How Sharp is SHARP ?

Dixit Kumar
Indian Institute of Technology Kanpur
dixit@cse.iitk.ac.in

Chavhan Sujeet Yashavant
Indian Institute of Technology Kanpur
sujeetc@cse.iitk.ac.in

Biswabandan Panda
Indian Institute of Technology Kanpur
biswap@cse.iitk.ac.in

Vishal Gupta
Manipal University Jaipur
and Indian Institute of Technology Kanpur
vishalgupta7972@gmail.com

Abstract

Cross-core last-level cache based side-channel attacks are becoming practical, affecting all forms of computing devices like mobiles, desktops, servers, and cloud based systems. Mitigating last-level cache based side channel attacks has become an active area of research and many proposals target to mitigate cross-core based conflict attacks. Secure Cache Hierarchy Aware Replacement Policy (SHARP) is one of the recent proposals that mitigate the conflict attacks by changing the underlying last-level cache replacement policy. Though SHARP is an elegant proposal; there are many subtle points, which were not part of the original SHARP proposal that appeared in the ISCA '17. Through this paper, we discuss and debate the subtle issues that are left unanswered in the original SHARP paper.

1 Introduction

Cache conflict timing attacks at the last-level cache (LLC) are becoming ubiquitous, and one of the fundamental reasons behind many attacks at the LLC is its inclusiveness property. An inclusive LLC is a super-set (in terms of cache contents) of the per-core private caches. This creates a security loophole when exploited carefully as an eviction of a cache block from the LLC, back-invalidates cache blocks in private caches. A back-invalidation is a normal invalidation request that is triggered by the LLC controller (on every LLC eviction) to invalidate the cache blocks corresponding to the evicted address that are present at the private caches. Moreover, future accesses by the victim incur cache misses at its private caches, forcing the victim to access the LLC. There are flush based attacks such as Flush+Reload [20] and Flush+Flush [5], where the attacker does not exploit the inclusiveness nature of the LLC, rather uses a `clflush` instruction to flush a cache block address from all the cache levels and later reloads the same block address. While reloading, if it gets a hit, then the attacker concludes that the victim has accessed the cache block.

Many proposals have been proposed to mitigate LLC timing attacks by changing the cache layout, cache replacement policies, cache addressing, and fuzzing the timers [11, 12, 14, 17, 18, 21]. One of the recent proposals that claim to mitigate the LLC timing attacks is an LLC replacement policy named secure hierarchy-aware cache replacement policy (SHARP) that appeared in one of the flagship Computer Architecture Conferences named ISCA '17 [19]. In this work, we ask a few subtle and interesting questions and try to answer them one by one.

1.1 A Primer on SHARP

SHARP is an LLC replacement policy that prevents cross-core back-invalidations (inclusion victims) at the private caches, nullifying the cross-core attacks that exploit the inclusiveness property. The premise used by SHARP is that cross-core back-invalidations make the attacker successful at the LLC. SHARP can be applied to any baseline cache replacement policy like LRU or a more recent re-reference interval prediction (RRIP) [9] based policies. SHARP affects the underlying replacement policy as follows: Whenever an LLC miss happens, SHARP kicks in. If that miss causes a cross-core eviction (meaning a core *i* evicting a cache block that was brought by core *j*) of a cache block then SHARP kicks in and it does not allow the cross-core eviction if the block that selected for eviction is present in other cores' private caches. SHARP searches for a cache block (as per the priority order of the underlying cache replacement policy). The eviction process happens in three stages, as follows: (i) a block that is not present in any cores' private caches, (ii) if stage (i) fails then it searches for block(s) that is present only in the attacker's private cache (it allows eviction of block(s) that cause intra-core back-invalidation), and (iii) in the worst case, if SHARP cannot find a suitable cache block then it goes for a random replacement policy and replaces a random block and increments an alarm counter (which is a per core counter) for the core that is causing the cross-core back-invalidation, anticipating an attack. Figure 1 shows the three stages of interest from SHARP's point of view.

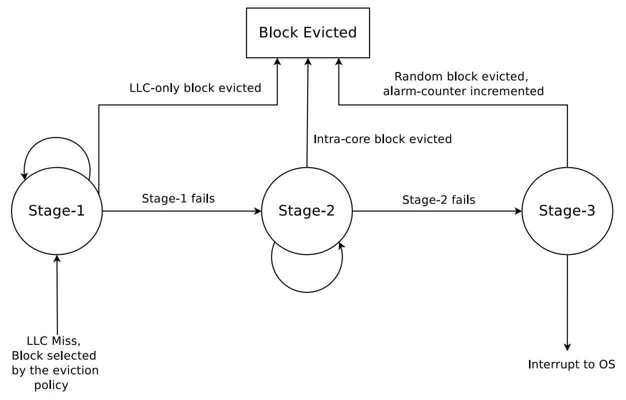

Figure 1: Different stages of SHARP in a nutshell. LLC-only block is the block that is present only in the LLC and not present in any core's private caches. Intra-core is the block that is present only in the evictor's private cache and the LLC.

