Noname manuscript No.
(will be inserted by the editor)

PrODACT: Prefetch-Obfuscator to Defend Against Cache Timing Channels

Hongyu Fang · Sai Santosh Dayapule · Fan Yao · Miloš Doroslovački · Guru Venkataramani

the date of receipt and acceptance should be inserted later

Abstract Cache timing channels operate stealthily through modulating the cache access latencies, and exfiltrate sensitive information to malicious adversaries. Among several forms of such timing channels, covert channels are especially dangerous since they involve two colluding processes (namely, the trojan and spy), and are often difficult to stop or prevent. In this article, we propose and demonstrate PrODACT, a low-cost mitigation mechanism using hardware prefetchers to defend against cache-based timing channels. Our detection mechanism first identifies the target cache sets that are being exploited by the adversaries, and then the counterattack mechanism fetches cache blocks to obliterate the pattern of cache accesses (misses and hits) created to construct timing channel between the trojan and the spy. We evaluate PrODACT on different classes of cache timing channel protocols that use different numbers of cache block groups for covert communication in a round-robin or parallel fashion. We observe that the cache timing channels suffer an average 50% bit error rate (with a minimum of at least 30%) which makes it very difficult or impossible for spy to decipher any useful information.

Hongyu Fang
The George Washington University
E-mail: hongyufang_ee@gwu.edu

Sai Santosh Dayapule
The George Washington University
E-mail: saisantoshd@gwu.edu

Fan Yao
The George Washington University
E-mail: albertyao@email.gwu.edu

Milos Doroslovacki
The George Washington University
E-mail: doroslov@email.gwu.edu

Guru Venkataramani
The George Washington University
E-mail: guruv@gwu.edu

Keywords Covert timing channel · Hardware Prefetcher · Information Leakage · Cache attacks · Hardware Security

1 Introduction

Computer users have increasingly turned to shared computing platforms, such as cloud services, to satisfy their processing needs. Such platforms allow application processes from different clients to be co-located on a same physical machine. Malicious clients could exploit such features for information leakage purposes. Hardware vendors have already become aware of the danger, and have started providing primitives to prevent the information leakage. For example, Intel's Software Guard Extensions (SGX) [2] supports enclaves in memory and prohibits processes from accessing memory belonging to other enclaves. While such protection schemes exist, a vast majority of hardware resources (including caches) can still be exposed to untrusted processes since they are still shared by processes running in the system.

Among various forms of information leakage attacks, timing channels are especially notorious for their stealthy exfiltration of sensitive information leaving no physical evidence for forensics. These hardware-based timing channels work by observing the modulation of shared resource access timing such that sensitive data can be secretly transmitted [57,1,56,19]. Since hardware resources are inherently shared among processes to maximize utilization of hardware and improve ease of data sharing between genuine users, it is difficult to prevent any timing channels simply through isolating the resources. Such timing channels can manifest hardware either as *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 access timing of a shared resource to exfiltrate secrets [11]. Note that the system security policy explicitly prohibits any form of direct communication between trojan-spy pairs, and as such any form of communication (including indirect means) shall be deemed illegitimate communication between them [17].

Caches are among the *most exploited hardware structures for timing channel attack*s as they are frequently shared between multiple CPU cores. Unlike functional units whose usage can be monitored, caches contain numerous cache sets that may be accessed by several processes concurrently at any given times. This makes guarding of caches against timing channels challenging and important. Cache-based covert timing channel operates using a trojan that intentionally manipulates the latencies of cache data accesses such that the spy can decipher the secrets based on the observed latency [50,41,52,53,33,25]. Most existing solutions aim to redesign caches [51] to prevent timing channels altogether. These methods incur higher costs caused by huge amount of hardware modification and disrupt locality.

In this article, we propose PrODACT, an efficient, low-cost approach to defend against cache timing channels using hardware prefetchers. Our solu-

tion analyzes the cache for suspicious cache access activity by using a low-cost trigger pattern detector that tracks potential cache timing channels. Our defense framework targets the suspicious cache sets identified as being exploited by the timing channel, and then obfuscates any trojan-initiated timing modulation (corresponding to covert bit transmission). This is accomplished through prefetching cache blocks that counter any cache replacement (and non-replacement/hits) as performed by the trojan on suspected cache sets. We experiment and analyze the efficacy of PrODACT against various timing channel protocol implementations. By designing a front-end detector, we make sure that the benign processes do not suffer any performance impact, and that the cache sets belonging to them aren't targeted by the hardware prefetcher unnecessarily. A recent proposal, Disruptive Prefetching [20] has used prefetchers to interrupt side channels. However, it has three major drawbacks:

1. Disruptive prefetcher *randomly* fetches a number of additional blocks (besides the victim's memory blocks) to essentially pollute the cache and thereby, avoid any potential for side channels. We note that such an approach may lead to unnecessary cache pollution, and negatively impact the performance of genuine applications. In contrast, our framework mounts a targeted defense by only disturbing cache sets utilized by the trojan/spy without causing any additional cache pollution.

2. Disruptive prefetching is designed to only defend L1 caches. We note that covert timing channels can also be implemented in non-private, shared caches [33]. We aim to prevent timing channels on any shared cache and disrupt covert communication that exploit them.

3. Disruptive prefetching uses set balancing to create uniform traffic to cache sets. While this is useful to guard against side channels by masking traffic flow between caches and memory, we note that covert channels intentionally construct specific groups of cache sets and repeatedly use them to communicate bits. Our framework is aimed at targeting the specific cache sets (instead of masking traffic patterns) that ultimately helps obfuscate the bit regardless of the number of cache blocks used in communication.

Prefetch-guard [23] introduces the concept of potentially using hardware prefetchers to defend against cache timing channels, and discusses a preliminary strategy to prefetch cache blocks that were primed by the spy after the trojan replaces them. However, Prefetch-guard does not discuss any optimizations to improve system performance in order to realistically use prefetchers as a defense against cache timing channels. Also, Prefetch-guard does not study effectiveness against several variants of cache timing channels.

In this article, we add several major contributions, and demonstrate a number of practical design considerations, as well as defenses against real-world timing channel implementations: Specifically, the new key contributions in this article include:

1. We show how aggressive hardware prefetching can be avoided with a practical trigger pattern recognizer that implements first level filtering to avoid analyzing benign processes that do not show timing channel activity,

2. In order to evaluate the effectiveness of our solution, we study several variants of cache timing channel implementations such as round-robin and parallel, as well as protocols that exploit single and multiple groups of cache sets.
3. We also discuss how our framework can be used to defend against prominent classes of timing channels, namely Prime+Probe, and Flush+Reload.

PrODACT provides two key advantages over prior solutions:

- *Scalability*: Our framework targets misbehaving trojan-spy processes. Therefore, regardless of the total number of processes running in the system, as long as suspicious pairs are identified (through observing repeated, intentional cache block replacements), the timing channel activity can be annulled between these malicious pairs without adversely affecting benign applications that utilize the remaining cache sets. We note that cache partitioning-based defenses [49,38] are hard to scale because the number of cache partitions are limited. If the cache is partitioned heavily, benign processes may slow down because of insufficient cache capacity.
- *Low cost*: We leverage existing hardware prefetchers, and make minimal hardware modifications to track cache conflict misses unlike prior approaches that fundamentally alter cache designs by adding secure and non-secure partitions [51] or obfuscating data placement [32].

