitions the two domains for a period of time, and then allows access to all of the LLC partitions after reaching a certain timeout period. This policy provides the flexibility to accommodate benign application pairs that need to be partitioned tentatively.

VI. Implementation

We implement our framework prototype on a real system with Intel Xeon E5-2698 v4 processor. The server runs Centos 7.0 with Linux kernel 4.10.12. COTSknight is deployed as an OS-level service that has two major modules, the LLC Occupancy Monitor and the COTSknight kernel (Figure 7).

LLC Occupancy Monitor. The monitor dynamically traces the LLC occupancy for a *watch-list* of domains. It designates newly created domains (e.g., VMs, applications) with RMIDs, and also performs recycling of RMIDs. By default, all running domains are monitored separately. The occupancy monitor exports interface to the system administrator to override domain configurations. For instance, multiple domains belonging to the same user can be grouped together. The monitor periodically queries the LLC occupancy MSRs at a configurable sampling rate (setup by the system administrator). The occupancy data for all the monitored domains are stored in a secure memory buffer. When the monitor receives notification from the COTSknight kernel about partitioned domains, it removes them from its watch-list.

COTSknight Kernel. This module combines the analyzer and allocator. It periodically empties the memory buffer by reading the LLC occupancy traces for the monitored domains, and performs signal analysis based on our methodology discussed in Section V-B. Once newly suspicious domains are recognized, it generates a new *domain to CLOS mapping* so that these domains will be isolated and potential timing channels can be annulled. It can flexibly manage the partitioned domains based on the partition policy inputs provided by the system administrator (discussed in Section V-C).

Abbr.	Encoding	Timing	Refs.
para-onoff	On-off	Parallel	[35], [27], [4]
serial-onoff	On-off	Serial	[7], [18]
para-pp	Pulse-position	Parallel	[29], [26]
serial-pp	Pulse-position	Serial	[30], [36], [16]

TABLE I: Cache timing attack classes studied in our paper.

VII. Experimental Setup

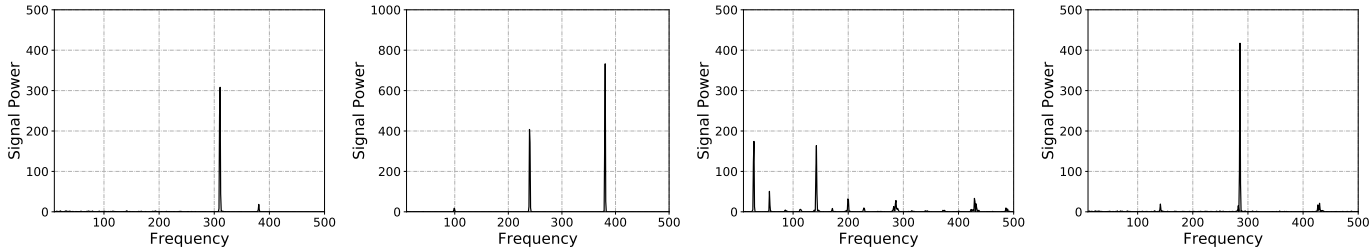
Our experimental testbed is an Intel Xeon V4 with 16 CLOS and 20 LLC slices, and each LLC slice has 20×2048 64-byte blocks. By default, all logical cores are assigned a *RMID0* (the default resource monitoring ID), and the associated CLOS configuration MSR is set to `0xFFFFF`. This means that all domains can use all of LLC ways initially. COTSknight initializes a memory buffer to accumulate LLC MSR readings sampled at 1,000 per second (maximum stable rate supported by the current hardware).

A. Cache Timing Channel Attacks

Table I shows the four attack variants (discussed in Section II) along with the recent studies that have demonstrated or characterized such attacks. We obtained source code for timing channels from authors of prior works when possible, and implemented the rest on our own based on their description. We adapted techniques from Liu et. al. [26] to dynamically generate address sets that create conflict misses. Every such set contains at least S entries that map to the same cache set (S is the LLC associativity). Each variant is configured to perform the prime+probe attack using a specific number of cache sets (32 to 128). For *serial-onoff* and *para-onoff*, all target cache sets are treated as one group, and for *serial-pp* and *para-pp*, we generate two equally-sized groups of cache sets in our experiments.