Whenever a random block is replaced by SHARP it increments an alarm counter and the moment the counter crosses a predefined threshold, the core generates an interrupt to the operating system (OS). However, SHARP does not explain what exactly the OS does. Some possible OS responses include: (i) de-schedule the application that crosses the alarm counter threshold, (ii) migrate the application to another socket of a multi-core system if available, or (iii) it may have to kill the application.

For `clflush` based attacks, SHARP argues that "there is no need to use `clflush` in user mode for pages that are read-only or executable". The argument holds for most of flush based attacks that use shared library code that are either read-only or executable. Next, we ask a series of questions pertinent to SHARP.

1.2 Questions of Interest

SHARP raises some interesting questions that follows:

- Does SHARP mitigate all kinds of LLC eviction attacks? If the blocks of interest are writable then the attacker may first invalidate the victim blocks in the victim L2 through cache coherence, and then evict them from the LLC. In such cases, the LLC eviction will not cause back-invalidation hits at the private cache of the victim. So, SHARP does not seem to cover all possible cross-core conflict based LLC side channels.

- Does SHARP mitigate few cross-core attacks and facilitate a few more attacks? For example, does SHARP help to mount a new Denial of Service (DOS) attack? To fool the SHARP if an application occupies the entire L1, L2, and LLC with the overlapping data with the same set index. Now, any eviction made by another application to the first application's block will be prevented if it hits at the first application's L2. So this is some form of DOS attack where a fixed number of blocks of a cache set are locked, decreasing the effective LLC capacity usage? (Section 4)

- Does the threshold used by SHARP based on alarm counter can affect legitimate applications? Can a SHARP-aware attacker play with SHARP to mount new form of tricky attacks that are only possible because of SHARP? (Sections 5 and 6)

- What exactly the OS does when it receives an interrupt (whenever the alarm counter crosses the threshold)? (Section 6)

- How secure is SHARP in terms of information leakage? (Section 7)

2 Background

This section provides background on different cross-core conflict based side-channel attacks (miss type and hit type) at the LLC. In miss type attacks, the attacker is interested in observing longer cache access time, because of cache misses (miss access can be either from the victim or the attacker). In contrast, in hit-based attacks, the attacker is interested in shorter access time (hits). All the attacks measure LLC access time. However, some attacks do it precisely per memory access (access based attacks), and some accumulate the timing information for the entire security-critical accesses (timing based attacks). Primarily, there are three different strategies such as (i) Evict+Reload [6] (a variant of Flush+Reload attack where the Flush operation is replaced by the Evict operation), (ii) Evict+Time [15], and (iii) Prime+Probe [16]. In flush based attacks such as Flush+Reload [20], the attacker uses `clflush` instruction to flush a cache block address from all the cache levels and later reloads the same block address. While reloading, if it gets a hit, then the attacker concludes that the victim has accessed the cache block.

Evict+Time [15]: In this attack, the spy observes the execution time of the victim over a large number of intervals. First, the spy evicts cache blocks from a few set(s) at the LLC. Later, when the victim accesses the evicted block(s), it results in a longer access time and the spy observes the same.

Evict+Reload [6]: In Evict+Reload attack, the spy evicts a cache block from the LLC. After an interval (predetermined fixed value), the spy reloads the same address and if it gets a shorter access time (an LLC hit), then it concludes that the victim has accessed the same cache block.

Prime+Probe [16]: In this attack, the attacker loads its cache blocks by evicting the blocks of the victim (the prime part). Then the victim executes its secure operation and in the process, gets LLC misses, evicts the blocks brought by the attacker. Next, the attacker probes its execution time by reloading its blocks, to see whether it gets longer access time

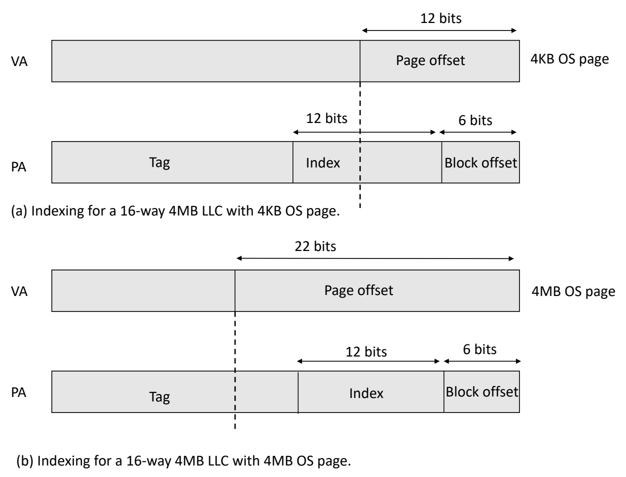

Figure 2: Effect of the huge page on LLC indexing. VA: virtual address and PA: Physical address.

because the victim has evicted the block (an LLC miss).