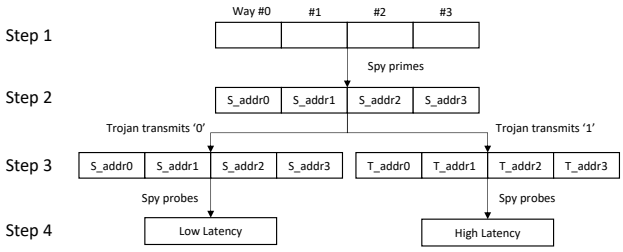
The major contributions of our article are as follows:

1. We propose PrODACT, an efficient approach to counter cache-based timing channels. We demonstrate novel ways to utilize hardware prefetchers and obfuscate the trojan-spy communication in a targeted manner such that the spy will *not* be able to correctly decipher the bits transmitted by the trojan.
2. We design a two-level, scalable low-cost detector for cache timing channels and suspicious target cache sets. We show obfuscation methods that perturb the cache access timing modulation orchestrated by the trojan by increasing the error rate for the spy.
3. We show the efficacy of our approach through evaluating on various classes of timing channel protocol implementations, namely round-robin and parallel with different numbers of cache set groups used for covert communication. Our experimental results show that hardware prefetchers can be highly effective in defending against cache timing channels.

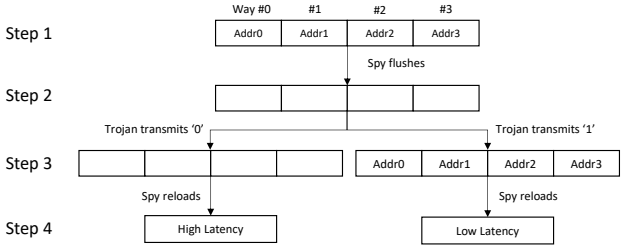
2 Background

2.1 Cache Timing Channels

Cache covert timing channel usually involves two processes: trojan/victim and spy, where the spy learns of sensitive secrets from trojan/victim through timing modulation of cache access latencies. Note that any form of direct communication between them will be prohibited by the underlying system security pol-



(a) Prime+Probe



(b) Flush+Reload

Fig. 1: Steps to Implement Cache Timing Channel Attack

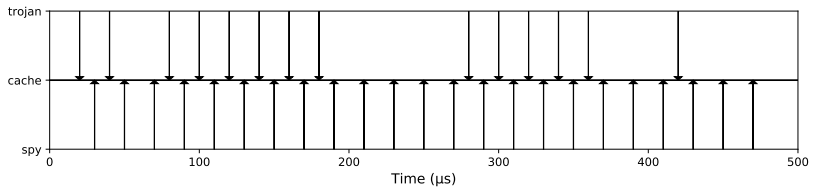
icy [17]. Caches present a good choice for trojans to leak information because of the following two reasons:

1. Caches are shared by different processes (both trojan and spy have access to same cache).

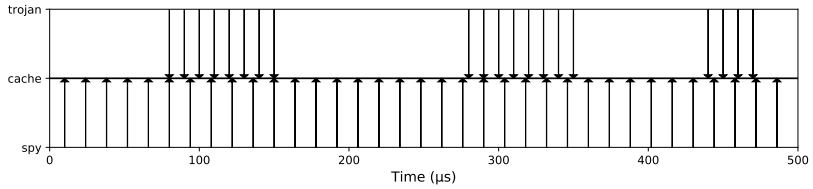
2. Processes can observe the access latency to infer whether a memory address is in cache.

When a process requests a memory line in the cache (cache hit), the time it takes would be much shorter than requesting a memory line which is not in the cache (cache miss). There are mainly two ways for a process to influence cache access latency of other processes: cache conflict and *clflush command.* Cache conflict miss happens when a memory line is allocated in a cache set that is already occupied. Then a cache block in that set would be replaced by the new one. If the owner of the evicted cache block requests that block later, it would suffer cache miss and observe a longer latency. *Clflush* command enables processes to evict its memory line from cache. If other processes which share the evicted memory line try to access it, they would observe a cache miss. These features can be exploited to modulate cache timing and covertly transmit information [46].

Three primary ways are proposed to implement cache timing channel: prime+probe [52,34,11,24], flush+reload [59,6,61,58], and evict+time [37]. Evict+time can only be implemented in side channel because it requires spy to send service requests to victim and time the latency of responses. To implement prime+probe timing channels, as shown in Figure 1a, the spy primes the cache sets with its memory lines, and the trojan replaces them to create conflict patterns and encode bits. The spy then probes the same cache sets,



(a) Cache Access Diagram for Round-Robin Protocol



(b) Cache Access Diagram for Parallel Protocol

Fig. 2: Prime+Probe Communication Protocols

and measures the cache access latencies to infer the trojan's activity. As shown in Figure 1b, for flush+reload, the spy flushes shared memory lines from cache set first. Then, the trojan encodes bits by either accessing the memory lines or by staying idle. The spy reload those shared memory lines and measure access latency to figure out whether trojan has accessed them. Among above implementations we discussed, prime+probe has least number of prerequisites to realize the attack, and it is merely based on cache accesses without specifically depending on any shared address patterns seen in other types of attacks.

There are several communication protocols to implement prime+probe cache timing channels in real systems. The cache access activity of trojan and spy can be manifested in a round-robin [11] or in a parallel fashion [55]. For round-robin protocol, the trojan and spy have a schedule and only access cache when the other one finish its activity. As shown in Figure 2a, the trojan encodes bits by either accessing cache or staying idle. And the spy access cache after trojan's encoding (access or idle). Then, the trojan starts encoding next bit when spy finish its access. For parallel implementation, trojan and spy don't have accurate information about each other's timing. At the beginning, the trojan transmits predetermined symbol sequences to inform spy the approximate bit boundary. Then the trojan encode one single bit with multiple cache accesses in order to make sure that spy would receive its information. As shown in Figure 2b, the trojan access cache four times to encode one bit by using repetition coding and spy keeps accessing cache to make sure it won't miss the trojan's information. Trojan may also implement different encoding strategy. The trojan could create conflict misses on either a specific group of cache sets [52, 24] or multiple groups of cache sets [41, 53]

2.2 Prefetcher Module in Processors

Data prefetching has been utilized to bridge the performance gap created between processors and DRAM. Prefetching can be implemented based on both software and hardware. Software prefetching requires support from architecture which enables prefetch instruction to bring memory lines from DRAM. For instance, Intel®Xeon Phi™ [13] provides *vprefetch1* instruction which brings a 64B memory lines to L2 cache and *vprefetch0* to push it further to L1 cache. When an instruction tries to bring an unavailable memory line, the system would stay silent and ignore it rather than throw page faults. The prefetch instruction could be inserted by expert programmers or automatically generated by state of art compilers.

The on-chip hardware prefetchers work by monitoring the cache misses, and predicting the memory addresses that satisfy CPU's data needs in the near future. Hardware prefetchers are, by default, enabled in most of modern processors. The common types of hardware prefetchers based on spatial locality are *Stream*: loads next sequential addresses in the page, and *Stride*: which brings the addresses at a fixed stride from the requested address. To satisfy temporal locality, prefetchers use Global history buffer-based policy that predicts the next cache reference based on previous access pattern [36].

