International Journal of Engineering, Management, Humanities and Social Sciences Paradigms (Volume 31, Special Issue of January 2019)
ISSN (Online): 2347-601X and Website: www.ijemhs.com

Protecting Against Cache-Based Side Channel Attacks with Secure Power structure Cache Replacement Policy (SHARP)

Ms. Swarnakanti Samantaray[1], Mr.Narottam Sahu[2]*

*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
swarnakanti@thenalanda.com, narottam@thenalanda.com*

ABSTRACT

In memory buffer side channel attacks, a spy who shares a cache with the target queries the locations of the cache to gather details about the target's access habits. For instance, in the technique known as "evict+reload," the spy repeatedly evicts and then reloads a probe address while monitoring if the victim has accessed the address in between the two actions. Although there are numerous solutions to stop these cache attacks, they all have drawbacks: either they degrade speed, demand programmer participation, or can only stop specific kinds of assaults. The following finding is made for an environment with an inclusive cache hierarchy: When the spy removes the probing address from the shared cache, the address will also be removed from the victim process's private cache, resulting in an inclusion victim. So, to prevent a process from creating inclusion victims in the caches of cores executing other processes, this study proposes to change the shared cache's line replacement method in order to eliminate cache attacks. By upholding this rule, the spy is prevented from removing the probe address from the shared cache and, as a result, is unable to spy on the victim's access habits. Our proposal is known as SHARP (Secure Hierarchy-Aware cache Replacement Policy). All current cross-core shared-cache threats are successfully defended against by SHARP, which requires no coding changes and only minor hardware adjustments. We use a cycle-level full-system simulator to implement SHARP. We demonstrate that it offers minimal average performance decrease and defends against real-world threats.

CCS CONCEPTS

• **Security and privacy**—**Side-channel analysis and countermeasures**; • **Computer systems organization**→ **Architectures**; **Multicore architectures**;

KEYWORDS

Security, Side channel, Cache, Cache replacement

1 INTRODUCTION

Side channel attacks [1, 17, 33, 40, 42] obtain private information from a system by observing its behavior, rather than by directly gaining access to private information. Such attacks are both popular and often highly effective. Due to their nature, they are hard to prevent with existing software techniques. Moreover, they are very difficult to detect, as they leave no trace within the system. Many instances of such attacks have been identified, which are able to discover security-sensitive information by monitoring features such as a program's cache use, power consumption, network activity, or timing behavior.

A very common side channel attack is the cache-based attack (e.g., [12, 15, 19, 28, 31, 36, 42, 50, 51]). Cache-based side channel attacks, or cache attacks for short, observe a program's cache behavior to infer details about the program's private information. A cache attack involves a *victim* and a *spy* process. The victim is the program of interest, which runs normally, unaware of the attack. The spy is a malicious program that probes key locations in the cache. With these probes, it extracts information about the cache behavior of the victim. In recent years, cache attacks have grown ever more sophisticated. The attack scope has expanded to include the mobile [28], desktop [36], and cloud domains [42, 51]. Also, new attacks monitor multiple facets of a victim, including keyboard presses [12] and web search history [15].

The most effective type of cache attack involves spy and victim processes executing on different cores, sharing the L2 or L3 level of an inclusive cache hierarchy. The reason for the attack's effectiveness is that it leverages widely-used commodity hardware, and is relatively easy to set up. For example, in evict+reload, the spy issues references that evict from the shared cache a *probe* address *A*. Later, the spy references *A*. Based on the latency of the reference to *A*, the spy knows whether the victim has accessed *A* since the eviction. These types of attacks can target fine-granularity addresses, and exploit a high-bandwidth, low-noise channel [19, 31, 50].

There have been many previous proposals to combat cross-core cache attacks (e.g., [16, 24, 29, 30, 34, 46, 47, 52]). However, these defensive techniques are deficient in one way or another. First, many of these proposals significantly hurt performance. Others require substantial programmer intervention. Finally, others cannot defend against all varieties of these cache attacks.

In the most widely-used environment, where each core has one or more levels of private caches, and shares an inclusive lower-level cache with all the other cores, we make the following observation: when the spy wants to evict the probe address from the shared cache, the address is also practically always in the private cache

of the core running the victim process. This is because of the tight timing requirements to mount a successful attack. Because caches are inclusive, the probe address also needs to be evicted from the private

cache of the victim process. Hence, the probe address becomes what is referred to as an *inclusion victim*.

To disable cache attacks, the main proposal of this paper is to alter a shared cache's replacement algorithm to prevent a process from creating inclusion victims in the caches of cores running other processes. By enforcing this rule, the spy cannot evict the probe address from the shared cache and, hence, cannot glimpse any information on the victim's access patterns. While minimizing inclusion victims has been proposed in the past to improve cache performance [22], ours is the first proposal that uses this idea for security purposes.

Cache attacks do not always use load instructions to force the eviction of a victim's probe addresses from the cache; sometimes they use an instruction called *clflush*. Hence, our proposal also involves a slightly modified *clflush* instruction to thwart these attacks.

We call our proposal SHARP (Secure Hierarchy-Aware cache Replacement Policy). SHARP is an efficient approach to defend against all existing cross-core shared-cache attacks. It requires minimal hardware modifications. It works for all existing applications without requiring any code modifications. Finally, it induces negligible average performance degradation.

To validate SHARP, we implement it in a cycle-level full-system simulator and test it against real-world attacks. SHARP effectively protects against these attacks. In addition, we run many workloads derived from SPEC and PARSEC applications on SHARP to evaluate SHARP's impact on performance. We find that SHARP introduces negligible average performance degradation.

The contributions of this paper are:

- The insight that, to effectively prevent cache attacks in an inclusive cache hierarchy, we can alter the shared cache replacement algorithm to prevent a process from inducing inclusion victims on other processes.
- The design of SHARP, which consists of a new cache line replacement scheme that prevents inclusion victims on other processes, and a slightly modified *clflush* instruction.
- A simulation-based evaluation of SHARP that shows that it is effective against real-world attacks, and induces negligible average performance degradation.

2 BACKGROUND

Cache-Based Side Channel Attacks

A basic cache-based side channel attack involves a victim process and a spy process sharing a cache. It usually consists of an offline phase and an online phase. In the offline phase, the attacker identifies *probe addresses*, which are addresses whose access patterns can leak information about the victim's program. The spy can deduce the value of private information, such as a private key or a user's keyboard input, just by observing the access patterns to probe addresses. Probe addresses are identified by analyzing the victim's program manually or with automatic tools [12, 15].

Algorithm 1 shows a simple Square-and-Multiply algorithm [9] from GnuPG version 1.4.13, which is vulnerable to side channel attacks. In the process of computing its output, the algorithm iterates over exponent bits from high to low. For each bit, it performs a *sqr* and *mod* operation. Then, if the exponent bit is "1", the algorithm performs a *mul* and a *mod* operation that are otherwise skipped. Effective probe addresses are the entry points of the *sqr* function in Line 3 (which tells that the iteration is executed) and of the *mul* function in Line 6 (which tells that the bit is "1"). By observing access pattern on probe addresses, the spy can recover all the bits in the exponent.

Algorithm 1: Square-and-Multiply exponentiation.

```
   Input  : base b, modulo m, exponent e = (e_{n-1}...e_0)_2
   Output : b^e mod m
1  r = 1
2  for i = n − 1 downto 0 do
3  |   r = sqr(r)
4  |   r = mod(r, m)
5  |   if e_i == 1 then
6  |   |   r = mul(r, b)
7  |   |   r = mod(r, m)
8  |   end
9  end
10 return r
```

The online phase usually consists of three steps: *Eviction*, *Wait*, and *Analysis*. In the first one, the spy evicts the victim's probe addresses from the cache. In the second one, the spy waits a designated amount of time to allow the victim to potentially access probe addresses. In the last one, the spy determines if the victim has accessed any probe addresses. These steps are repeated multiple times.

According to the approach used in the Eviction step, we classify attack strategies into conflict-based and flush-based (Table 1). In conflict-based strategies, the spy creates cache conflicts to evict cache lines containing probe addresses. Specifically, it accesses addresses that map to multiple cache lines in the same cache set as a probe address. Often, these addresses are called *conflict addresses*.

Strategies	Attacks
Conflict-based	prime+probe [40], evict+reload [12], evict+time [37], alias-driven attack [13], evict+prefetch [10]
Flush-based	flush+reload [50], flush+flush [11], invalidate+transfer [20], flush+prefetch [10]

Table 1: Classification of cache-based side channel attacks.

In flush-based strategies, the spy can access the probe addresses — e.g., when the probe addresses are in shared libraries. The attacker simply executes *clflush* instructions to evict the probe addresses from the cache [18]. clflush guarantees that the addresses are written back to memory and invalidated from the cache.