As per SHARP, all these attacks are successful because of cross-core back-invalidation hits. Out of all these attacks, Evict+Reload attack demands the notion of sharing of OS pages between the victim and the spy. The attack is more precise (operates at specific block addresses).

3 SHARP and a Series of Attacks

In this section, we showcase some of the attacks that are possible with SHARP using huge OS pages. SHARP claims that it mitigates all kinds of cross-core conflict based attacks. However, we find that there are still a good number of attacks that are successful in the presence of SHARP. The premise used by SHARP to mitigate the cross-core conflict based attacks is that it does not create cross-core inclusion victims and prevents cache replacement that can lead to creation of inclusion victim. However, what if the attacker makes sure that it attacks a cache set where the blocks that are present in that set at the LLC are not present in its L1 or L2.

3.1 Simulation Methodology

We use the ChampSim [1] simulator to simulate SHARP replacement policy on a 16-core system. As SHARP is not yet implemented on real machines, we simulate it using an architectural simulator. We use dynamic RRIP (DRRIP) [9] as the baseline replacement policy and apply SHARP on it. ChampSim is a trace driven simulator used for Cache Replacement Championship held with ISCA '17. Table 1 shows the parameters used in our simulated system. Note that the parameters correlate to Intel machines that used to have inclusive LLCs. Next, we discuss a series of attacks that are still possible with SHARP and with huge OS page. Note that we showcase

Table 1: Parameters of the simulated system.

Processor	16-cores, out of order
L1 D/I, L2	8 KB (8 way), 256KB (8 way, inclusive)
Shared L3	2MB× cores, #slices=#cores, 16 way, inclusive
MSHRs	8, 16, 32×16 MSHRs at L1, L2, L3
Cache line size	64B in L1, L2 and L3
Replacement policy	DRRIP and SHARP with DRRIP
DRAM controller	4 controllers for 16-cores, Open Row, 48 read/write queues, FR-FCFS, drain-when-full
DRAM bus	split-transaction, 800 MHz, BL=8
DRAM	DDR3 1600 MHz (11-11-11) Max bandwidth/channel - 12.8 GB/sec

the same on non-crypto applications like SPEC CPU 2017 benchmarks. Also, because of space limitations, we do not explain all the details related to all the attacks. Instead, we provide a top level view on each of these attacks.

3.2 SHARP with Huge OS Page

The original SHARP proposal shows its effectiveness with a small OS page size of 4KB. However, if modern systems employ huge pages (in MBs and GBs) to improve system performance. Figure 2 shows the effect of huge pages on LLC indexing as most of the lsbs do not get changed from the virtual to physical address translation. For example, for a 4MB OS page, the lower 22 bits remain the same. As mentioned in prior works like [13], it is easier for an attacker to mount the cross-core attacks as the attacker knows the LLC set index from its virtual address itself.

3.3 Prime+Reprime+Probe Attack [8]

For this attack, we simulate a 16-core system using ChampSim, sharing an LLC of 32MB having 16 ways, with huge OS page size of 4MB. The attacker uses cooperative threads using multi-threading so that 15 attacker threads mapped to 15 physical cores, can access a particular cache set, which is also the cache set of interest. In a nutshell, the attacker does the following:
(i) the attacker primes the LLC by evicting victim's data. However, SHARP does not allow evicting victim's blocks. To fool the SHARP, attacker threads make sure that SHARP does not evict their data. To accomplish this the attacker makes sure that the data present in the LLC is also present in their respective private caches. At this point, if one of the core tries to evict a block, SHARP finds that all the blocks are failing

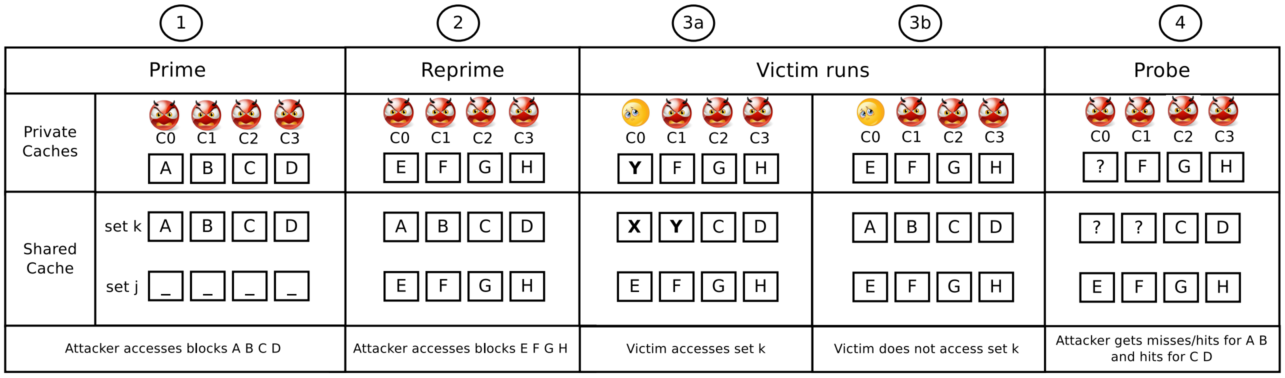

Figure 3: Prime+Reprime+Probe attack with SHARP. in (④, attacker gets misses for blocks A and B if victim accesses (③a)) and hits if the victim has not accessed (③b))