3 Threat Model and Assumptions

Our attack model assumes that a trojan has accesses to sensitive information that a spy is trying to steal by observing the access to one of the largest shared hardware structures, namely caches. We evaluate our design using the covert timing channels where an insider trojan process intentionally manipulates the hardware resource timing to communicate secrets. We note that our solution approach can also prevent side channels. This is due to the fact that malicious trojan (covert channels) and benign victim (side channels) have similar interactions with the spy, and manifest themselves similarly in terms of cache timing modulation behavior. The only difference between them is that a trojan communicates intentionally, and therefore, is far more difficult to detect or prevent.

Without loss of generality, the spy and trojan are assigned to different cores that share hardware caches. (e.g., Last Level Cache or LLC). Note that, under the system security policy, any form of inter-process communication between the trojan-spy is prohibited by OS since the trojan has access to sensitive data that is usually not available to the spy. Besides, the processes from different security domains have no shared library and flushing cache line contents (e.g., via clflush instruction) belonging to the different process shall not be allowed. Therefore, through isolation mechanisms that prevent accesses to shared libraries simultaneously, the OS and VM can prevent certain types of timing channel implementations such as Flush+Reload [59]. Prime+probe techniques, which doesn't require any such sharing prerequisites, will still work under this

system security model. Therefore, our threat model includes the most potent *Prime+Probe*-based attacks (that does not require any shared memory between trojan and spy) to create conflict patterns for covert communication. The spy primes some cache sets with its own data blocks where the cache sets used in timing channels could be separated into one or more groups.

In this article, we demonstrate timing channels that assume a trojan covertly encoding bit '1' through evicting cache blocks owned by the spy, and transmitting bit '0' simply by staying idle. Without the loss of generality, we note that such a transmission scheme may be extended to other encoding schemes where multi-bit encoding is performed with multiple latency bands [56], and the timing of trojan and spy's activity could be round-robin and parallel. We observe that a covert channel, where trojan and spy operates in parallel, is similar to side channel since there is no synchronization between two processes. Therefore, our evaluation results on covert channels with parallel protocols capture side channel behavior.

Finally, we note that certain system administrators could simply terminate the program instead of deploying mitigation strategies such as PrODACT. This may be undesirable on two counts: 1. In case of side-channels, where a spy intentionally create conflict misses with innocent victim, terminating both involved processes may impact benign applications unnecessarily. 2. When two benign applications compete for cache resources temporarily during short time periods, both benign applications will be terminated. In contrast PrODACT would improve the access latency through prefetching their cache blocks.

4 Motivation

4.1 Uncovering Timing Channels

The requirements of cache-based timing channels include: 1. the spy's ability to distinguish between cache hit and miss latencies, and 2. the trojan's capability to orchestrate a series of cache hits and misses for the spy's observation. As stated in Section 3, the spy cannot directly communicate with the trojan to obtain information. Therefore, the spy has to infer trojan's communicated bits covertly through measuring cache latencies for accesses.

Initally, the spy primes all of the cache sets through filling them with its own cache blocks. The trojan transmits bit '1' by evicting all of the cache blocks owned by the spy, and transmits bit '0' by staying idle (i.e., does not replace spy's blocks). After trojan's activity, spy probes and measures the access latency for those cache blocks it had primed previously. Let us demonstrate a timing channel attack (implemented on Gem5 simulator [7]), and record the spy's access latencies when it performs the probe phase. Figure 3, the red dashed line shows the spy's observations. The latencies are either above 2000 cycles or below 700 cycles when trojan transmit bits '1' and '0' respectively. To decipher the bits, the spy can simply pick a threshold equal to the mean of all latencies, and decipher the communicated bit as '1' if latency is larger than

the threshold, and '0' if the latency is less than the threshold. In this case, the spy would get 0% error rate. The above example shows that it is essential for the spy to have a clear decision boundary (threshold) to decipher the bit transmitted by the trojan.

As one straightforward observation, we can infer that by making this decision boundary non-perfectly separating, the spy will no longer be able to reliably decipher the covertly communicated bits. We can achieve this by *increasing* the spy's observed latency when trojan transmits bit '0', and through *decreasing* the spy's access latency when trojan transmits bit '1'.

4.2 Defense using Hardware Prefetcher

Hardware prefetcher has the ability to bring data blocks into the cache even before those blocks are actually being consumed by the processor. We note that such prefetchers can be leveraged to artificially increase or decrease cache access latencies in a controlled manner.

The prefetcher has the ability to obfuscate cache timing channels in the following two ways:

- *Convert certain Cache Misses to Hits:* If the prefetcher brings the spy's blocks back into the cache right after trojan evicts them, the spy would suffer from less number of cache misses during its probe phase. In other words, the access latency observed by spy would be lower than expected and this lowers the decision boundary.
- *Convert certain Cache Hits to Misses:* If the prefetcher replaces some of the spy's cache blocks that the trojan did not evict, the spy would experience greater number of cache misses during its probe phase. In other words, the access latency observed by spy would be higher than expected and pushes the decision boundary.

In the case of timing attack that we discussed in Section 4.1, in order to introduce frequent errors in timing channels, we should make the spy's cache latencies during '0' and '1' bit transmissions to be indistinguishable. That is, we should flip half of cache hits that occur during '0' bit transmission to cache misses; similarly, we ought to flip half of cache misses to hits when bit '1' is being transmitted.

In an 8-way set associative cache, after the trojan encodes bit '0' by staying idle, the spy should normally observe 8 hits during its probe phase. However, right before spy's probe, if we replace some of spy's blocks that were left intact from its last prime phase, the spy would suffer from cache misses that should have not happened. Therefore, spy experiences more misses than expected which would confuse the spy from correctly inferring the transmitted '0'.

Similarly, after trojan encodes bit '1' by evicting spy's cache blocks, the spy would normally suffer 8 misses because all of its data blocks are evicted from cache set. If we can prefetch some of data blocks owned by the spy before it probes, the spy would suffer lesser number of cache misses, and the latency

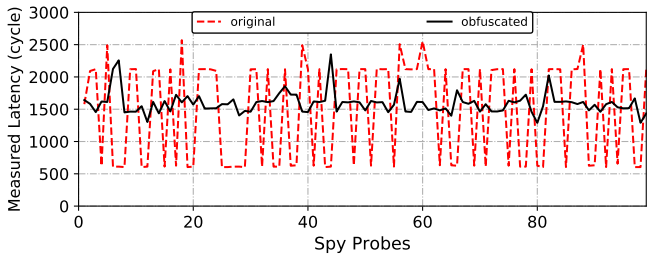


Fig. 3: Spy's observation of access latencies before and after prefetcher obfuscates the hit-miss pattern in cache accesses.

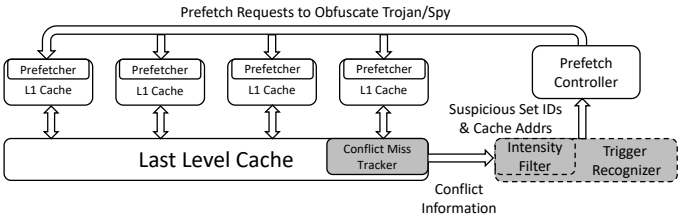


Fig. 4: PrODACT Design Overview. The white blocks stands for existing hardware. The gray blocks stands for the components belonging to PrODACT. Among them, the gray blocks with solid frame (Conflict Miss Tracker) denote hardware add-ons, while those with dotted frame are implemented in software.

that it measures would be lower than expected. This can confuse the spy from correctly inferring the transmitted '1'.