The *waiting interval* of the Wait step is carefully configured [50]. It should be precisely long enough for the victim to access a probe address exactly once before the Analysis step. If the interval is too long, the spy gets only one observation for multiple accesses to the probe address by the victim. If the interval is too short, the chances of overlapping the Eviction or Analysis step with the victim's probe address access increases. In both cases, accuracy of the attack decreases. Empirically, a waiting interval of 2,500–10,000 cycles works well.

In the Analysis step, the spy determines whether the probe address was accessed in the Wait step. There are several ways to accomplish this goal, including measuring the access time of either the probe or conflict addresses (prime+probe [19, 31, 40] and flush+reload [36, 50]), measuring the execution time of the victim program while evicting different addresses from the cache

(evict+time [35, 37]), or reading values in main memory to see if the writebacks of cache lines containing conflict addresses have occurred (alias-driven attack [13]).

There is one attack called the Cache Collision attack [4] that does not fit into either the conflict-based or flush-based categories. In this attack, the victim reuses data brought into the cache by the attacker. We do not know of any current programs that are susceptible to this type of attack. Hence, in this paper we do not address this type of attack.

Example of Cache Attack

Figure 1 shows the cache state in a simple example of the evict+reload [12] conflict-based attack. In this attack, the spy and victim processes share addresses — possibly because they use a shared library or due to page deduplication. The figure shows a timeline of the state of the six cache lines in a set of a six-way set-associative cache. At time *t0*, the victim loads a line with the probe address into the cache (black square). In the Eviction step (time *t1*), the spy accesses six conflicting addresses that bring six lines into the cache that fill the set (gray squares). In the Wait step (time *t2*), the spy idles and the victim accesses the probe address.

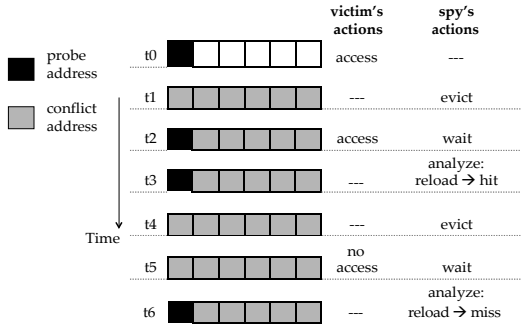


Figure 1: Evict+reload attack example.

In the Analysis step (time *t3*), the spy accesses the probe address and measures its access latency. If, as is shown in the figure, the victim accessed the probe address during the waiting interval, the spy will get a cache hit. The next three steps (times *t4*, *t5*, and *t6*) repeat the Eviction, Wait, and Analysis steps. This time, the victim does not access the probe address and the spy records a cache miss. The latency of the reload access is longer than before.

2.3 Inclusion Victims

In this paper, we focus on attacks leveraging a shared cache in an inclusive cache hierarchy. The victim process and the spy process(es) all run on different cores. Each core has one or more levels of private caches, and shares a lower level of cache (i.e., farther from the CPU) with all the other cores. The private higher-level caches must contain a subset of the lines held in the shared cache level [2]. In this environment, there are *inclusion victim* lines. These are lines that need to be evicted from a private cache because they are being displaced from the shared cache due to conflicts there.

Some authors have studied the impact of inclusion victims on performance (e.g., [7, 22]). In most designs, the cache replacement algorithm in the shared cache only uses information on shared cache hits and misses, and is oblivious of hits in the private caches. The TLA cache management policy [22] uses some hints that try to minimize the probability of selecting an inclusion victim that is being used in the private cache. In this paper, we consider the security implications of generating inclusion victims.

2.4 The *clflush* Instruction

The x86 *clflush* instruction invalidates a specific address from all levels of the cache hierarchy [18]. The invalidation is broadcasted throughout the cache coherence domain. If, at any cache, the line is dirty, it is written to memory before invalidation. In user space, *clflush* is used to handle memory inconsistencies such as in memory-mapped I/O and self-modifying codes. In kernel space, *clflush* is used for memory management, e.g., to flush from the caches all the lines belonging to a page that is being swapped out.

In Intel processors, a user thread can use *clflush* to flush readable and executable pages. This enables cache-based side channel attacks, as a spy can flush pages that it shares with a victim — e.g., a shared library. This attack has been reported for Intel [50], AMD [20], and ARM [28] processors.

3 ATTACK ANALYSIS

In this section, we analyze the two types of cache attacks based on the Eviction step.

Conflict-Based Attacks

We argue that all successful conflict-based attacks share two traits: (1) they generate inclusion victims in the private cache of the core running the victim thread, and (2) they exploit modern cache line replacement policies that do not properly defend against malicious creation of inclusion victims.

Consider the first trait. Existing conflict-based attacks generate inclusion victims in the private cache of the victim process' core. This is because the duration of an attack cycle between consecutive Eviction steps is very short — on the order of several thousand cycles. Attacks use such short cycles to reduce the noise in the Analysis step. As a result, if the victim accesses the probe addresses during the Wait step, then such addresses will typically remain in the victim's private cache by the next Evict step. Hence, when the spy performs the Evict step, it generates inclusion victims in the private cache of the victim's core.

Further, we note that there are no reported conflict-based attacks that work on *exclusive* cache hierarchies. We checked all existing attacks and found no exceptions. In processors such as the ARM Cortex-A53, which have an inclusive instruction cache and an exclusive data cache, existing attacks only target the instruction cache [28].

The second trait concerns the fact that deployed cache line replacement algorithms and deployed algorithms for inserting referenced lines in the replacement priority list, do not take into consideration the possible creation of inclusion victims. This makes commercial systems vulnerable to conflict-based attacks.

Recent proposals (e.g., [23, 32, 41, 48]) take into account the requesting core ID when deciding what line in the set to replace, or what priority in the replacement list to assign to the referenced line. However, they do it to improve resource allocation or to enhance performance, and do not try to eliminate inclusion victims. Only

the TLA cache management proposal [22] uses some hints that try to minimize the probability of creating inclusion victims. However, since TLA is focused on performance, it does not guarantee the elimination of inclusion victims and, hence, cannot provide security guarantees.

3.2 Flush-Based Attacks

Flush-based attacks rely on the *clflush* instruction to evict a victim's probe addresses from the cache. Entirely disabling the use of such instruction is impractical, however, due to both legacy issues and valid use cases. However, we make a key observation about the legitimate uses of *clflush*: in user mode, *clflush* is only really needed in uses that update memory locations. Specifically, it is needed to handle the case when the value of a location in caches is more up-to-date than the value of the same location in main memory. In such cases, *clflush* brings the memory to the right state.

We argue that there is no need to use *clflush* in user mode for pages that are read-only or executable, such as those that contain shared library code. Allowing the use of *clflush* in these pages only makes the system vulnerable to flush-based attacks.

4 SHARP DESIGN

We propose a novel approach to defend against cache-based side channel attacks that is highly effective, induces negligible average performance degradation, and requires minimal hardware modifications and no code modifications. The approach, called Secure Hierarchy-Aware cache Replacement Policy (SHARP) is composed of a new cache replacement scheme to protect against conflict-based cache attacks, and a slightly modified *clflush* instruction to protect against flush-based cache attacks. In the following, we discuss SHARP's two components, and then give some examples of defenses.

Protecting Against Conflict-Based Attacks

To protect against conflict-based attacks, SHARP's main idea is to alter a shared cache's replacement algorithm to minimize the number of inclusion victims that a process induces on other processes. The goal is to prevent a spy process from replacing shared-cache lines from the victim process that would create inclusion victims in the private caches of the victim process' core. The result is that the spy cannot create a conflict-based cache attack.

Assume that a requesting process *R* (potentially a spy) wants to load a line into a set of the shared cache that is full. The hardware has to find a victim line to be evicted. The high level operation of the SHARP replacement algorithm is shown in Figure 2. It has three steps. In Step 1, SHARP considers each line of the set at a time (①), in the order based on its replacement priority. For each line, it checks if the line is in any private cache (②)–(③). As soon as a line is found that is not in any private cache, it is used as the replacement victim (⑧). Victimizing this line will not create any inclusion victim. If no such line is found, the algorithm goes to Step 2.