stage-1 and stage-2 and it evicts a block randomly (① of Figure 3). (ii) Next all the attacker threads re-prime (②) so that their own data get evicted from their respective L1 and L2s as mentioned in [8].
(iii) Now, the victim accesses LLC and gets LLC misses. Victim evicts attacker's blocks (③) even with SHARP because the attackers have already ensured that the blocks present at the LLC are not present in their private caches. So at this stage, the security guarantee of the SHARP is compromised.
(iv) Attacker probes (④) the LLC addresses (using the 15 threads) and depending on the #accesses made by the victim in step (iii), the attacker finds out whether the victim has accessed the cache set (longer LLC access time) or not (shorter access time).

4 SHARP induced New Attacks

4.1 Prime+Reprime+Reprime Attack

With SHARP, a cross-core attacker cannot prime the entire LLC cache set, which already indicates there is a valid copy in the victim's private L2 cache, which is some form of information leakage already. Assuming the attacker knows SHARP is used at the LLC, the attacker can perform a Prime+Reprime+Reprime attack to know the behavior of a victim's private caches:
Prime: At time t1, the attacker primes the LLC by filling a cache set (with w lines) with its own data. In our case, with SHARP, w can be 16 or less than 16, for a 16-way LLC.
Reprime: The attacker reads data from the w cache blocks and measures how many of them are loaded from LLC (e.g., k out of w).
Reprime: After a fixed interval, say at time t2, the attacker reads data from the w memory blocks again and measures how many of them are loaded from LLC (e.g., k').

The attacker now learns the following information: at time t1, w-k memory blocks that map to this cache set of LLC are used by the victim in the private cache; at time t2, w-k' memory blocks that map to this cache set of LLC are used by the victim in the private cache. The difference, k'-k, indicates the changes of victim's memory accesses during this period. This provides the cache occupancy information about the victim's private cache. As per SHARP, "*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.*."

4.2 Denial of Service Attack

With SHARP, if the attacker deliberately occupies its entire private cache space (L1 and L2) with non-overlapping data and it also occupies this part in the LLC then any prevention of back-invalidation hits from LLC would lead to blocking of LLC space. This could be used to mount a denial of service attacks on specific critical cache sets. It also reduces the LLC capacity. Figure 4 explains the attack where a 4-threaded attacker (for better illustration, we use four threads only) can mount a denial of service on a 4-way LLC. The same can be done on a 16-way LLC using 16 threads. The attacker can find the critical sets of interest based on the attack that we discuss in the previous subsection (Prime+Reprime+Probe attack). Once the attacker(s) know the critical sets then a multi-threaded attacker can occupy the entire cache set and the attackers can make sure that the contents of 16 ways at the LLC are present in 16 private caches (in one of the ways of L1s/L2s) (① of Figure 4). Then when the victim comes, it finds the cache set full and SHARP tries its best to prevent cross-core eviction. However, as all the blocks are present in private caches too, SHARP does a random eviction evicting one of the attacker blocks (②). After this, all future accesses of the victim leads to eviction of victim's block only as SHARP prefers evictions of cache blocks that cause intra-

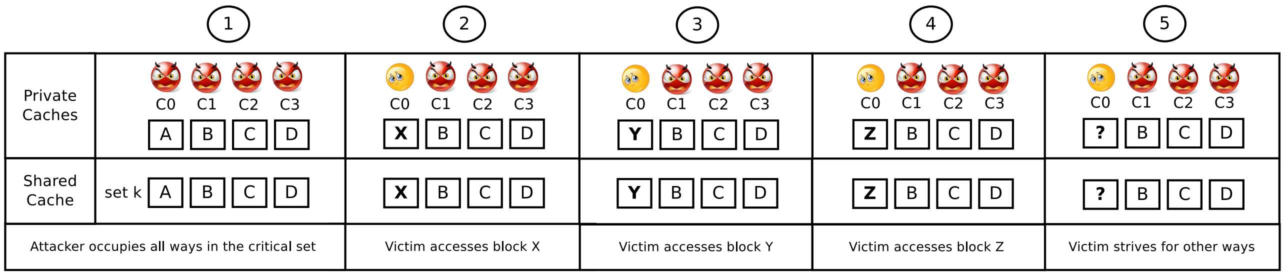

Figure 4: Denial of Service attack with SHARP. A particular way is used by the victim and rest are denied by the attacker.