To make the spy's observed latencies for bit '1' and bit '0' to overlap in value as much as possible, we make the spy to observe about 4 misses during its probe phase regardless of what the trojan transmits. This is done by flipping cache misses and hits as described above, effectively obfuscating the timing channel. The observed latencies on the spy side during transmission of alternating 1's and 0's after obfuscation is shown in Figure 3. For any given bit, we observe that the obfuscated latency is significantly different from the obfuscation-free cases, and the latency difference between bits 1 and 0 is practically non-exsistant. If spy uses the mean latency as its decision boundary, the error rate would be 53% which is just as good as a random guess. If a sophisticated spy hopes to separate the latencies into multiple decision regions to improve communication quality, it requires knowledge about distribution of access latency values which needs thousands of measurement and would result in overfitting of samples. In general, with the noise shown in this example, it is impractical for the trojan and spy to build a robust cache timing channel.

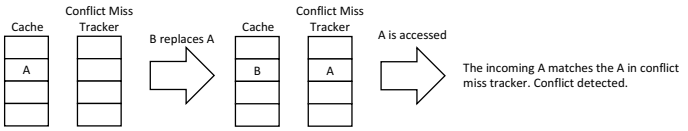


Fig. 5: The changes in cache and conflict miss tracker when a replaced memory line A is accessed again.

5 PrODACT Design

The design of our PrODACT framework involves three important modules: Conflict Miss Tracker, a *two-level*, low-cost Trigger Pattern Recognizer, and a Prefetch Controller shown in Figure 4. The Conflict Miss Tracker, that is built into shared Last Level Cache (LLC), collects information on cache misses and identifies the conflict misses among them. The two-level Trigger Pattern Recognizer analyzes the conflict miss patterns, and identifies the cache sets that are likely exploited by the malicious processes. Once suspicious sets and the corresponding memory addresses are sent to the prefetch controller, it sends prefetch requests to obfuscate the trojan-spy covert communication.

5.1 Conflict Miss Tracker

Conflict misses occur exclusively in set-associative caches when blocks are pre-emptively replaced from the cache even before the full cache capacity is reached. That is, *when a core A's cache block is replaced by a core B's cache block prematurely, and the core A accesses the same block again (that had been recently replaced), a conflict miss occurs.* Such conflict misses occur when many blocks that map to the same cache set are accessed successively, and the cache does not support enough associativity to accommodate all of the blocks. Note that such conflict misses would have never happened in a fully associative cache.

In order to track such conflict misses, a simple hardware buffer (Conflict Miss Tracker) maintains a list of addresses that are replaced during cache misses in LLC, along with the corresponding owner core ID for that block. If a currently replaced address to the cache is not in this hardware tracking buffer, it would be added. Upon every cache miss, the incoming cache address is checked against the buffer entries to verify if a conflict miss had occurred. If the incoming address to the cache is found in the buffer, we can infer that this address was recently replaced by another cache block, and hence is recorded as a conflict miss. An example of conflict miss identification is shown in Figure 5 where the memory line A suffers cache miss because it's replaced by memory line B. We note that such cache conflict miss identification techniques in hardware have been proposed for performance analysis reasons [48]. We note that the conflict miss tracker could achieve a high accuracy with less than 3% area overhead in L2 cache and 1.5% access time overhead.

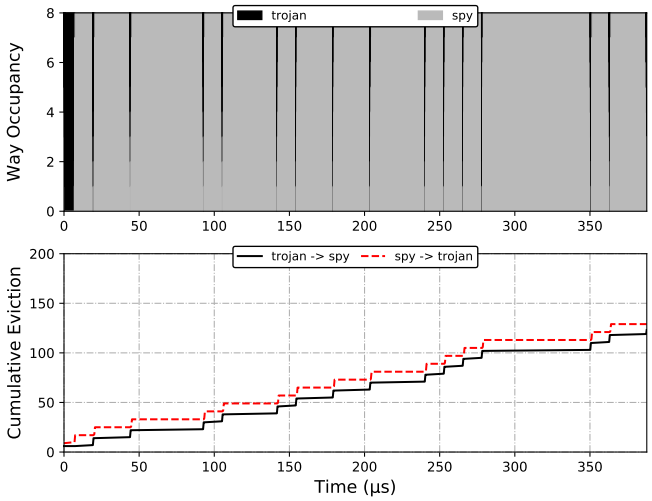


Fig. 6: Cache occupancy of trojan/spy and their cumulative number of evictions

Note that mutual cache block eviction activity is very common in cache timing channels. During such mutual evictions, both the incoming and outgoing addresses will be found in the conflict miss tracker. Since the trojan and spy pairs will do the evictions repeatedly using multiple addresses to covertly communicate with each other, the identities of the trojan and victim can be easily found. The output of Conflict Miss Tracker is the list of evicted addresses, the owners of evicting and evicted memory addresses (likely trojan and spy processes).

5.2 Trigger Pattern Recognizer

Once a Conflict Miss Tracker identifies a series of cache misses, the next step is to figure out whether there is a potential cache timing channel. We propose a two-level mechanism, where the first-level filters conflict misses that are likely benign, and the second-level uses pattern recognition to identify likely timing channels.

As mentioned in Section 1, the Prime+Probe typically involves three phases: 1) spy's prime, 2) trojan/victim's activity, and 3) spy's probe. In the Prime phase, in order to observe the trojan's activity, spy primes all of the cache sets by bringing its own blocks into those sets. Then the trojan transmits encoded bits by evicting spy's cache blocks or by staying idle. After trojan's activity, spy probes its primed cache blocks, and measures the access latency to infer the bit that was transmitted by the trojan. The entire prime+probe procedure has two features which are usually not found in benign processes: 1. The number of conflict misses is usually much higher because each prime+probe operation contains multiple conflict misses. To transmit with a reasonable

bandwidth, the spy and trojan would suffer more conflict misses than normal applications. 2. The conflict miss pattern between the spy and trojan happen in an alternating fashion.

Before doing the pattern analysis of conflict misses, we first filter the processes which have few conflict misses because they are either benign processes or an extremely low-bandwidth channel (as per DoD Standard [17], timing channel bandwidths below 0.1 bps are considered unavoidable in any real system). For the conflict-intensive processes, we send the conflict information to second level analyzer to do the further analysis.

Let us denote process A evicting process B's cache line as $\mathrm{A} \rightarrow \mathrm{B}$. In prime+probe attack, during probe phase there would be multiple spy $\rightarrow$ trojan conflicts. When trojan encodes information, we can observe trojan $\rightarrow$ spy. The goal of trigger pattern recognizer (in its second step after filtering benign processes) is to extract the alternating patterns of trojan $\rightarrow$ spy and spy $\rightarrow$ trojan.