B. Benign Applications

To evaluate COTSknight on benign workloads, we utilize the SPEC2006 benchmarks [17] that exhibit a variety of cache and memory access characteristics. We run combinations of SPEC2006 benchmarks with *reference inputs* that exhibit various level of cache intensiveness. With the prototype implementation of COTSknight on real system hardware, we are able to run each of the workloads from beginning to end that can fully stress the capability of our proposed framework.



(a) serial-onoff (b) serial-pp (c) para-onoff (d) para-pp

Fig. 8: Power Spectrum in attack variants including trojan/victim-spy pairs.

VIII. Evaluation

A. Analysis on Cache Timing Channels

We setup each attack variant (shown in Table I) to run for 90 seconds on Intel Xeon v4 server. To emulate real system environment, we co-schedule two other SPEC2006 benchmarks alongside the trojan and spy. We run each attack variant multiple times with different co-scheduled process pairs and numbers of target sets. The analyzer performs pair-wise normalized autocorrelation on time-differentiated LLC occupancy traces for 6 combination pairs of processes. In all cases, the trojan/victim-spy processes can be identified as these pairs consistently had the highest power in the frequency domain. In fact, our experiments show that the attacker pair's peak power spectrum values are at least an order of magnitude higher than that of the benign application pairs.

Figure 8 shows the analyzer's results on representative windows for trojan/victim-spy pairs. In the *serial-onoff* attack, we observe a single concentrated, sharp peak with the power value at around 300 in the frequency domain, while the other data points are almost all zeros. This indicates the existence of a dominating signal in the time domain corresponding to the repetitive gain-loss occupancy pulses due to timing channel activity (Figure 8a). The value in the horizontal axis for the peak captures the cache operation frequency of the two involved processes. We also observe a similar isolated peak for the trojan/victim-spy pair in *para-pp*, as shown in Figure 8d where the signal power is even higher compared to serial-onoff case. We note that this is because both '0' and '1' bit communications cause gain-loss pulses compared to serial onoff encoding, making the repetitive swinging patterns in LLC occupancy between trojan/victim-spy even stronger.

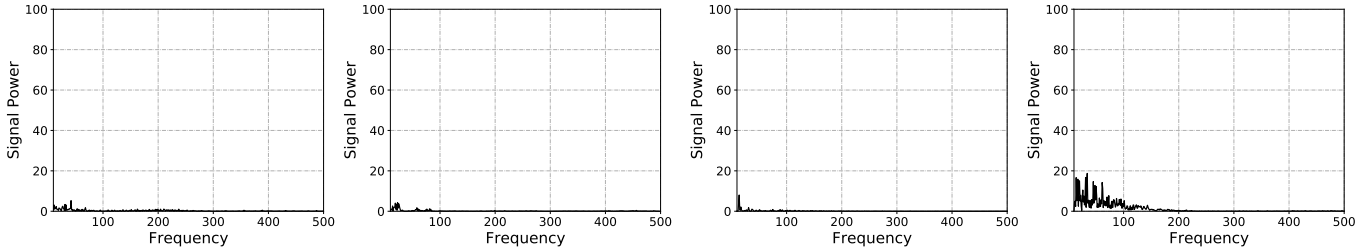
Interestingly, in some of the attack variants, there exist two sharp peaks (Figures 8b and 8c). This can be explained as follows: In some cache timing channels, there are usually two repetitive sets of behaviors at different frequency levels- 1. prime+probe operations by the spy, and 2. cache accesses by the trojan/victim. For example, in *serial-pp*, the spy performs cache evictions during prime+probe periodically and the trojan/victim activity can create variations in eviction patterns. This creates two different frequencies that are observed as two separate peaks in the power spectrum (Figure 8b). Similarly, in *para-onoff* attack, for every trojan/victim operation, the spy performs repeated multiple probes and during each probe, it causes repetitive cache set evictions. These two aspects are represented as periodic signals with two frequencies in the power spectrum (Figure 8c).