Table 2: Thrashing benchmarks used in 16 core combinations.

Mix No.	Thrashing Benchmarks
1	605.mcf-484B
2	605.mcf-665B
3	605.mcf-994B
4	607.cactubssn-2421B
5	620.omnetpp-141B
6	620.omnetpp-874B
7	621.wrf-6673B
8	623.xalancbmk-10B
9	649.fotonik-10881B
10	654.roms-523B

core back-invalidation hits over inter-core back-invalidation hits. In this way, the attackers always occupy the 15 ways of a 16-way LLC, blocking the cache capacity of critical sets of the victim and the victim keeps on evicting its own blocks (③, ④, and ⑤).

5 The Threshold Dilemma

As per SHARP, "*When the alarm event counter of any core reaches a threshold, a processor interrupt is triggered. 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.*"

SHARP argues an alarm threshold of 2000 per one billion cycles is needed to trigger the OS event anticipating that the application is an attacker and SHARP is unable to prevent it completely. This means if an application crosses the value of 2000 in terms of inter-core back-invalidation then this application must be an attacker application. So, once SHARP detects the attacker application, the OS kicks in. *However, the role of an OS is not clear. SHARP does not explain what exactly is the role of an OS*. We *speculate* the OS can delay that application by de-scheduling, migrating the application if the system is a multi-socket system, or in the worst case, the OS may kill the application.

SHARP showcases threshold numbers by running different LLC thrashing and LLC fitting applications. An LLC thrashing application is an application with high memory footprint and has higher misses per kilo instruction (MPKI) whereas a LLC fitting application is an application with LLC MPKI closer to zero meaning the memory footprint fits into the LLC. We reproduce the experiments of SHARP but for a 16-core system sharing 16-way LLC and with huge page ON. We find that the inter-core back-hit count crosses the threshold mentioned by SHARP.

Multi-core combinations: We consider four combinations of 16-core experiments: (i) 16-0 (16 thrashing applications running on a 16-core system), (ii) 12-4 (12 thrashing applications running along with four fitting applications on a 16-core system), (iii) 8-8 (eight thrashing applications running along with eight fitting applications), and finally (iv) 4-12 (four thrashing applications running along with 12 fitting applications). In total, we use 10 thrashing applications from SPEC CPU 2017 benchmark suite (traces collected from [4]) as mentioned in Table 2. These benchmarks are picked from their region of interests (marked with an instruction count followed by billions of instructions). For fitting application, we use `641.leela-1052B`. In the baseline, we use the dynamic RRIP (DRRIP) [9] policy. Figure 5 shows the inter-core back-hit rate for DRRIP replacement policy without SHARP for four different combinations of thrashing and fitting benchmarks, with ten specific thrashing applications as mentioned in Table 2. For a better understanding, the plot with 16-0 means 16 thrashing applications running on a 16-core system, where the thrashing application is denoted by the benchmark id, as mentioned in Table 2. So, the sixth point in the X-axis says 16 copies of 620.omnetpp-874B are running concurrently on a 16-core simulated system. It is clear from Figure 6 that SHARP is successful in reducing the inter-core back hit rate.

As expected SHARP reduces the inter-core back-hit rate, as it prioritizes intra-core eviction over cross-core eviction during the eviction of a cache block. In combination, 12-4 (12 thrashing applications running with four fitting applications), for thrashing applications, the inter-core back-hit rate reduces from a maximum of 12% to 0.5%. However, 0.5% of inter-core back-hit rate for thrashing applications is still good enough for exceeding the alarm counter threshold. To

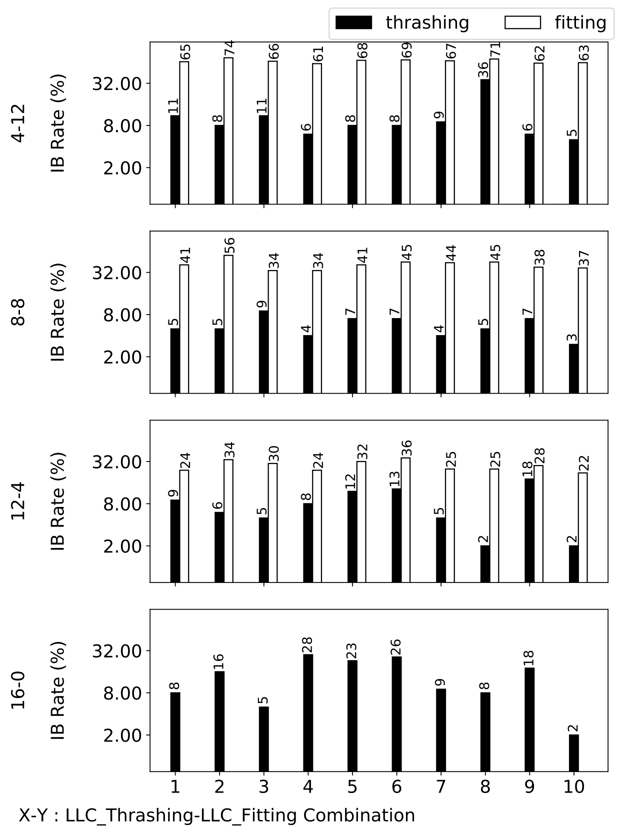
Figure 5: Inter-core back-hit rate without SHARP for four different 16-core combinations involving LLC thrashing and LLC fitting applications. IB rate: Inter-core back-hit rate.