To illustrate our trigger detection algorithm, we run a timing channel attack on an 8-way associative cache. Trojan transmits '1' by evicting eight ways primed by spy, and transmits '0' by staying idle. We record the way occupancy and the cumulative count of the eviction pairs. As shown in Figure 6, the spy primes 8 cache ways, and then, trojan evicts the spy's blocks. The spy's cache set occupancy decreases first and then increases back to 8, while the trojan's way occupancy shows the opposite behavior. These changes in cache way occupancy is caused by *trojan* $\rightarrow$ *spy* and *spy* $\rightarrow$ *trojan* eviction patterns. For different protocols, the order of these two kinds of eviction could be different, but the pattern of loss and gain in way occupancy resulting from cache conflict misses is the same. To extract the pattern we maintain a counter for each pair of processes in every cache set. For a pair of processes (a, b) and a cache set i, we record the vector (a, b, i). When a $a \rightarrow b$ eviction occurs on the cache set i, the counter corresponding to vector, $C_{a,b,i}$ increases by one. Similarly, if a $b \rightarrow a$ eviction happens, the counter decreases by one. The value of $C_{a,b,i}$ equals to the difference of cumulative number of $a \rightarrow b$ evictions and $b \rightarrow a$ evictions. As shown in Figure 6, the *difference of two cumulative numbers* first increases and then decreases in one prime+probe operation. During covert timing channel activity, we could observe the value switch back and forth for multiple times.

If the number of cumulative switches of the value (difference of the two counters) on a cache set becomes higher than a threshold, we infer repeated, *intentional* cache block replacements by trojan/spy pair to manipulate cache timing. Consequently, the trigger identifies such cache sets as suspicious for communication between the corresponding pair of owners a, b.

We note that our two-level trigger pattern detector improves scalability by filtering benign processes and their related conflict misses. This vastly reduces the number of pairs running in the system that need to be analyzed further for potential timing channels.

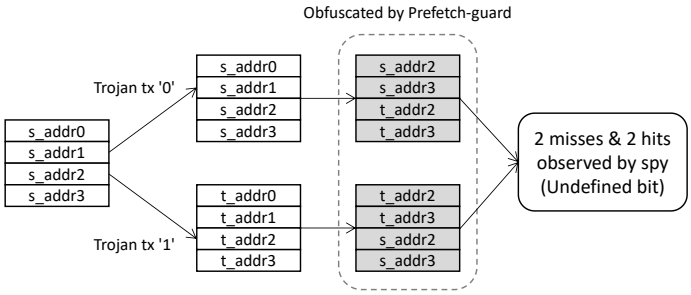


Fig. 7: Bit Boundary obfuscation by PrODACT

5.3 Prefetch Controller

The prefetch controller receives the information about suspicious processes and the cache addresses that are involved in timing channel-related activity. Based on this information, the prefetch controller analyzes the timing of the prime+probe and delivers prefetch requests to the L1 cache prefetchers to obfuscate cache timing channel.

In order to thwart the trojan-spy communication, the prefetch controller needs information about the addresses of memory lines that the trojan and spy exploit to create conflict misses. During their communication, these memory addresses are frequently involved in conflict misses, and therefore, would be recorded by Conflict Miss Tracker. After the suspicious cache sets are labeled, these exploited addresses are also sent to the prefetch controller. Then the prefetch controller issues requests to the L1 prefetcher to bring back memory lines for a specific times to obfuscate trojan-spy communication.

As mentioned in Section 4, the goal of mitigating a cache timing channel is to make spy receive bits incorrectly. Figure 7 shows our prefetch controller making the number of cache misses and hits observed by the spy to be the same irrespective of trojan's activity. In our illustration, for a 4-way associative cache, the spy observes zero misses when trojan transmits bit '0' and four misses when trojan transmits bit '1'. In the absence of any defense, the spy could easily discern the bit because the difference of latency of cache accesses is high. To obfuscate the spy's probe phase, the prefetch controller makes spy suffer from two misses all the time. When trojan encodes bit '1', the prefetcher brings back half of the spy's memory lines after the trojan's activity (Note that trojan-spy pair could be inferred by our Trigger pattern recognizer module 5.2). The spy would observe 2 conflict misses rather than 4 misses. And when trojan encodes bit '0', trojan's two memory lines are brought to the cache before spy's probe phase. Since the number of observed misses becomes independent on trojan's activity, it is impossible for the spy to infer trojan's activity by measuring cache access latencies.

6 Experiment Setup

We evaluate PrODACT using *Gem5* [7], a cycle-accurate, full-system simulator. We configure Gem5 with four x86 cores, 32 KB private L1 and 4 MB, 16-way shared L2 caches. All the experiments are run on full system mode under Linux kernel version 2.6.32.

6.1 Cache Timing Channel Attacks

Encoding	Timing	Attack Implementations
Single-group	Parallel	[52,34]
Single-group	Round-robin	[11,24]
Multiple-group	Parallel	[40,33]
Multiple-group	Round-robin	[41,53]

Table 1: Cache timing attack classes studied in our paper.

We launch prime+probe attack on L2 shared cache and run the trojan and spy on different cores. As shown in Table 1, we study four variants of cache timing protocols used by adversaries. In single-group attack, the spy primes a single cache set through generating addresses that map to the same cache set. The trojan transmits '1' by replacing all of spy's cache blocks from the set with its own addresses, and transmits '0' by staying idle. The spy decodes bits by measuring the access latency of cache blocks through re-issuing the set of addresses used during its prime phase. A higher latency represents bit '1', and the lower latency represents bit '0'. In multiple-group attack, the spy primes two sets of cache blocks. The trojan transmits '1' by evicting the spy's cache blocks in the first set, and transmits '0' by evicting the blocks in second set. The spy probes both sets and deciphers the bit. Without loss of generality, each group in single/multiple-group attacks can include multiple cache sets.

For round-robin channel attacks, the spy and trojan take turns in accessing the cache. For parallel protocols, the spy and trojan operate simultaneously, where the spy probes at a high frequency and the trojan evicts multiple times for a bit to make sure that spy infers the information covertly.

6.2 Stress Test

To test the performance of PrODACT when run alongside cache-intensive application, we run stress tests with single-group round-robin cache timing channel and PrODACT using a micro-benchmark with high memory access intensity. The micro-benchmark repeatedly keeps allocating different sizes of memory blocks and frees them in order to create high memory access intensity.

The size of memory lines allocated by micro-benchmark ranges from 1MB to 4 MB which is significantly larger than L1 and L2 caches themselves. We measure the effectiveness of PrODACT by observing the bit error rate of cache timing channels caused by our prefetch-based defense mechanism.

6.3 Benign Workloads

To evaluate the influence PrODACT has to benign workloads, we test PrODACT on both cache-intensive[26] and non-cache-intensive workloads from SPEC2006[22] benchmarks. We mix the cache-intensive and non-cache-intensive workloads, with each workload running on individual cores. Each pair of benign workloads are analyzed by trigger pattern recognizer and the number of counter switches are computed in sliding windows with duration of 1 second. We record the maximum number of switches that happen during this one second interval.