In Step 2, SHARP considers again each line of the set at a time (①), in the order based on its replacement priority. For each line, it checks if the line is present only in the private cache of *R* (④)–(⑤). As soon as one such line is found, it is used as the replacement victim (⑧). Evicting this line will at worst create an inclusion victim

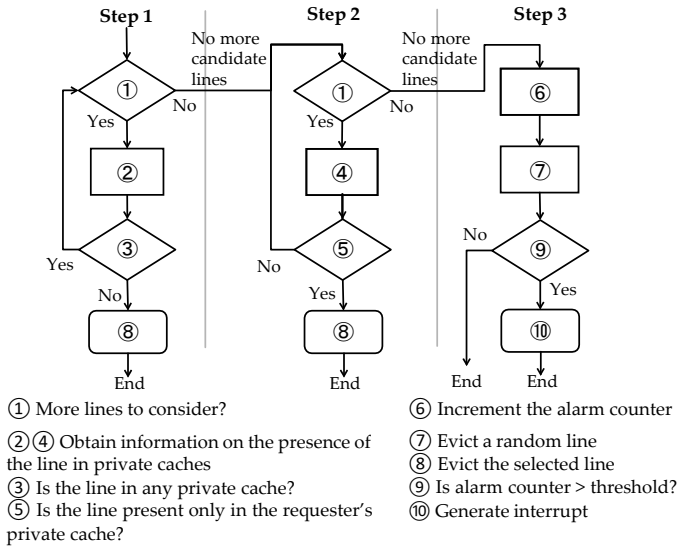


Figure 2: SHARP replacement algorithm.

in *R*. No other process will be affected. If no such line is found, the algorithm goes to Step 3.

In Step 3, SHARP increments a per-core local alarm event counter (⑥) and selects a random line as the replacement victim (⑦). In this case, SHARP may create a replacement victim in a process that is being attacked. For this reason, when the alarm event counter of any core reaches a threshold (⑨), a processor interrupt is triggered (⑩). The operating system is thus notified that there is suspicious activity currently in the system. Any relatively low value of the threshold suffices, as a real spy will produce many alarms to be able to obtain any substantial information.

From this discussion, we see that SHARP has very general applicability, requires no code modification (unlike [24, 29]), and does not partition the cache among processes (unlike [8, 46]). It allows multiple processes to dynamically share the entire shared cache, while transparently protecting against conflict-based side-channel attacks.

SHARP requires hardware modifications to implement its replacement policy. Specifically, SHARP must be aware of what lines within the shared cache are present in the private caches. Such information is needed in operations ② and ④ of Figure 2. In the subsequent subsections, we present three different ways of procuring this information.

Using Core Valid Bits. In SHARP, each line in the shared cache is augmented with a bitmap with as many bits as cores. The bit for core *i* is set if the line is present in core *i*'s private cache.These are the *Presence* bits used in directory-based protocols [26].For example, in Intel, they are used in multicores since the Nehalemmicroarchitecture [45], where they are called Core Valid Bits (CVB).In this first design, SHARP simply leverages these bits to deter- mine the information needed in operations ② and ④ of Figure 2. Note, however, that these bits carry *conservative* information. Thismeans that if bit *i* is set, core *i may* have the line in its private cache,while if bit *i* is clear, core *i* is guaranteed not to have the line initsprivate cache. Such conservatism stems from silent evictions of non-dirty lines from private caches; these evictions do not update

the CVB bits. As a result, the CVB bits will still show the evicting core as having a copy of the line in its private cache. Overall, this conservatism will cause Steps 2 and 3 in Figure 2 to be executed more often than in a precise scheme. However, correctness is not compromised.

Using Queries. A shortcoming of the previous design is that it often ends-up assuming that shared cache lines are present inmore private caches than they really are. As a result, a process may unnecessarily fail to find a victim in Step 1 and end-up victimizing its own lines in Step 2, or unnecessarily fail to find a victim in Step 2 and end-up raising an exception.

To solve this problem, this second SHARP design extends the first one with core queries. Specifically, Step 1 in Figure 2 proceeds as usual; it often finds a victim. In Step 2, however, as each line is examined in order based on its replacement priority, the SHARP hardware queries the private caches of the cores that have the CVB bit set for the line, to confirm that the bit is indeed up to date.

The CVBs of the line are refreshed with the outcome of the query. With the refresh, the CVBs may show that, in reality, the line is in no private cache, or only in the private cache of the requesting processor. In this case, the line is victimized and the replacement algorithm terminates; there is no need to examine the other lines.

As a line's CVBs are refreshed, the line is considered to be accessed, and is placed in its corresponding position in the replacement priority. This is done to ensure that such a line is not considered and refreshed again in the very near future.

This design generally delivers higher performance than the first one. The reason is that the queries of private caches refresh the CVB bits, obtaining a more accurate state of the system for the future. Note that the queries are typically hidden under the latency of the memory access that triggered them in the first place.

Similar query-based schemes have been proposed in the past. They have been used to reduce inclusion victims with the aim of improving performance [22].

Using Core Valid Bits and Queries. A limitation of the previous design is that it does not scale well. For multicores with many cores, the latency of the queries may not be hidden by the cache miss latency. Moreover, the traffic induced by the queries may slow down other network requests. Consequently, we present a third SHARP design that reduces the number of queries.

Specifically, in Step 2 of Figure 2, SHARP only sends queries for the first *N* lines examined. For the remaining lines in the set, SHARP uses the CVBs as in the first scheme. As usual, Step 2 finishes as soon a victim line is found. There are no other changes relative to the second design.

Protecting Against Flush-Based Attacks

As argued in Section 3.2, there is no need to use *clflush* in user mode for pages that are read-only or executable. Hence, in user mode, SHARP only allows *clflush* to be performed on pages with write permissions.

With this restriction, sharing library code between processes and supporting page deduplication do not open up vulnerabilities to flush-based attacks. Specifically, if spy and victim process share library code and the spy invokes *clflush*, the spy will suffer an exception because the addresses are execution-only. Hence, the victim process will not suffer inclusion victims in its cache. Similarly, if spy and victim share a deduplicated page and the spy invokes *clflush*, since the page is marked copy-on-write, the OS will trigger a page copy. All subsequent *clflush*es by the spy will operate on the spy's own copy. As before, the attack will be ineffective.

SHARP allows *clflush* to execute unmodified in kernel mode, as it is necessary for memory management.

Examples of Defenses

We give two examples to show how SHARP can successfully defend against conflict-based attacks. In the examples, victim and spy share a probe address, and the spy uses the evict+reload attack (Section 2.2). Private caches are 4-way set-associative, and the shared one is 8-way. We consider first a single-threaded spy and then a multi-threaded spy.

Attack Using a Single-Threaded Spy. Figure 3 shows the cache hierarchy, where the victim runs on Core *0* and the spy on Core *1*. In Figure 3(a), the victim has loaded the probe address, and the spy has loaded four lines with conflict addresses. In Figure 3(b), the spy loads four more lines with conflict addresses. Since the corresponding set in the shared cache only had three empty lines, one of the existing lines has to be evicted. SHARP forces the eviction of one of the old lines of the spy — not the one with the probe address.

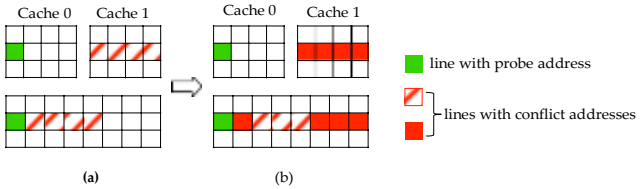


Figure 3: SHARP defending against a single-threaded attack.

Attack Using a Multi-Threaded Spy. Figure 4 shows a cache hierarchy with four private caches, where the victim runs on Core *0* and three spy threads run on Cores *1*, *2*, and *3*. In the figure, the victim has loaded the probe address, and the spy threads tried to evict it. Spy *1* loaded four conflicting lines, and Spy *2* three conflicting lines. If Spy *2* now loads another conflicting line, it will only victimize one of its own lines. The same is true for Spy *1*. SHARP is protecting the probe address.

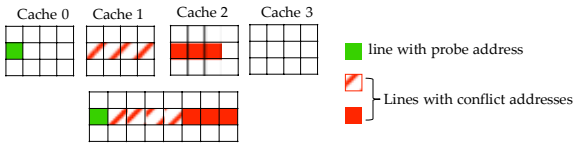


Figure 4: SHARP defending against a multi-threaded attack.

To have a chance to evict the probe address, a third Spy thread (Spy *3*) needs to load a conflicting line. However, such access will only evict a *random* line in the set, and it will increment the alarm counter. Additional lines loaded by Spy *3* will only victimize Spy *3*'s line. To be able to cause multiple random replacements, the Spy threads must be highly coordinated to ensure that, for each round of attack, at least one Spy thread does not occupy any line in the corresponding shared cache set.

SHARP makes it very difficult for a multi-threaded spy to get victim's information through random replacements and, at the same time, not be detected. Specifically, the timing of the requests from many spy threads needs to be finely coordinated, to ensure that, at every round of attack, at least one spy thread does not occupy any line in the shared cache set. Second, spies need to handle and tolerate the unavoidably high noise, as they try to distinguish between misses caused by other spies and by the victim. Third, spies suffer from a high risk of being detected, as every single random eviction will increment the alarm counter. For these reasons, SHARP is highly effective against multi-threaded attacks.

5 DISCUSSION

Handling Related Attacks

To put SHARP in perspective, we examine how it handles two additional situations.