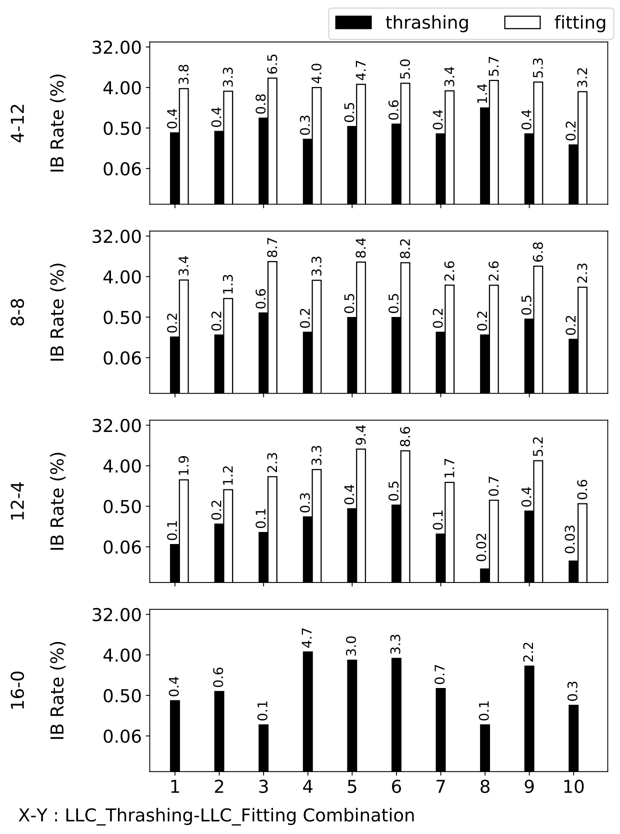
Figure 6: Inter-core back-hit rate with SHARP for four different 16-core combinations involving LLC thrashing and LLC fitting applications. IB rate: Inter-core back-hit rate.

understand the situation better, Figure 7 shows the average MPKI numbers averaged across thrashing applications at the L2 and LLC. As we can see, thrashing applications have high MPKI, so even a small inter-core back-hit rate can magnify itself to large alarm counter values.

Effect on the alarm counter: Figure 8 shows the range of alarm counter values that we see for four 16-core combinations. As, we can see, with 16-0 combination (16 different copies of thrashing applications running concurrently), `607.cactubssn-2421B` has a counter value of 264,479, which is 132X times more the threshold set by SHARP. For other combinations, counter values in the range of above 40,000 is observed (for combinations 4-12 and 12-4). Note that these counter values are per one billion cycles and SHARP resets these counters after every one billion cycles. It is obvious that with 16-0 combination, the counter value becomes so large as all 16 applications are mapped to same cache sets thanks to the 4MB huge page. This causes significant cross-core evictions by SHARP raising the alarm counter 132X times of the SHARP threshold. For combinations like 12-4 and 4-12, this number reduces, but it is still 20X more than the SHARP threshold. In 4-12 and 12-4 combinations, thrashing applications cause cross-core evictions of fitting applications. *This shows that SHARP can treat legitimate applications as attackers. SHARP can increase the threshold and make it core count specific. However, SHARP claims the following and we quote: "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*".

Our experiment shows that 16-0 combination of legitimate applications show alarm counter value of 264,479. An immediate solution is to increase the threshold value to a high number, say 300,000. However, we find that 300,000 cross-core evictions is more than enough for any Evict+Reload and Prime+Probe attacker attacking applications mentioned in the SHARP paper like GnuPG [2] and Poppler [3]. So, a better