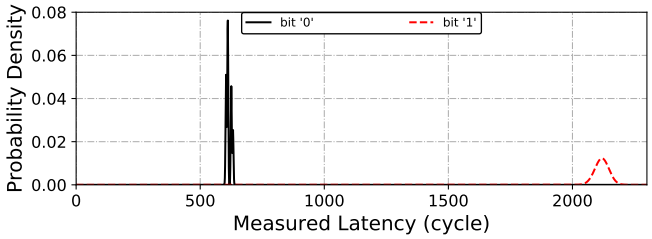
7 Evaluation

7.1 Analysis on Cache Timing Channel

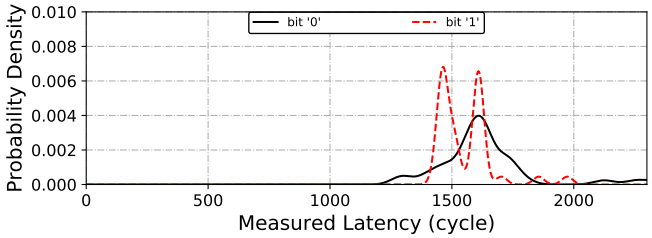
(a) Single-Group, Round-robin Attack. We record the spy's observed cache latencies with and without obfuscation by PrODACT. The estimated conditional probability densities of spy's cache latencies are shown in Figure 8. When there is no obfuscation (as shown in Figure 8a), the measured latencies for bit '0' and '1' transmissions are significantly distinguishable. Therefore, the spy can easily pick a threshold equal to the mean of all observations to decipher bits.

With PrODACT enabled, Figure 8b shows that the estimated conditional probability densities change significantly. Specifically, the latency distributions of bit '0' and bit '1' overlap significantly, so that there is no clear boundary to separate them. With the thresholding-based detection mechanism, the bit error rate for the spy is as high as 53%, which practically disables any communication. We note that spy and trojan can transmit predetermined symbol sequences to reveal these distributions, and find an optimum threshold. Even if this is the case, our experiments show that the lowest achievable error rate is 37%, which is still too high for any practical communication.

(b) Multiple-Group Round-robin Attack. We implement a two-group round-robin attack where we assume that trojan evicts the first group when transimitting bit '1' and the second group when transmitting bit '0'. In the spy's probe phase, it observes two different latencies from the two cache set groups. We visualize spy's observations in a 2-D plots shown in Figure 9. The horizontal axis is the observed latency for the first group and the vertical axis is the observed latency for the second group. Figure 9a shows the spy's cache latencies during the attack. When bit '0' is transmitted, the trojan evicts spy's

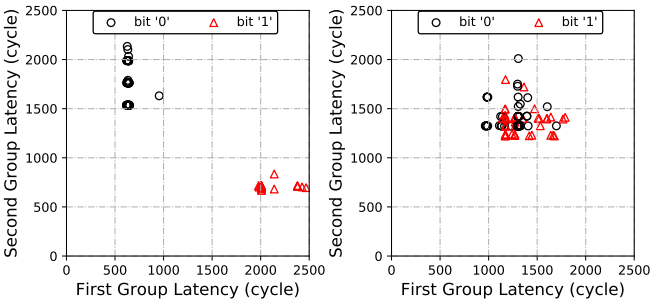


(a) Attack



(b) With PrODACT

Fig. 8: Conditional probability densities of spy's cache latency in single-group, round-robin attack



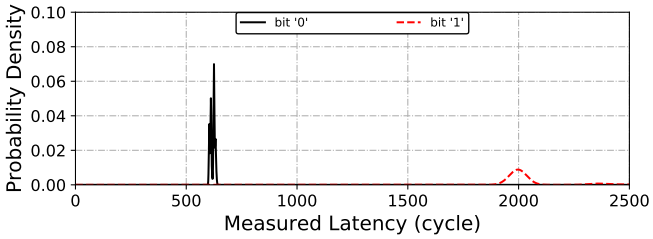
(a) Attack (b) With PrODACT

Fig. 9: Spy's cache latencies in two-group, round-robin attack

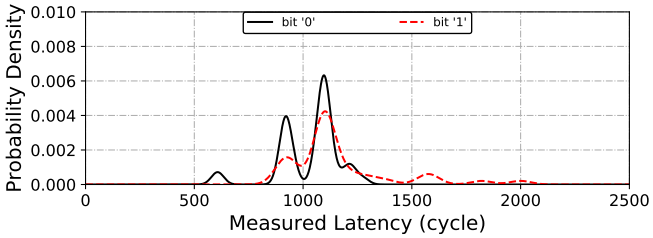
memory lines in the second group and let those in the first group to remain in the cache. So the spy observes a higher latency for the second group of cache blocks, and a low latency for the first group. The converse effect is observed when bit '1' is transmitted. Due to the difference in latency bands between these two cache set groups under different bit transmissions, the spy could easily find a decision boundary to separate the latency bands and decode the secret bits.

With PrODACT, the spy observes about four cache misses on each cache set regardless of the bit transmitted. There does not exist a clear decision boundary for bit '0' and '1' as seen in figure 9b. If the spy forcibly applies the decision boundary approach, the bit error rate would be as high as 35%.

(c) Single-group, Parallel Attack We note that the decoding of parallel protocols is slightly more difficult than round-robin protocols since the boundaries

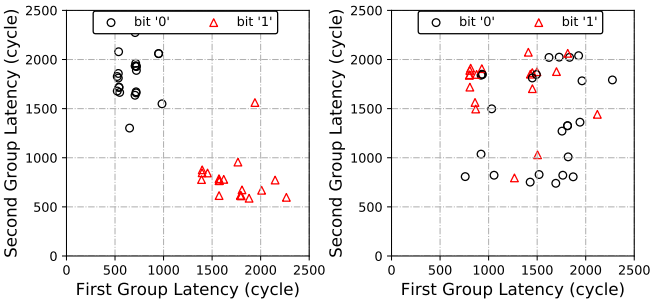


(a) Attack



(b) With PrODACT

Fig. 10: Conditional Probability densities of spy's cache latencies in single-group, parallel attack



(a) Attack (b) With PrODACT

Fig. 11: Spy's cache latencies in two-group, parallel attack

between contiguous bits may overlap. A practical way is to have the spy record the latencies, and only keep the values corresponding to the majority of relative latency observations. In Figure 10a, we plot the spy's measured latencies for bit '0' and '1' respectively. It is straightforward for the spy to find the threshold that equals to the mean of all observations for decoding bits. Figure 10b shows that PrODACT disrupts this cache timing channel by making the conditional probability densities of bit '0' and '1' to be heavily overlapping. The spy's corresponding bit error rate for bit reception would be 56% if the mean of all observations is chosen as the threshold.

(d) Multiple-group, Parallel Attack The decoding strategy for multiple-group parallel attack is similar to single-group, parallel attack. Once the bit boundary is found, the spy can decode every bit based on the majority of relative

observations. We plot the spy's latency in Figure 11a. The error rate is 0% without obfuscation.Figure 11b shows that some of the triangle-marked points (bit '0') shift to the right half of the plane while many circle-marked points (bit '1') move to the left half when PrODACT is activated. The decision boundary works with 70% error rate (or 30% error rate if inverse logic is applied), which prevents any feasible cache timing channel.

7.2 Analysis on Benign Applications

As discussed in Section 5, PrODACT only obfuscates the access latencies when suspicious behaviors are detected by our trigger pattern recognizer. When there is a $A \rightarrow B$ and $B \rightarrow A$ evict pattern, the counter in trigger pattern recognizer would switch. If the number of counter switches for a pair of processes is larger than a threshold, PrODACT would issue prefetch requests to L1 cache and disrupt any potential timing channels. As shown in Table 2, the largest number of counter switches per second for benign workload pairs is less than 40, while we observe at least 1,000 switches per second for all of the realistic timing channels studied in our work. Because of this huge gap between benign workloads and attacks, we note that the threshold separating the malicious timing channels and benign applications can be set easily. A 100 counter switches/second can be a conservative threshold to make sure that no benign application would be disrupted by PrODACT, while all of the realistic cache timing channels are correctly captured.