Initial Access Vulnerability. There is one type of cache-based side channel attack where the spy simply wants to know whether the program execution loaded a given probe address. For example, the spy repeatedly loads the probe address and evicts it, while timing the latency of the load. After the victim loads the probe address, the load by the spy is fast because it hits in the cache. We call this vulnerability the Initial Access vulnerability. The SHARP designs that we have presented are not able to thwart it. This is because, in these attacks, the spy does not need to evict the lines that have been accessed by the victim.

An attack targeting this general vulnerability has been implemented and called the Cache Collision attack [4]. The Initial Access vulnerability can be thwarted by adopting the preloading techniques proposed in [25, 29]. They involve loading into the cache all the security-sensitive addresses, so that the spy cannot know which address the victim really needs to access. Such loading can be done with plain loads or with prefetches. SHARP can use such techniques to eliminate the Initial Access vulnerability.

Exploiting Private Cache Conflicts. Since the private cache used by the victim process has limited size and associativity, it is possible that a probe address gets evicted due to lack of space. After this happens, the SHARP designs that use queries may detect that the line is no longer in the private cache and pick it as a replacement victim in the shared cache. Hence, it is theoretically possible for a spy to exploit the capacity and conflict misses in the private cache of the victim to bypass SHARP's protection and mount a cache-based side channel attack.

In practice, mounting such an attack is very difficult. The reason is that the spy has no control on the way that lines evict each other in the private cache of the victim. In addition, the spy can at best find out when a line with the probe address was *evicted*, but not when the probe address was last *accessed* before the eviction. If knowing when a line was evicted was enough to mount an attack, there would probably be proposals of conflict-based attacks on non-inclusive caches, which we have not seen yet.

Hardware Needs & Performance Impact

SHARP has modest hardware requirements. As per Section 4.1, itneeds presence bits in the shared cache (i.e., the CVBs) and cache queries. The CVBs are already present in Intel multicores to support cache coherence, and can be reused. In directory-based multiprocessors that use limited-pointer directories, SHARP can be modified to also reuse the hardware.

To support queries, SHARP adds two additional messages to the coherence protocol, namely a query request and a query reply. The cache controller needs corresponding states to handle the two new messages. Such modification has also been used by Intel researchers to improve cache management [22].

SHARP induces negligible average performance degradation. This is because, unlike schemes that explicitly partition the shared cache among threads (e.g., [8, 46]), SHARP allows multiple threads to dynamically share a cache flexibly. In addition, the queries are performed in the background, in parallel to servicing a cache miss from memory. In practice, the great majority of the replacements that use queries are satisfied with the first query.

It can be argued that, in some cases, SHARP will cause a thread to be stuck with a single way of the shared cache, and repeatedly victimize its own private cache lines. This may be the case with the victim thread in Figure 4. While such case is possible, it is rare. Recall that the lines in a set in the private cache can map to multiple sets in the bigger, shared cache (say around 8 or so). The pathological case happens when many referenced lines across all cores map to the same set in both private and shared caches, and the shared-cache associativity is not enough. While possible, this case is rare, only temporary, and only affects the relevant cache set.

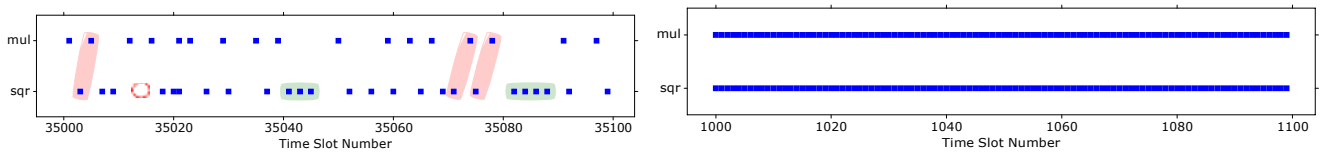
6 EXPERIMENTAL SETUP

To evaluate SHARP, we modify the MARSS [38] cycle-level full-system simulator. We model a multicore with 2, 4, 8, or 16 cores. Each core is 4-issue and out-of-order, and has private L1 and L2 caches. All cores share a multi-banked L3 cache, where the attacks take place. The chip architecture is similar to the Intel Nehalem [45]. The simulator runs a 64-bit version of Ubuntu 10.4. Table 2 shows the parameters of the simulated architecture. Unless otherwise indicated, caches use the pseudo-LRU replacement policy.

Parameter	Value
Multicore	2–16 cores at 2.5GHz
Core	4-issue, out-of-order, 128-entry ROB
Private L1 I-Cache/D-Cache	32KB each, 64B line, 4-way, Access latency: 1 cycle
Private L2 Cache	256KB, 64B line, 8-way, Access latency: 5 cycles after L1
Query from L3 to L2	3 cycle network latency each way
Shared L3 Cache	2MB bank per core, 64B line, 16 way, Access latency: 10 cycles after L2
Coherence Protocol	MESI
DRAM	Access latency: 50ns after L3

Table 2: Parameters of the simulated architecture.

We evaluate the 7 configurations of Table 3, which have different L3 line replacement policies: *baseline* uses the conventional pseudo-LRU policy; *cvb*, *query*, and *SHARPX* use the SHARP designs of Sections 4.1.1, 4.1.2, and 4.1.3, respectively. *SHARPX* includes 4 configurations (*SHARP[1-4]*), which vary based on the maximum number *N* of queries emitted. Recall that, for a given set, a query needs to be fully completed before a second one can be initiated.



(a) Using *baseline*. **(b) Using *SHARP4*.**

Figure 5: Cache hits on the probe addresses by the spy process in GnuPG.

Config.	Line Replacement Policy in L3
baseline	Pseudo-LRU replacement.
cvb	Section 4.1.1 design: Use CVBs in both Step 1 and 2.
query	Section 4.1.2 design: CVBs in Step 1 & queries in Step 2.
SHARP*X*	Section 4.1.3 design: CVBs in Step 1. In Step 2, limit the max number of queries to *X*, where *X* = 1, 2, 3, or 4.

Table 3: Simulated L3 line replacement configurations.

7 EVALUATION

Defense Analysis

In this section, we evaluate the effectiveness of SHARP against two real cache-based side channel attacks. We implement the attacks using *evict+reload*, which consists of the spy evicting the probe address and then accessing it. If, in between, the victim has accessed the probe address, the spy's reload access hits in the cache; otherwise, the spy's reload access misses.

To achieve page sharing between spy and victim, the spy *mmaps* the victim's executable file or shared library into the spy's virtual address space. To select conflict addresses, the spy first accesses the system files (i.e., */proc/$pid/pagemap* on Linux) to identify the physical addresses of probe addresses. It then selects 16 addresses that map to the same L3 set as each of the probe addresses to form an eviction set. When performing the evict operation, the spy accesses the 16 addresses twice to ensure that the probe address is replaced. When doing the reload operation, the spy accesses the probe address and measures the access time using *rdtsc*. Based on the time measured, it determines if it is an L3 hit. If so, it knows that the address has been accessed by the victim.

We measure the L3 hit and miss time for our architecture. We find that, on average, an L3 hit takes 48 cycles, and an L3 miss 170 cycles. Hence, we use 100 cycles as a threshold to decide if it is a hit or a miss.

In the following attacks, we launch the victim process on one core and the spy on another. We show results for the *SHARP4* configuration; the other configurations work equally well in terms of defense.

Defending against Attacks on GnuPG. Our first attack example targets GnuPG, a free implementation of the OpenPGP standard. The modular exponentiation in GnuPG version 1.4.13 uses a simple Square-and-Multiply algorithm [9]. The calculation is shown in Algorithm 1, and is described in Section 2.1.

In this attack, the spy divides the time into fixed time slots of 5,000 cycles, and monitors for 10,000 time slots. In each time slot, it evicts and reloads two probe addresses: the entry points of the *sqr* and *mul* functions (Algorithm 1). Figure 5 shows the result of 100 time slots for the *baseline* and *SHARP4* configurations. In the figure, a dot represents a cache hit in the reload of the corresponding probe address.

In *baseline*, when a hit occurs, it is because the victim has accessed the probe address during the interval between evict and reload. In Figure 5(a), we see the pattern of victim accesses to *sqr* and *mul*. When a *sqr* hit is followed by a *mul* hit, the value of the bit in the exponent vector is 1. The figure highlights three examples of this case with a shaded vertical pattern. When a *sqr* hit is not immediately followed by a *mul* hit, the value of the bit in the exponent vector is 0. The figure highlights two examples of multiple *sqr* hits in a row with a shaded horizontal pattern. In some cases, the timing is such that the evict follows the victim's access. In that case, the reload may miss an access. The figure highlights one such example with a circle. Even with some such misses, the spy can successfully attain most of the bits in the exponent vector, which is enough for the attack to succeed.