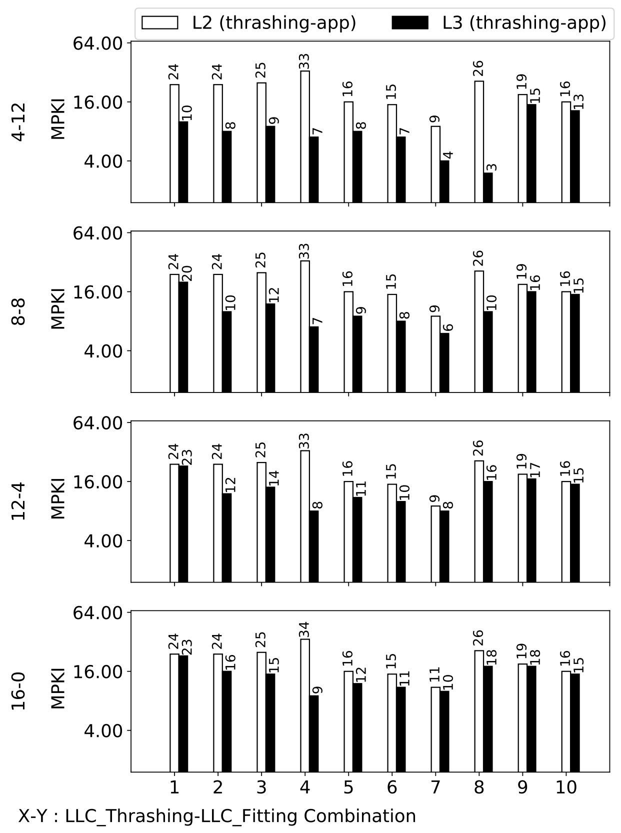
Figure 7: L2 and LLC MPKI of thrashing applications for different 16-core combinations with SHARP.

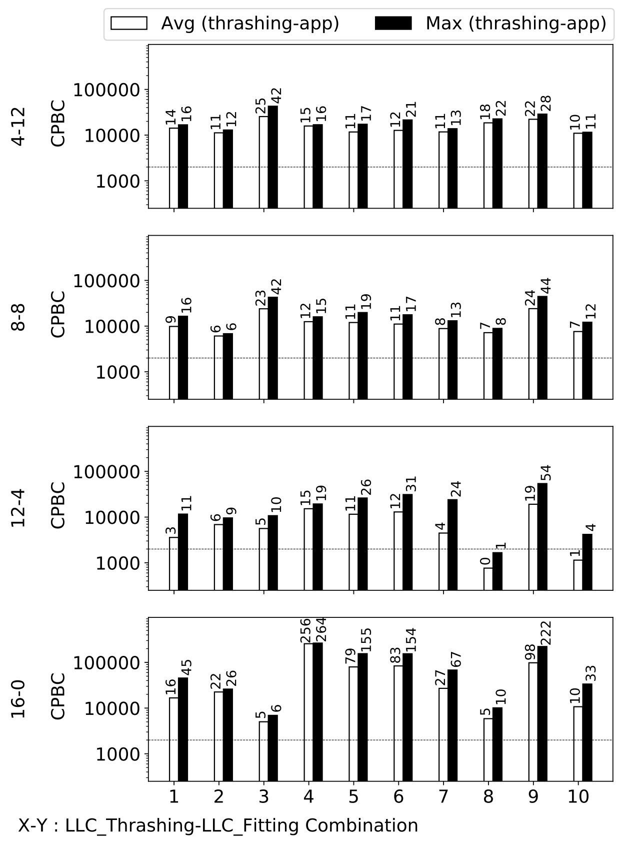
Figure 8: Average and maximum values of alarm counter (in thousands) with SHARP. The X-axis shows the thrashing benchmarks used as per Table 2. CPBC: counter per billion cycles.

solution is needed to detect the attacker.

6 Role of an Operating System

SHARP does not explicitly mention the role of an OS. OS gets an interrupt when a processor core crosses the SHARP threshold for the alarm counter. We debate and discuss about three possibilities that an OS can explore:

- **To de-schedule:** The OS can de-schedule the application running on a particular core that has crossed the threshold. However, as we have seen, there are combinations where all 16 applications cross the threshold. So the OS has to de-schedule all 16 applications within the interval of one billion cycles. Note that thrashing applications like `mcf and fotonik` take trillions of cycles to complete their execution. So, we believe, it will have a serious impact on the execution time of individual applications that are part of the 16-0 combination. The most practical solution that we can think of is the OS can provide a time quantum to each of the applications that have crossed the threshold. The OS can run in two different modes: (i) high priority mode, and the (ii) normal mode. If an OS runs a group of applications in the high priority mode then these applications will finish quickly as the shared resource contention will be less. However, this policy may not work if the number of applications that cross the threshold does not follow a uniform distribution.

- **To migrate:** This is one of the simple but costly options where the OS can choose to migrate the application from one socket to another socket provided the multi-core system consists of multiple sockets (Figure 9). However, as we see in the behavior of 16-0 combination, SHARP causes the system to enter a situation where an OS will have a ping-pong effect, migrating multiple applications from one socket to another socket after one billion cycles or after a smaller interval than one billion cycles (if it reaches the threshold before one billion cycles). If we run 32 copies of `649.fotonik` on a 32-core system having two sockets, each having 16 cores them each of them will have performance overhead as the migration cost is highly non-deterministic and it can go up to millions of cycles.