Table 2: Maximum number of switches/second observed in highly cache-intensive benchmarks.

Benchmark	Max. number of switches / second
GemsFDTD, hmmer, xalancbmk, namd	6
bzip2, gobmk, h264ref, namd	25
bzip2, gobmk, sjeng, mcf	34
bzip2, gobmk, sjeng, specrand	26

7.3 Evaluation of Stress Test

The performance of PrODACT running alongside cache-intensive applications is shown in Figure 12. The x-axis is the size of memory blocks being accessed by micro-benchmark. The y-axis shows bit error rate (measured as the percentage of bits received incorrectly or not received at all). The mean and standard deviation of bit error rates are observed for round-robin single-group cache timing channels. The average bit error rates are all above 40%. The standard deviation slightly increase when the number of memory blocks accessed by micro-benchmark increases. The lowest bit error rate that we have observed is

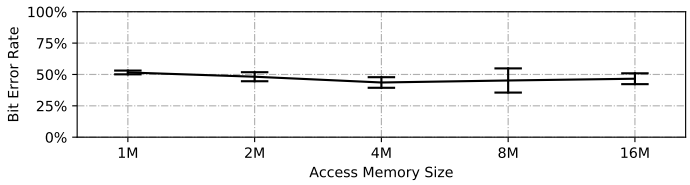


Fig. 12: Bit Error Rates of Cache Timing Channels when run alongside background noise (using a micro-benchmark) with different memory access levels.

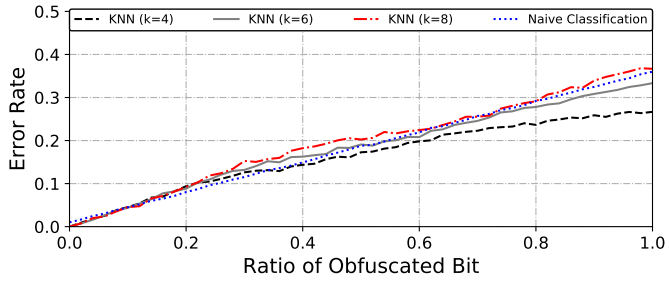


Fig. 13: Error rates of different classification techniques with varying fractions of obfuscated bits

37%, which is still sufficient to obfuscate any communication. In general, the cache timing channel could still be successfully mitigated by PrODACT when memory bandwidth is severely occupied.

7.4 Case Study: Defense against Smarter Spy

As sophisticated adversaries, the spy and trojan may try to learn the *latency distribution under obfuscation* and manage to recover their communication. In this subsection, we are going to demonstrate that even an advanced classification algorithm cannot recover the cache timing channel with PrODACT deployed effectively.

For illustration, we will show adversaries that launch a sophisticated attack using multi-group, round-robin protocol. In this attack scenario, the trojan transmits predetermined symbol sequences so that the spy is able to collect data sets of latency observations labeled with bit '0' or '1'. Now the spy can leverage the latency data sets to decode bits even if the latency observations are obfuscated. Specifically, the spy can formulate the decoding process as a classical classification problem using machine learning approaches. K-nearest neighbors (KNN) [18] is an efficient algorithm for this type of scenario, and we assume that the spy leverage this algorithm to classify bits. For a new observation, the KNN algorithm computes its distance to all labeled observations and picks the K closest observations among them. It classifies the new observation using the majority of these K closest neighbors.

In this experiment, we activate PrODACT in a probabliistic way such that only a fraction of the bits (a tunable parameter) would be obfuscated. We perform 1000 experiments to test the bit error rate using KNN ($K = 4, 6, 9$), as well as the naive decision boundary-based classification we discussed in Section 7.1. The sample mean of bit error rates is recorded for every fraction. In Figure 13, with all of the bits obfuscated, the naive decision boundary leads to 35% bit error. The KNN algorithm with relatively low K value (K=4) improves the communication quality slightly compared to the naive classification, but the error rate is still as high as 25%, which is still sufficiently noisy to correctly decipher bits. The KNN algorithms with higher K value is worse than the naive classification because the two sets of samples are mixed significantly. In general, the error rate is still high enough to stop the information leakage even with smarter spy.

Figure 13 also shows that it is *not* necessary to obfuscate every bit. In fact, to make the cache timing channel suffer from 20% bit error rate, enabling PrODACT for 60% of the bits is sufficient. We note that the system administrators could tune the corresponding obfuscation rate to effectively trade-off system security and memory bandwidth overheads.

8 Discussion

In this section, we will discuss the extension of PrODACT for flush+reload attacks with a relatively small change in our design. Later, we will discuss how to defeat adversaries who try to evade detection by lowering their bandwidths.

8.1 Flush+Reload Attack

In contrast to prime+probe attack that rely on conflict misses, flush+reload attack exploit *clflush* command to evict memory lines of victim/trojan. During the flush phase, the spy flushes the memory lines using *clflush* command and waits. Then, the trojan encodes bit '1' by accessing the flushed memory lines, and encodes bit '0' by stay idle. During the reload phase, the spy reloads all of the flushed memory lines and measures the cache access latency. To detect flush+reload attack, we record the addresses of flushed memory lines (similar to recording the replaced conflict memory addresses in prime+probe). The *clflush* commands and the conflict memory address loads by the spy or trojan would be seen alternatively if there is a flush+reload-based timing channel. To extract this pattern, we modify the counter in trigger pattern recognizer to make it increase if we observe a *clflush* command and decrease when a flushed memory line suffers from cache miss. After detecting a flush+reload attack, the prefetcher brings back memory lines to make spy suffer from the same number of cache misses regardless of trojan's activity in the cache. With this relatively small modification to PrODACT, our design could successfully obfuscate flush+reload timing channels. We note that the trigger pattern recognizer

for flush+reload attack and prime+probe attack can be run simultaneously as well.

8.2 Low Bandwidth Timing Channels

Fortunately, our PrODACT design offers flexible granularity of monitoring by providing the capability to adjust the thresholds for timing channel detection. For a low bandwidth timing channel, the trojan $\rightarrow$ spy and spy $\rightarrow$ trojan eviction would still appear in an alternating pattern. To detect such low bandwidth timing channels, we could lower or remove the threshold in our filter during trigger pattern recognition. While this may produce more processes for further analysis, low bandwidth channels could be effectively detected from further analysis by the trigger recognizer module.

9 Related Work

Side and covert channels have been implemented on various types of hardware in a number of ways. To name a few, storage [15,16], power analysis [30, 30,14,9,44], program execution [35,60] or access latency [57,1,56,19,10,27] are among the prominently studied information leakage channels. In many such channels, the adversary can reveal secrets about sensitive processes or endanger system security without leaving any trace. Prior works have proposed counter strategies for power and storage channels through memory safety and inspection [21,8,45,42,47,43]. For timing channels, the access time directly influences the performance of processes. Techniques such as injecting noise may lead to severe performance degradation of all running processes. Among all of the timing channels, cache timing channels are notorious because they can exploit numerous cache sets with relative ease.