B. Analysis on Benign Workloads

To generate benign workloads, we first classify SPEC2006 benchmarks into two groups: 1. *H-Group*, that has cache-sensitive applications with high rate of Misses per Kilo Instructions (MPKI) and LLC accesses (including *GemsFDTD*, *leslie3d*, *mcf*, *lbm*, *milc*, *soplex*, *bwaves*, *omnetpp*, *bzip2*); and 2. *L-Group* that contains rest of the applications with relatively low cache-sensitivity [19]. We generate workloads with three levels of cache sensitivity from these two groups: (i) *highly cache-intensive workloads (**hh-wd**)* where all four applications are assembled from within *H-Group*; (ii) *medium cache-intensive workloads (**hl-wd**)* with two applications randomly selected from *H-Group* and the other two from *L-Group*; (iii) *low cache-intensive workloads (**ll-wd**)* where all four applications are chosen from *L-Group*.

We run 60 benign multi-program workloads (20 in each sensitivity level) where each application is an individual domain. Figure 9 illustrates the analyzer's results on representative windows for benign workloads. Our results show that a vast majority of domain pairs (79%) in benign workloads have very low normalized autocorrelation (0-lag) for the time-differentiated LLC occupancy traces. Obviously, their corresponding power spectrums show no observable peaks in these cases as observed in Figures 9a, 9b and 9c. Figure 9d shows an interesting hh-wd workload where there is a high normalized autocorrelation (0-lag) and a number of small peaks in the frequency domain, (*GemsFDTD* and *mcf*). However, note that the peaks are simply numerous (unlike timing channels) and their relative signal strengths are weak (< 20). We found that the high autocorrelation (0-lag) results from a series of swing pulses due to cache interference between GemsFDTD and mcf, and the cache timing modulation is simply too chaotic (at many different frequencies) for any real communication.

Figure 10 shows the cumulative distribution function of peak signal power among the benign workloads in thousands of analysis window samples (2.5 sec) during execution of workloads. We observe the peak signal power to be less than 5 about 80% of the time, and higher than 50 for only about 2% of the time. This shows that a vast majority of benign workload samples do not exhibit high peak signal power, and



(a) (cal hmm) gob lib—*ll-wd* (b) (Gem hmm) xal bwa—*hl-wd* (c) (lbm mil) sop Gem—*hh-wd* (d) (Gem mcf) bzi bwa—*hh-wd*

Fig. 9: Power spectrum for representative subset of benign workloads. Domain pairs with highest autocorrelation (0-lag) are shown within parentheses.

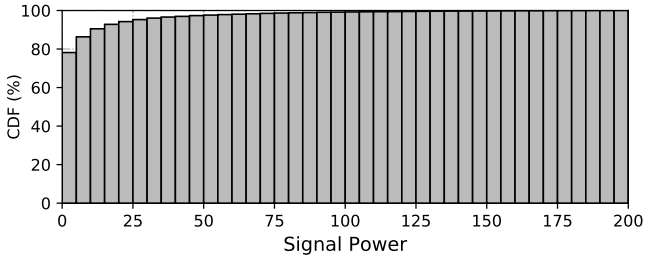


Fig. 10: Peak signal power CDF for benign workloads.

their power is significantly less than in the case of any known timing channels (which usually have signal strength at well above 100).

C. Effectiveness of COTSknight

We evaluate COTSknight's effectiveness on two aspects: 1. ability to counter cache timing channels, and 2. partition trigger rate and performance impact on benign workloads. To minimize the victim's performance impact, we note that migrating the spy to a different server may be also considered as an alternative mitigation strategy in side channels.