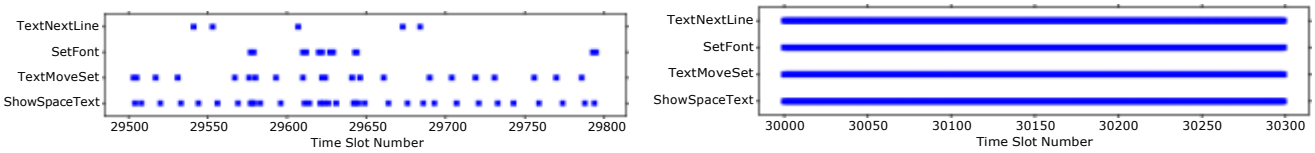
Consider now *SHARP4*. The first time that the victim calls *sqr* and *mul*, the probe addresses are loaded into the shared cache and into the victim's private cache. Then, *SHARP4* prevents the spy from evicting the probe addresses from the shared cache. As a result, every single reload by the spy will hit in the cache. The result, as shown in Figure 5(b), is that the spy in unable to glean any information from the attack.

Defending against Attacks on Poppler. Our second attack example targets Poppler, a PDF rendering library that is widely used in software such as Evince and LibreOffice. We select *pdftops* as the victim program. Pdftops converts a PDF file into a PostScript file. The execution of pdftops is very dependent on the input PDF file. Hornby et al. [15] design an attack that probes the entry points of four functions in pdftops that allow the attacker to distinguish different input PDF files with high fidelity:

- Gfx::opShowSpaceText(Object*, int)
- Gfx::opTextMoveSet(Object*, int)
- Gfx::opSetFont(Object*, int)
- Gfx::opTextNextLine(Object*, int)

Their attack consists of three stages: training, attack, and identification. In the training stage, they collect the probing address sequence for different input PDF files multiple times, to obtain the unique signature for each file. In the attack stage, the spy records the probe address sequence of the victim as it executes pdftops with an input PDF file. In the identification stage, the spy computes the Levenshtein distance[1] [27] between the victim's probe sequence and all of the training probe sequences. The training sequence with the smallest Levenshtein distance to the victim's is assumed to correspond to

[1] Levenshtein distance is the smallest number of basic edits (single-character insertions, deletions, and replacements) needed to bring one string to the other.



(a) Using *baseline*. **(b) Using *SHARP4*.**

Figure 6: Cache hits on the probe addresses by the spy process in Poppler.

the input file that the victim used. By using this approach, they can reliably identify 127 PDF files on a real machine.

In our attack, the spy monitors the entry points of those functions using evict+reload. The spy divides time into fixed time slots of 10,000 cycles each. During each time slot, it evicts and reloads the 4 probe addresses. Figure 6 shows the reload hits in 300 slots. In *baseline* (Figure 6(a)), we can clearly monitor the execution of the 4 functions over time. Since each input PDF file results in a different execution order and frequency for these functions, the pattern can be used as a signature to uniquely identify the input file. In *SHARP4* (Figure 6(b)), the reloads always hit, which makes it impossible to distinguish different input files by their probed cache behavior.

Performance Impact

In this section, we evaluate the performance impact of SHARP using both mixes of single-threaded applications (SPECInt2006 and SPECFP2006 [14]), and multi-threaded applications (PARSEC [3]).

Single-Threaded Application Mixes. We start by evaluat-ing mixes of 2 SPEC applications at a time, using 2 cores with a total L3 size of 4MB. To choose the mixes, we use the same approach as Jaleel et al. [22]. We group the applications into three categories according to their cache behavior [21]: SW (small working set), MW (medium working set), and LW (large working set). SW applications, such as *sjeng*, *povray*, *h264ref*, *dealII*, and *perlbench*, fit into the L2 private caches. MW applications, such as *astar*, *bzip2*, *calculix*, and *gobmk*, fit into the L3. Finally, LW applications, such as *mcf* and *libquantum*, have a footprint larger than the L3. We choose a set of mixes similar to Jaleel et al. [22], which the authors suggest are representative of all mixes of the SPEC applications.

We use the *reference* input size for all applications. In each experiment, we start two applications and pin them to separate cores. We skip the first 10 billion instructions in each application; then, we simulate for 1 billion cycles. We measure statistics for each application.

Figure 7 shows the IPC of each application in each of the 9 mixes considered. For a given application, the figure shows bars for *baseline*, *cvb*, *query*, and *SHARP[1,4]*, which are all normalized to *baseline*. In the figure, higher bars are better. Figure 8 shows the L3 misses per kilo instruction (MPKI). It is organized as Figure 7 and, as before, bars are normalized to *baseline*.

From Figure 7, we see that the performance of the applications in *query* and *SHARP[1,4]* is generally similar to that in *baseline*. Hence, SHARP has a negligible performance impact. In addition, the reason why *query* and *SHARP[1,4]* all behave similarly is that, in the large majority of cases, the first query successfully identifies a line to evict.

The figure also shows that *cvb* is not competitive. For applications such as *astar* in MIX3 and *perlbench* in MIX6, *cvb* reduces the IPC substantially. The reason is that the imprecision in the *CVB*s causes suboptimal line replacement. In particular, threads end up victimizing themselves. In some of these applications, the relative MPKI increases substantially (Figure 8). Note, however, that these are normalized MPKI values. In these SW and MW applications, while the bar changes may seem large, they correspond to small changes in absolute MPKI. For example, the MPKI of *dealII* in MIX8 is 0.025 in *baseline*, and it increases by 15x to a still modest value of 0.392 in *cvb*.

Some of the mixes expose the effects of SHARP more than others. For example, the mixes that contain an SW and an LW application are especially revealing (e.g., MIX0). In these workloads, *SHARP[1,4]* helps the SW application retain some L3 ways for itself — rather that allowing the LW application to hog all the L3 ways as in *baseline*. As a result, the SW application increases its IPC and reduces its MPKI (*povray* in MIX0). At the same time, the LW application does not change its IPC or MPKI much (*mcf* in MIX0). The reason is that the LW application already had a large MPKI, and the small increase in misses has little effect.

Multi-Threaded Applications. We now evaluate PARSEC applications running on 4 cores with a total L3 size of 8MB. The applications' input size is *simmedium*, except for *facesim*, which uses *simsmall*. The applications run for the whole region of interest, with at least 4 threads, and with threads pinned to cores. For these applications, we report total execution time, rather than average IPC, as the performance metric. This is because these applications have synchronization and, therefore, may execute spinloops.

Figures 9 and 10 show the normalized execution time and L3 MPKI, respectively, for each application and for the average. These figures are organized as Figures 7 and 8. In Figure 9, lower is better.

In this environment, threads share data and, therefore, a given cache line can be in multiple private caches. As a result, a thread may be unable to evict data that it has brought into the shared cache because another thread has reused the data and is caching it in its own private cache.

This property may have good or bad effects on the MPKI (Figure 10). For example, in *ferret*, the inability of a thread to evict shared data causes cache thrashing and a higher MPKI. On the other hand, in *fluidanimate*, the MPKI decreases slightly.

If we look at the execution time (Figure 9), we see that *query* and *SHARP[1,4]* have similar performance as *baseline*. The only difference is a modest slowdown of 6% in *canneal*. This application has a large working set, and the new replacement policy ends up creating more misses which, in turn, slow down the application. Overall, however, the impact of *query* and *SHARP[1,4]* on the execution time

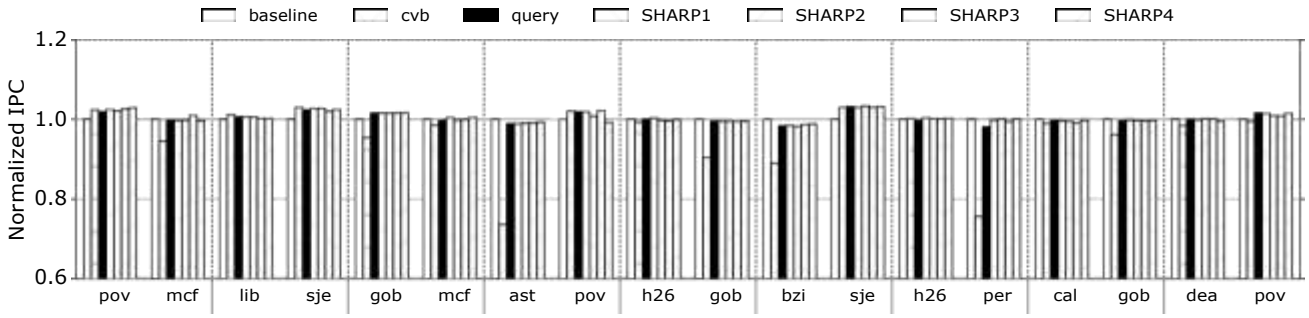


Figure 7: Normalized IPC of SPEC application mixes with different replacement policies on 2 cores.