Cache side- and covert timing channels have been demonstrated on real hardware in several prior studies [33,28,3,4]. To detect and prevent these cache timing channels, solutions have been proposed in [10,11,39,55]. CC-hunter [11] proposes a generic framework for cache covert timing channel detection using autocorrelation between cache conflict misses. Chiapetta et al. [12] and HexPads [39] leverage performance counters to correlate trojan and spy's activities for detection. ReplayConfusion [55] records and replays cache access traces from the trojan and spy and detects cache timing attacks based on differences of cache misses. Most of these works are aimed at detecting cache timing channels.

A number of hardware mitigation schemes have been proposed to defend against the cache timing channel attacks. For L1 cache timing attack, Bao et al. [5] explore the implication of faster 3D integrated caches to perform low cost obfuscation. CATalyst [31] propose a secure cache partition for security-sensitive application to access secretive data. This mechanism is limited to protect voluntary victim processes that utilize the secure partition. SecDCP [49]

performs dynamic cache partition on different security domains. However, it suffers from scalability issues due to limited number of cache partitions. Mitigation mechanisms that modify cache line replacement algorithms such as SHARP [54] and RIC [29] influence all applications and may cause considerable cache performance degradations.

Fuchs et al. [20] propose a disruptive prefetching scheme that utilizes existing prefetch policies to pollute caches. They also propose the set balancing mechanism to bring random addresses on every other cache set for each cache replacement to confuse the spy. This approach only works for L1 caches, and may largely reduce the inherent cache performance for benign applications. Differently, PrODACT can handle multiple timing channel variants effectively regardless of the number of cache blocks used in communication. Our proposed mechanism also has less performance impact due to the prefetching on *targeted cache sets*.

10 Conclusion

In this article, we propose PrODACT, an efficient, scalable and low-cost solution to prevent information leakage through cache timing channels. PrODACT analyzes the conflict misses in shared caches, and targets the cache sets that are likely to be exploited by malicious processes. Hardware prefetchers are leveraged to obfuscate cache accesses on suspicious cache sets. PrODACT retrieves back the cache blocks owned by trojan and spy to obfuscate spy's observation using cache latencies. We evaluate using several cache timing channel protocols. With PrODACT, we observe that the cache timing channels suffer an average 50% bit error rate (with minimum 30%) which makes it very hard or impossible for spy to decipher any useful information.

11 Acknowledgement

This material is based on work supported by the US National Science Foundation under CAREER Award CCF- 1149557 and CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of the NSF or SRC.

References

1. Murugappan Alagappan, Jeyavijayan JV Rajendran, Miloš Doroslovački, and Guru Venkataramani. Dfs covert channels on multi-core platforms. In *25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2017.
2. Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

3. A. Andreou, A. Bogdanov, and E. Tischhauser. Cache timing attacks on recent microarchitectures. In *IEEE International Symposium on Hardware Oriented Security and Trust*, 2017.
4. Alexandres Andreou, Andrey Bogdanov, and Elmar Tischhauser. Cache timing attacks on recent microarchitectures. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*. IEEE, 2017.
5. C. Bao and A. Srivastava. 3D integration: New opportunities in defense against cache-timing side-channel attacks. In *IEEE International Conference on Computer Design*, 2015.
6. Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. ooh aah... just a little bit: A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014.
7. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.
8. Marco Bucci, Luca Giancane, Raimondo Luzzi, and Alessandro Trifiletti. Three-phase dual-rail pre-charge logic. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006.
9. Abhishek Chakraborty, Ankit Mondal, and Ankur Srivastava. Correlation power analysis attack against stt-mram based cyptosystems. *IACR Cryptology ePrint Archive*, 2017:413, 2017.
10. Jie Chen and Guru Venkataramani. An algorithm for detecting contention-based covert timing channels on shared hardware. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 1. ACM, 2014.
11. Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.
12. Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, pages 1162 – 1174, 2016.
13. George Chrysos. Intel® xeon phi coprocessor-the architecture. *Intel Whitepaper*, 176, 2014.
14. Christophe Clavier, Damien Marion, and Antoine Wurcker. Simple power analysis on aes key expansion revisited. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014.
15. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.
16. Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.
17. Department of Defense Standard. *Trusted Computer System Evaluation Criteria*. US Department of Defense, 1983.
18. Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. Wiley, New York, 1973.
19. Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.
20. Adi Fuchs and Ruby B. Lee. Disruptive prefetching: Impact on side-channel attacks and cache designs. In *ACM International Systems and Storage Conference*, 2015.
21. Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2011.
22. John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
23. Fan Yao Miloš Doroslovački Hongyu Fang, Sai Santosh Dayapule and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetchers to defend against cache timing

channels. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. IEEE, 2018.
24. Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *IEEE International Symposium on High Performance Computer Architecture*, 2015.
25. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ACM on Asia Conference on Computer and Communications Security*, 2016.
26. Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.
27. Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on gpus. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 167–172. ACM, 2017.
28. M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *ACM/EDAC/IEEE Design Automation Conference*, 2016.
29. Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed inclusion caches for mitigating llc side-channel attacks. In *ACM Design Automation Conference*, 2017.
30. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*. Springer, 1999.
31. 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 *IEEE International Symposium on High Performance Computer Architecture*, 2016.
32. Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014.
33. Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Symposium on Security and Privacy*, 2015.
34. 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 *Network and Distributed System Security Symposium*, 2017.
35. Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. Eddie: Em-based detection of deviations in program execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM*, 2017.
36. K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 2005.
37. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers Track at the RSA Conference*. Springer, 2006.
38. Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint archive*, 2005.
39. Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, 2016.
40. Colin Percival. Cache missing for fun and profit, 2005.
41. 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 *Conference on Computer and communications security*, 2009.
42. Adi Shamir. Protecting smart cards from passive power analysis with detached power supplies. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2000.
43. Jianli Shen, Guru Venkataramani, and Milos Prvulovic. Tradeoffs in fine-grained heap memory protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006.
44. Arvind Singh, Monodeep Kar, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. Integrated all-digital low-dropout regulator as a countermeasure to power attack in encryption engines. In *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 2016.

45. Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*. IEEE, 2002.
46. US Department of Defense. Trusted computer system evaluation criteria. *Department of Defense Standards*, 1983.
47. Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):5, 2009.
48. Guru Prasadh V Venkataramani. *Low-cost and efficient architectural support for correctness and performance debugging*. Georgia Institute of Technology, 2009.
49. Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *IEEE Design Automation Conference*, 2016.
50. Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference*, 2006.
51. Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*. ACM, 2007.
52. 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.
53. Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *ACM workshop on Cloud computing security workshop*, 2011.
54. Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *IEEE International Symposium on Computer Architecture*, 2017.
55. Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *IEEE International Symposium on Microarchitecture*, 2016.
56. Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *24th IEEE International Symposium on High-Performance Computer Architecture*, 2018.
57. Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 2017.
58. Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014, 2014.
59. Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.
60. Baki Yilmaz, Robert Callan, Milos Prvulovic, and A Zajic. Quantifying information leakage in a processor caused by the execution of instructions. In *MILCOM 2017 - 2017 IEEE Military Communications Conference*, pages 255–260, 10 2017.
61. Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 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*. ACM, 2016.