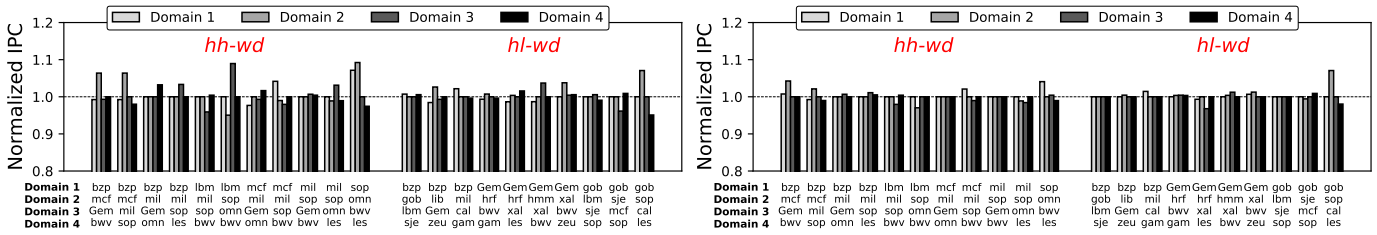
Defeating LLC timing channels. As noted in Section VIII-A, we run multiple instances of cache timing channel attack variants with different background processes, and also vary the number of target cache sets. We observe that the power peaks are well above 100 most of the time in all of our timing channels. There are a few windows during the attack setup phase where the peak values drop slightly below 100. It is worth mentioning that, as a runtime protection scheme, COTSknight's dynamic way allocation mechanism has considerably less impact even on false alarms, where we trigger cache partitioning instead of terminating processes prematurely. We choose a rather conservative signal power threshold of 50 to trigger LLC partitioning. COTSknight identifies all of the trojan-spy domain pairs within five consecutive analysis windows after they start execution.

Paritition trigger rate for benign workloads and the corresponding performance impact. On benign workloads in *ll-wd* category, we observe that LLC partitioning was never triggered during their entire execution. Among all workloads with low to high cache intensiveness, only 6% of the domain pair population had LLC partitioning - these benchmarks covered 2% of the analysis window samples. Figure 11 shows the performance impact represented as normalized IPC for the workloads that *were actually LLC-partitioned at runtime* (we note that partitioning didn't impact other benign workloads). LLC partitioning minimally impacts most of the applications (less than 5% slowdown), and interestingly, we observe performance boost for many of them (up to **9.2**% performance speedup). The overall impact on all the applications that ran with partitioned LLC was positive (about **1**% speedup averaged among all workloads that trigger way allocation). In other words, the dynamic way allocation scheme can potentially help the performance for benign workloads while it protects the system. This happens because even benign applications can suffer from significant cache contention and LLC partitioning can be beneficial as it alleviates the interference (e.g., soplex and omnetpp). From our results, we see that Aggressive Policy (that fully partitions suspicious pairs) shows higher variations in both performance gains and losses, while the conservative Jail policy (that partitions tentatively for 30 seconds until timeout) incurs better worse-case performance penalties.

Runtime Overhead. COTSknight implements the non-intrusive LLC occupancy monitoring for *only* mutually distrusting domains identified by the system administrator. The time lag to perform the autocorrelation and power spectrum analysis for the domain pairs is 25 ms, which means that COTSknight offers rapid response to cache timing attacks. COTSknight incurs less than 4% CPU utilization with 4 active mutually-distrusting domains. Note that the runtime overhead of COTSknight does not necessarily scale quadratically with the number of system domains, since not all of them would have active LLC traces in each analysis window and only mutually-distrusting domains would need to be analyzed.

IX. Discussion

COTSknight offers a new framework that builds on COTS hardware and can be augmented with a host of signal processing techniques to eliminate noise, randomness or distortion to unveil the timing channel activity. We already incorporated filtering patterns that are non-negatively correlated and window-based averaging techniques to eliminate short swings for benign applications (See Section V-A).



(a) Aggressive Policy

(b) Jail Policy

Fig. 11: Performance impact on benign workloads where COTSknight allocator triggers LLC partition. Instructions/Cycle (IPC) is normalized to the baseline when no LLC partitioning is performed.