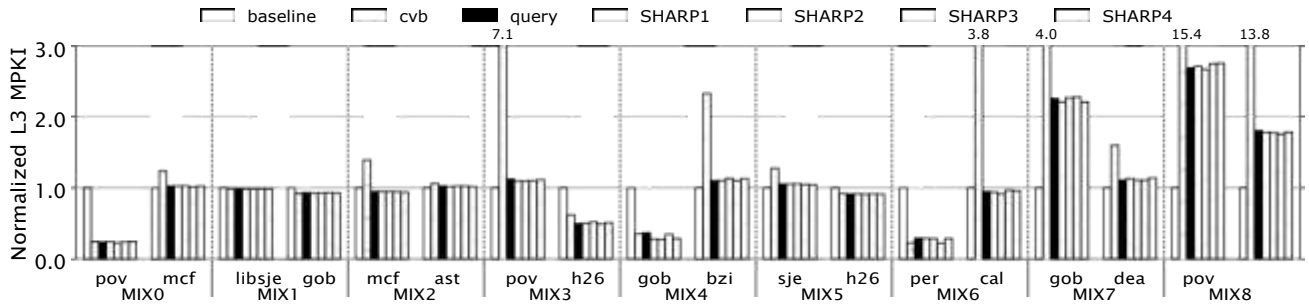


Figure 8: Normalized L3 MPKI of SPEC application mixes with different replacement policies on 2 cores.

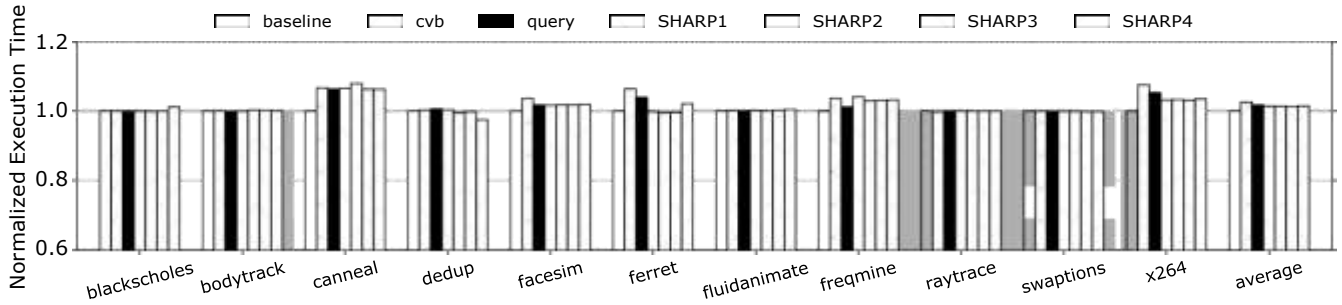


Figure 9: Normalized Execution Time of PARSEC applications with different replacement policies on 4 cores.

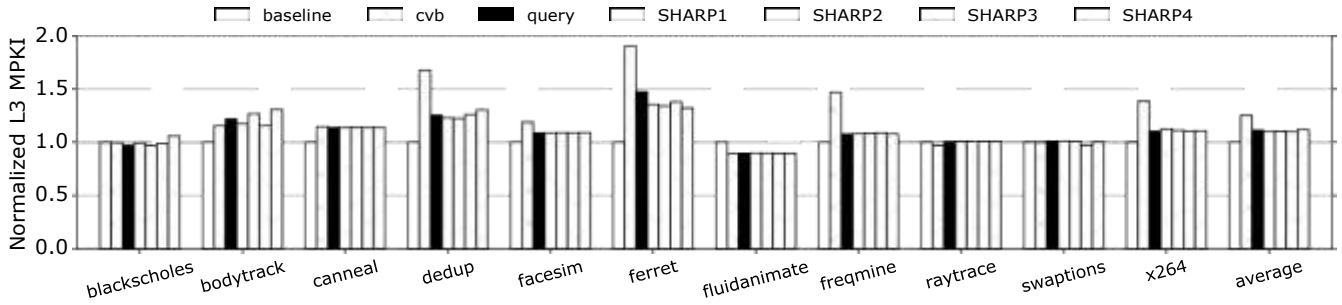
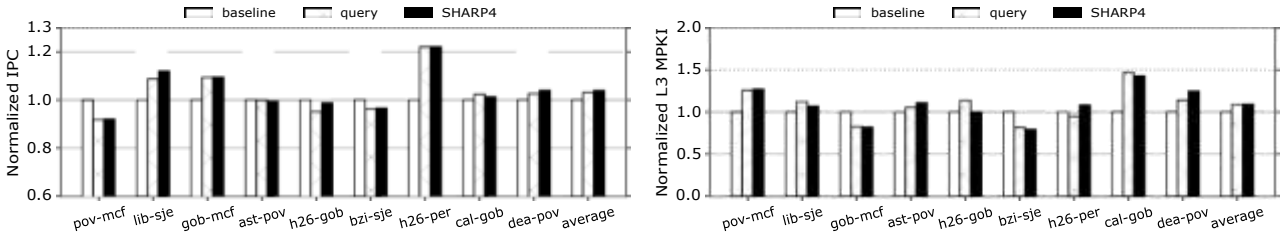
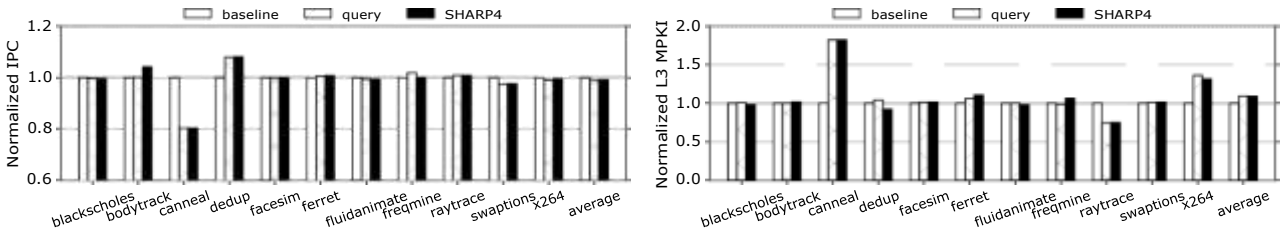


Figure 10: Normalized L3 MPKI of PARSEC applications with different replacement policies on 4 cores.



(a) Normalized IPC. **(b) Normalized L3 MPKI.**

Figure 11: Normalized IPC and L3 MPKI of SPEC application mixes with different replacement policies on 8 cores.



(a) Normalized IPC. **(b) Normalized L3 MPKI.**

Figure 12: Normalized IPC and L3 MPKI of PARSEC applications with different replacement policies on 16 cores.

is negligible. *cvb* is slightly worse. The reason is that the imprecision of the CVBs causes a higher MPKI and a slightly slower execution.

Scalability. To assess the scalability of SHARP, we run SPEC application mixes and PARSEC applications with larger core counts. Specifically, the SPEC application mixes run on 8 cores with a total L3 size of 16MB. The pairs of applications in each mix are the same as in Section 7.2.1, except that we run 4 instances of each of the two applications. We use the *reference* input sets, and collect statistics for 1 billion cycles after all applications have reached the region of interest.

Figures 11(a) and 11(b) show the normalized IPC and L3 MPKI, respectively, for the SPEC mixes. For each mix, we show the average of the 8 applications and, due to space limitations, only the *baseline*, *query*, and *SHARP*4 configurations. The figures also have bars for the average of all mixes.

The changes in the line replacement algorithm affect what data is kept in the L3 caches. Our SHARP designs try to avoid creating replacement victims in the private caches of other cores. As shown in Figure 11(a), sometimes this causes the average IPC of the applications to decrease (*povray+mcf*) and sometimes to increase (*h264ref+perlbench*). However, on average for all the mixes, the IPC under *SHARP*4 and *query* is about 3–4% higher than *baseline*. Overall, therefore, we conclude that SHARP has a negligible average performance impact.

As shown Figure 11(b), the L3 MPKI also goes up and down depending on the application mix, but the average impact is small. Note that, for a given application mix, it is possible that *SHARP*4 increases both the average IPC and the average MPKI. For example, in *libquantum+sjeng*, the average IPC goes up because *sjeng*'s IPC increases more than *libquantum*'s decreases. At the same time, *libquantum*'s average MPKI goes up more than *sjeng*'s goes down.

We also run PARSEC applications on 16 cores with a total L3 size of 32MB. The applications' input size is *simlarge*. Given the long simulation time, we report statistics for the first 1 billion cycles in the region of interest. Consequently, we report performance as IPC rather than execution time. Figures 12(a) and 12(b) show the normalized IPC and L3 MPKI, respectively. The figures are organized as Figures 11(a) and 11(b).