We note that COTSknight framework can be easily extended with additional mechanisms to deal with even more advanced attacks. For instance, one may think of a trojan-spy pair that pseudo-randomizes the intervals between two consecutive bits to obscure their communication pattern. To be clear, in practice, cache timing channels with randomized bit intervals are very hard to synchronize at these random times in a real system environment amidst noise stemming from hardware, OS and external processes. Even if such attacks were feasible, COTSknight can be adapted to recognize them through a signal pre-processing procedure called *time warping* [11], that removes irrelevant segments from the occupancy traces.

In the current implementation, COTSknight leverages CAT to dynamically partition cache ways among suspicious processes. Our results in Section VIII-C have shown that the partitioning rates and the resulting performance impact are low. However, for long-running applications, restricting the cache ways over time may not be desirable, and migrating the spy processes to other processors can be employed as an alternative option, especially for victims in side channels. Note that migration of processes and virtual machines are widely studied and well supported in existing systems.

X. Related Work

Prior works have demonstrated covert and side channels exploiting caches [26], [38]. The *Prime+Probe* technique is most commonly exploited as it requires the least level of system privileges and requirements (e.g., no need to have shared memory). We have shown that COTSknight can effectively mitigate these variants of attacks on such caches.

A number of works have been proposed to defend system against cache timing channel attacks [25], [34], [7], [24], [13], [12], [39]. Wang et al. [34] propose a new cache design that randomizes the mapping of data blocks to cache lines in the L1 cache. CC-Hunter [7], [33] detects covert timing channel in caches by capturing fined-grained cache conflict miss patterns between two processes. Yao et al. [40] propose a statistical method to quantify the presence of cache timing channels in NUMA-based architectures. Fang et al. [12] have studies techniques that leverage hardware prefetchers to stop cache timing channels. ReplayConfusion [37] records program's cache accesses, replays them using a different mapping from addresses to caches and observes differences in cache miss patterns. Unlike COTSknight that works with existing hardware, most of these prior mechanisms require hardware modifications. More importantly, COTSknight provides more systematic protections as compared to pure detection approaches due to the judicious integration of runtime detection and performance-friendly countermeasures.

Bazm el al. [6] have employed cache occupancy statistics to detect anomalous behavior in conjunction with other performance counters such as cache misses. However, their proposed technique scans for unusual access behavior based on LLC footprint, which can be subject to high false positive alarms. Differently, COTSknight's defense mechanism is based on detecting pair-wise gain and loss patterns in cache occupancy that is shown to be the unique characteristic for parties involving timing channel activity, and our experimental results show that COTSknight is both effective and efficient.

CATalyst [24] leverages the Cache Allocation Technology (CAT) to reserve static cache partitions where secure pages are pinned upon request from the application. In contrast, COTSknight can successfully defeat these attacks without application or user-level inputs. Aga et al. [5] have demonstrated that CAT can be used to enhance rowhammer attack due to the fact that smaller number of LLC ways can help increase the speed of cache miss generation. DAWG [21] has proposed secure cache partitioning by strictly isolating both cache hits and misses between application domains. We note that COTSknight can use alternate mitigation strategies such as job migration instead of CAT as well. Also, such works highlight the potential of developing robust and adaptive COTS-based solutions to evolving adversaries. Such techniques can be combined with memory protection [31] to improve overall memory safety, including emerging memory types [8], [9].

XI. Conclusion

In this paper, we proposed *COTSknight*, a novel framework to protect caches against timing channel attacks through smartly leveraging COTS support for cache monitoring and performance tuning. COTSknight leverages existing hardware, and utilizes robust signal processing techniques to identify the presence of cache timing channels. We implemented COTSknight prototype on Intel Xeon v4 server and evaluated its efficacy extensively using different spatial encoding schemes, as well as serial and parallel implementations of LLC

timing channels. Our results showed that COTSknight can successfully thwart several classes of timing channel attacks through dynamic LLC partitioning. COTSknight introduces less than 5% performance slowdown in the worst case for benign applications. In conclusion, our work highlights the promise of COTS hardware in providing flexible support for robust and adaptive security against cache timing channels.

ACKNOWLEDGMENT

This paper is based on work supported by the US National Science Foundation under grant CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684.

REFERENCES

[1] Intel Cache Monitoring: Present and Future, 2016. https://xenbits.xen.org/people/dariof/docs/public/designs/CMT-in-scheduling.pdf.

[2] Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family, 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[3] Intel-CMT-CAT Pacakage, 2017. http://https://github.com/01org/intel-cmt-cat.

[4] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Topics in Cryptology–CT-RSA*. Springer, 2008.

[5] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks. In *International Symposium on Hardware Oriented Security and Trust*. IEEE, 2017.

[6] M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J. Menaud. Cache-based side-channel attacks detection through Intel Cache Monitoring Technology and Hardware Performance Counters. In *International Conference on Fog and Mobile Edge Computing*, 2018.

[7] Jie Chen and Guru Venkataramani. CC-hunter: Uncovering covert timing channels on shared processor hardware. In *International Symposium on Microarchitecture*, pages 216–228. IEEE, 2014.

[8] Jie Chen, Guru Venkataramani, and H Howie Huang. RePRAM: Recycling PRAM faulty blocks for extended lifetime. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[9] Jie Chen, Guru Venkataramani, and H Howie Huang. Exploring dynamic redundancy to resuscitate faulty PCM blocks. *ACM Journal on Emerging Technologies in Computing Systems*, 10(4):31, 2014.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3B*, 2016.

[11] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid. Warp: Time warping for periodicity detection. In *International Conference on Data Mining*. IEEE, 2005.

[12] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *IEEE International Symposium on Hardware Oriented Security and Trust*. IEEE, 2018.

[13] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *Intl Journal of Parallel Programming*, pages 1–24, 2018.

[14] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding control flows using intel processor trace. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Usenix Security*, volume 15, pages 897–912, 2015.

[16] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[17] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[18] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *International Symposium on High Performance Computer Architecture*. IEEE, 2015.

[19] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *International Symposium on Computer Architecuture*, pages 361–372. IEEE, 2014.

[21] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *International Symposium on Microarchitecture*. IEEE, 2018.

[22] B.P. Lathi and Z. Ding. *Modern Digital and Analog Communication Systems*. Oxford University Press 4rd ed., 2009.

[23] Alberto Leon-Garcia and Alberto Leon-Garcia. *Probability, statistics, and random processes for electrical engineering*. Pearson/Prentice Hall 3rd ed., 2008.

[24] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *International Symposium on High Performance Computer Architecture*. IEEE, 2016.

[25] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *International Symposium on Microarchitecture*. IEEE, 2014.

[26] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*. IEEE, 2015.

[27] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *The Network and Distributed System Security Symposium*, 2017.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[29] Colin Percival. Cache missing for fun and profit, 2005.

[30] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *International Conference on Computer and Communications Security*. ACM, 2009.

[31] Jianli Shen, Guru Venkataramani, and Milos Prvulovic. Tradeoffs in fine-grained heap memory protection. In *1st workshop on Architectural and system support for improving software dependability*. ACM, 2006.

[32] US Department of Defense. Trusted computer system evaluation criteria. *Department of Defense Standards*, 1983.

[33] Guru Venkataramani, Jie Chen, and Milos Doroslovacki. Detecting hardware covert timing channels. *IEEE Micro*, 36(5):17–27, 2016.

[34] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[35] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security Symposium*, 2012.

[36] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Cloud Computing Security Workshop*. ACM, 2011.

[37] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *International Symposium on Microarchitecture*. IEEE, 2016.

[38] Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Are Coherence Protocol States Vulnerable to Information Leakage? In *International Symposium on High Performance Computer Architecture*. IEEE, 2018.

[39] Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Covert timing channels exploiting cache coherence hardware: Characterization and defense. *Intl Journal of Parallel Programming*, pages 1–26, 2018.

[40] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Great Lakes Symposium on VLSI*. ACM, 2017.

[41] Google Project Zero. Exploiting the DRAM rowhammer bug to gain kernel privileges , 2015. https://goo.gl/FGcCFh.

[42] Google Project Zero. Reading privileged memory with a side-channel, 2018. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html?m=1.