In Figure 12(a), we see that most of the applications have similar IPCs for *baseline*, *query*, and *SHARP*4. The one application where SHARP hurts IPC is *canneal*. This application has a very large working set. In *baseline*, the L3 MPKI of *canneal* is 5.18, compared to an MPKI lower than 1 for the other applications. In addition, there is fine-grained synchronization between threads. Data that is brought into the cache by one thread is used by other threads. This causes SHARP to avoid evicting such lines. The result is higher MPKI (Figure 12(b)) and lower IPC. This behavior is consistent with the one displayed for 4-core runs (Figure 9). On average across all the applications, however, *query* and *SHARP*4 have negligible impact on IPC and (to a lesser extent) on MPKI.

Alarm Analysis

Recall that when SHARP needs to evict a line from the shared cache, it looks for a victim that is not in any private cache or only in the private cache of the requester. If SHARP cannot find such a victim, it increments an alarm counter in the requesting core and evicts a random line in the set. For normal applications, the number of alarm increments is low. In an attack, however, the number of

alarm increments will be very high. To see why, consider a spy that launches multiple threads to attack a victim by generating accesses to conflict addresses. Empirically, we find that, in a successful side-channel attack, the time between consecutive evictions is about 2,500-10,000 cycles. So, the attackers will need to cause an alarm every 10,000 cycles. In practice, since the operation evicts a random line in the set, for a 16-way associative cache, they will need 16 times more alarms to evict the victim line. Let us assume that we have 16 attacker threads and the worst case that each attacker creates an equal number of alarms. We then have that each attacker thread will increment its counter at least 100,000 times in 1 billion cycles.

To see how this number compares to the alarm count in a benign execution, Table 4 shows the maximum alarm count observed per 1 billion cycles in any core while running benign workloads. Specifically, we run the 8-threaded SPEC mixes and 16-threaded PARSEC applications of Section 7.2.3, and try the *cvb*, *query*, *SHARP*1, *SHARP*2, *SHARP*3, and *SHARP*4 configurations. The last row of the table shows the maximum number across applications for each configuration.

Appls.	*cvb*	*query*	*SHARP*1	*SHARP*2	*SHARP*3	*SHARP*4
pov-mcf	285238	89	7285	171	121	84
lib-sje	1618715	1460	4747	2033	1820	1403
gob-mcf	549976	687	10001	1160	1426	1045
ast-pov	22701	19	1774	137	36	7
h26-gob	511	0	16	2	0	0
bzi-sje	38669	7	177	9	7	2
h26-per	60536	1	974	184	6	2
cal-gob	132169	0	37	25	33	1
dea-pov	3	0	0	1	0	0
blackscholes	0	0	0	0	0	0
bodytrack	0	0	0	0	0	0
canneal	153165	37	1192	39	43	37
dedup	145079	13	410	32	18	36
facesim	46409	12	97	32	16	1
ferret	91443	6	2097	102	15	9
fluidanimate	25643	2	556	144	26	3
freqmine	0	0	0	0	0	0
raytrace	10013	1	85	5	1	1
swaptions	0	0	0	0	0	0
x264	35897	2	423	10	5	14
MAX	1618715	1460	10001	2033	1820	1403

Table 4: Alarms per 1 billion cycles in benign workloads.

From the table, we see that *cvb* can trigger many alarms in multiple workloads. These high numbers are due to the lack of precision of this replacement policy, where the CVBs can be stale. Consequently, *cvb* is not recommended.

With the other policies, the number of alarms decreases substantially. This is because the policies refresh CVBs. For most workloads, the number of alarms is less than 100 per 1 billion cycles. A few SPEC mixes, such as *libquantum+sjeng* and *gobmk+mcf* reach several thousand alarms. These alarms occur because four instances of memory-intensive applications (*libquantum* and *mcf*) cause contention on many cache sets, and force evictions of cache lines belonging to their companion computation-intensive applications (*sjeng* and *gobmk*). SHARP is unable to find a line that only exists in the requester's private cache, and the alarm counter is incremented.

PARSEC applications with very little shared data, such as *blackscholes* and *swaptions*, have no alarms. This is because SHARP can always find a line that exists only in requester's private cache. Applications with a larger amount of sharing between threads (*ferret* and *canneal*) have a relatively higher number of alarms. Even in such cases, however, the number of alarms is orders of magnitude lower than when an attack takes place.

Looking at the last row of the table, we see that *SHARP*4, *SHARP*3, and *query* have less than 2,000 alarms per 1 billion cycles in the worst case. Of them, *SHARP*4 is the best design. Hence, we recommend to use *SHARP*4 and use a threshold of 2,000 alarm events in 1 billion cycles before triggering an interrupt. This is two orders of magnitude lower than that required for a successful attack.

8 RELATED WORK

Various approaches have been proposed to defend against cache-based side channel attacks. They can be categorized into two groups: using cache partitioning to eliminate cache interference, and introducing runtime diversification to limit the effectiveness of these attacks.

Cache Partition Techniques

Cache partitioning prevents a spy from interfering with the victim's cache state by using isolation. Each process is provided a separate portion of the cache. Researchers have proposed both software and hardware partition techniques [8, 24, 29, 44, 46, 52]. SHARP is different from these proposals, as it does not partition the cache. Instead, it changes the line replacement policy to prevent inclusion victims.

Cache partitioning techniques can be divided into process-based and region-based, depending on the granularity of the isolation they support. Process-based cache partitioning divides the cache into multiple partitions, and assigns each partition to a process or a process group. Region-based cache partitioning assigns each partition to a specific region within a program, such as several pages containing security-sensitive code and data. In both categories, no interference between partitions is allowed.

Process-based cache partitioning struggles to attain both good performance and security. Godfrey [8] implements process-based cache partition using page coloring on Xen. Even though this scheme can successfully prevent side channel attacks, it has been shown that it suffers from significant performance degradation when supporting a high number of partitions. SecDCP [46] is a way-partitioning scheme where each application is assigned a security class. Based on the security classes of the applications running concurrently, the scheme dynamically adjusts the partition layout to ensure an application cannot attack another application with a higher security class. However, when applications are in the same security class, the scheme is forced to use static partitioning. In both of the previous schemes, selective cache flushing of partitions is required when the number of processes exceeds the number of partitions available. In addition, both schemes must disable both deduplication and the use of shared libraries.

CacheBar [52] periodically and probabilistically configures the maximum number of ways that a security domain can occupy within each cache set. However, since an attacker can use multiple cooperating threads, CacheBar must limit the number of ways for all unknown processes. This tends to result in unfairness and performance degradation. Moreover, this scheme cannot efficiently support a large number of security domains.

Several pseudo cache partitioning techniques have been studied to provide a fair allocation of resources and/or improve the performance in a shared cache [23, 32, 41, 48]. They try to minimize the interference caused by thrashing/streaming threads by either prioritizing them during victim selection or inserting their lines with a lower priority. Such schemes do not provide any security guarantees and are aimed only at improving the performance of the cache. When they are modified to provide additional security guarantees [43], the mechanisms look similar to the ones we discussed earlier.

Region-based partitioning [24, 29, 44] divides the cache into a secure partition and an insecure partition. The secure partition is reserved for security-critical addresses, while the insecure partition is shared by all processes dynamically.

StealthMem [24] uses page coloring. It reserves several stealth pages via special colors for security-sensitive data. In their scheme, the operating system ensures that these pages are not evicted by normal cache accesses. Catalyst [29] leverages Intel's CAT (Cache Allocation Technology) hardware mechanism to divide the cache into secure and non-secure partitions, and uses software page coloring within the secure partition to block interference between processes requesting protection. Cache line locking [47] allows processes to exclusively use the cache at the granularity of a cache line.

These region-based techniques have a relatively smaller performance overhead than process-based techniques, since they try to maximize the number of dynamic accesses to the shared cache while maintaining sufficient isolation. However, region-based schemes heavily rely on the programmer to achieve good performance. These schemes require the programmer to label the secure-sensitive regions within an application. This is easy to do for cryptography algorithms, since these public libraries are well studied and verified. However, for ordinary applications, it is not trivial to locate important data or execution path regions precisely. SHARP is more practical. SHARP leverages the private caches to hide secret information from the attackers. It does not need any security analysis or modifications to existing software.

Runtime Diversification

Runtime diversification techniques are varied and range from introducing noise to the system clock [16, 34] to randomizing the address mapping [47] and adding noise to the cache insertion policy [30]. Each of these approaches has drawbacks: either it cannot defend against all types of attacks, or it suffers from significant performance degradation.

Wang and Lee [47] propose to dynamically randomize the memory line mapping in L1. Liu and Lee [30] propose the Random Fill Cache for the L1 to defend against reuse-based side channel attacks. Both approaches may suffer substantial performance degradation if applied to the much larger last level cache.

FuzzyTime [16] and TimeWarp [34] disrupt timing measurements by adding noise to the clock or slowing it down. They can protect against attacks which measure cache access latency and execution time, but they are unable to prevent alias-driven attacks [13]. Furthermore, they hurt benign programs that require a high-precision clock.

Detection Techniques

Several approaches have been proposed to detect cache-based side channel attacks. Chiappetta et al. [6] detect side channels based on the correlation of last level cache accesses between victim and spy processes. HexPADS [39] detects side channel attacks by the frequent cache misses caused by the spy process. These heuristic approaches are not robust, and tend to suffer high false positives and false negatives.

CC-Hunter [5] and ReplayConfusion [49] can effectively detect cache-based covert channel attacks. However, they may not be effective for side channel attacks.

SHARP's alarm mechanisms can effectively detect side channel attacks that bypassed the first two steps in the SHARP algorithm. This is because the alarm is incremented every time a spy generates an inclusion victim in another thread. The threshold rate we use is two orders of magnitude lower than the rate of evictions required for a successful attack.

9 CONCLUSION

To combat the security threat of cross-core cache-based side channel attacks, this paper made three contributions. First, it made an observation for an environment with an inclusive cache hierarchy: when the spy evicts the probe address from the shared cache, the address will also be evicted from the private cache of the victim process, creating an inclusion victim. Consequently, to disable cache attacks, the spy should be prevented from triggering inclusion victims in other caches.

Next, the paper used this insight to defend against cache-based side channel attacks with SHARP. SHARP is composed of two parts. It introduces a new line replacement algorithm in the shared cache that prevents a process from creating inclusion victims in the caches of cores running other processes. It also slightly modifies the *clflush* instruction to protect against flush-based cache attacks. SHARP is highly effective, needs only minimal hardware modifications, and requires no code modifications.

Finally, this paper evaluated SHARP on a cycle-level full-system simulator and tested it against two real-world attacks. SHARP effectively protected against these attacks. In addition, SHARP introduced negligible average performance degradation to workloads with SPEC and PARSEC applications.

ACKNOWLEDGMENT

This work was supported in part by NSF under grants CCF 1536795 and CCF 1649432.

REFERENCES

[1] Onur Aciiçmez, Çetin K. Koç, and Jean P. Seifert. 2006. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*. Springer-Verlag, Berlin, Heidelberg, 225–242. https://doi.org/10.1007/11967668_15

[2] J. L. Baer and W. H. Wang. 1988. On the inclusion properties for multi-level cache hierarchies. In *The Conference Proceedings 15th Annual International Symposium on Computer Architecture*. IEEE, 73–80. https://doi.org/10.1109/isca.1988.5212

[3] Christian Bienia, Sanjeev Kumar, Jaswinder P. Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[4] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES, In Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems. *Cryptographic Hardware and Embedded Systems*, 201–215. https://doi.org/10.1007/11894063_16

[5] Jie Chen and Guru Venkataramani. 2014. CC-Hunter: uncovering covert timing channels on shared processor hardware. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 216–228. https://doi.org/10.1109/micro.2014.42

[6] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.* 49, C (Dec. 2016), 1162–1174. https://doi.org/10.1016/j.asoc.2016.09.014

[7] Katherine E. Fletcher, W. Evan Speight, and John K. Bennett. 1995. Techniques for reducing the impact of inclusion in shared network cache multiprocessors. *Rice ELEC TR* (1995). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.6340

[8] Michael M. Godfrey. 2013. *On the prevention of cache-based side-channel attacks in a cloud environment*. Master's thesis. Queen's University.

[9] Daniel M. Gordon. 1998. A survey of fast exponentiation methods. *Journal of Algorithms* 27, 1 (April 1998), 129–146. https://doi.org/10.1006/jagm.1997.0913

[10] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 368–379. https://doi.org/10.1145/2976749.2978356

[11] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag New York, Inc., New York, NY, USA, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[12] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 897–912. http://portal.acm.org/citation.cfm?id=2831200

[13] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: alias-driven attacks and verified countermeasures. In *IEEE Symposium on Security and Privacy*. IEEE, 38–55. https://doi.org/10.1109/sp.2016.11

[14] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[15] Taylor Hornby. 2016. Side-channel attacks on everyday applications: distinguishing inputs with FLUSH+RELOAD. https://www.blackhat.com/docs/us-16/materials/us-16-Hornby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf. (2016). Accessed on 22 April 2017.

[16] Wei M. Hu. 1992. Reducing timing channels with fuzzy time. *Journal of computer security* 1, 3-4 (May 1992), 233–254. http://portal.acm.org/citation.cfm?id=2699810

[17] R. Hund, C. Willems, and T. Holz. 2013. Practical timing side channel attacks against kernel Space ASLR. In *IEEE Symposium on Security and Privacy*. IEEE, 191–205. https://doi.org/10.1109/sp.2013.23

[18] Intel. 2017. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. (2017). Accessed on 22 April 2017.

[19] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: a shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*. IEEE, 591–604. https://doi.org/10.1109/sp.2015.42

[20] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/2897845.2897867

[21] Ameer Jaleel. 2010. Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf. (2010). Accessed on 22 April 2017.

[22] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely, and Joel Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: temporal locality aware (TLA) cache management policies. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 151–162. https://doi.org/10.1109/micro.2010.52

[23] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, NY, USA, 208–219. https://doi.org/10.1145/1454115.1454145

[24] Taesoo Kim, Marcus Peinado, and Gloria M. Ruiz. 2012. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 11. http://portal.acm.org/citation.cfm?id=2362804

[25] J. Kong, O. Aciicmez, J. P. Seifert, and Huiyang Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 393–404. https://doi.org/10.1109/hpca.2009.4798277

[26] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1990. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ACM, New York, NY, USA, 148–159. https://doi.org/10.1145/325164.325132

[27] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (Feb. 1966), 707.

[28] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. ARMageddon: last-level cache attacks on mobile devices. In *25th USENIX Security Symposium*, Vol. abs/1511.04897. USENIX Association.

[29] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture*. IEEE, 406–418. https://doi.org/10.1109/hpca.2016.7446082

[30] Fangfei Liu and Ruby B. Lee. 2014. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 203–215. https://doi.org/10.1109/micro.2014.28

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 605–622. https://doi.org/10.1109/sp.2015.43

[32] Wanli Liu and D. Yeung. 2009. Using aggressor thread information to improve shared cache management for CMPs. In *18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 372–383. https://doi.org/10.1109/pact.2009.13

[33] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power analysis attacks: revealing the secrets of smart cards* . Springer-Verlag New York, Inc., Secaucus, NJ, USA. http://portal.acm.org/citation.cfm?id=1208234

[34] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 118–129. http://portal.acm.org/citation.cfm?id=2337173

[35] Michael Neve and Jean P. Seifert. 2007. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*. Springer-Verlag, Berlin, Heidelberg, 147–162. http://portal.acm.org/citation.cfm?id=1756531

[36] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The spy in the sandbox: practical cache attacks in JavaScript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 1406–1418. https://doi.org/10.1145/2810103.2813708

[37] DagArne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology*, David Pointcheval (Ed.). Lecture Notes in Computer Science, Vol. 3860. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter 1, 1–20. https://doi.org/10.1007/11605805_1

[38] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: a full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. ACM, New York, NY, USA, 1050–1055. https://doi.org/10.1145/2024724.2024954

[39] Mathias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 138–154. https://doi.org/10.1007/978-3-319-30806-7_9

[40] Colin Percival. 2005. Cache missing for fun and profit. http://www.daemonology.net/papers/cachemissing.pdf. (Oct. 2005). Accessed on 22-April-2017.

[41] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 423–432. https://doi.org/10.1109/micro.2006.49

[42] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 199–212. https://doi.org/10.1145/1653662.1653687

[43] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, Vol. 39. ACM, New York, NY, USA, 57–68.

https://doi.org/10.1145/2024723.2000073

[44] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*. IEEE Computer Society, Washington, DC, USA, 194–199. https://doi.org/10.1109/dsnw.2011.5958812

[45] Ronak Singhal. 2008. Inside Intel core microarchitecture (Nehalem). In *2008 IEEE Hot Chips Symposium (HCS)*. IEEE, 1–25. https://doi.org/10.1109/hotchips.2008.7476555

[46] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, New York, NY, USA. https://doi.org/10.1145/2897937.2898086

[47] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, New York, NY, USA, 494–505. https://doi.org/10.1145/1250662.1250723

[48] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Vol. 37. ACM, New York, NY, USA, 174–183. https://doi.org/10.1145/1555815.1555778

[49] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. 2016. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–14. https://doi.org/10.1109/micro.2016.7783742

[50] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 719–732. http://portal.acm.org/citation.cfm?id=2671271

[51] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 990–1003. https://doi.org/10.1145/2660267.2660356

[52] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-Level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 871–882. https://doi.org/10.1145/2976749.2978324