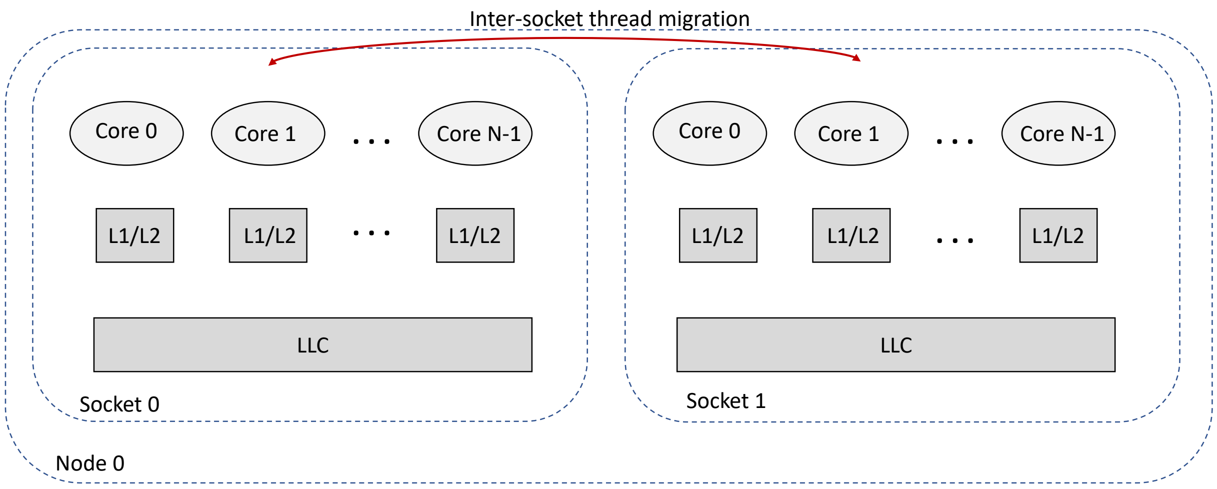

Figure 9: Thread migration with multiple sockets.

Table 3: Conditional probabilities of events of interest based on PIFG.

	Events
p1	Memory block getting mapped into a cache set
p2	Cache block selected for replacement given the cache set
p3	Cache block selected by the replacement policy is evicted
p4	Evicted block when accessed again gets an LLC miss and the very next access gets a hit.
p5	LLC hit/miss getting mapped to the shorter/longer access time.

We try to quantify the effect of de-scheduling and migration on the execution time slowdown. Figure 10 shows the slowdown in the execution time. Out of 16 applications that are mapped to 16 cores, more than 50% of the applications experience slowdown. For delay of less than 1M cycles, the slowdown is negligible. However, beyond that, there are slowdowns of 2.5X and 30X for delay of 16M and 256M cycles, respectively. Latency numbers of 16M to 256M are some of the worst case latency numbers that we observe on a real 16-core machine.

- **To kill:** The last option any OS designer will prefer is to kill an application. As we discuss in the previous Section, if killing is an option then all legitimate applications can get killed. We do not believe, this is a viable solution. Figure 11 shows the fraction of applications that can get descheduled, migrated, and killed if the OS chooses to de-schedule, migrate, or kill the applications that have cross the threshold. For combinations like 16-0 and 8-8, almost 100% applications get affected. So we do not show the same in Figure 11.

Not specifying the OS response to alarm counter threshold is a limitation of the SHARP proposal. In the discussion above we show that several intuitive approaches have detrimental effect on the system performance. We are not aware of potential approaches for handling alarm counter threshold that do not affect system performance.

6.1 SHARP Threshold Aware Attack

Apart from the threshold dilemma that we discussed in the previous section, we believe, a new kind of attack can be proposed, which we describe next. SHARP maintains alarm counter per core and not per process. Second, the alarm counter gets reset in one of these two cases: when alarm counter reaches the threshold value of 2000 or after every one billion cycles. An attacker can first experiment to find out the alarm counter value. It can be easily done depending on the OS role and attacker's own execution time. To do this, the attacker can perform a lot of cross-core eviction by thrashing the LLC. Assuming, the attacker knows the threshold value, the attacker can run for a while doing cross-core eviction until it reaches counter value nearer to the threshold. At this point, the attacker forces itself to sleep, fooling the OS. Then the OS schedules another process and the moment that process starts doing cross-core eviction, the processor core generates an interrupt and the recently scheduled process gets de-scheduled, migrated, or killed as per the OS functionality defined based on SHARP. Note that, the attacker has to make sure it reaches nearer to threshold within one billion cycle window else the counter will be reset. Figure 12 illustrates this attack.

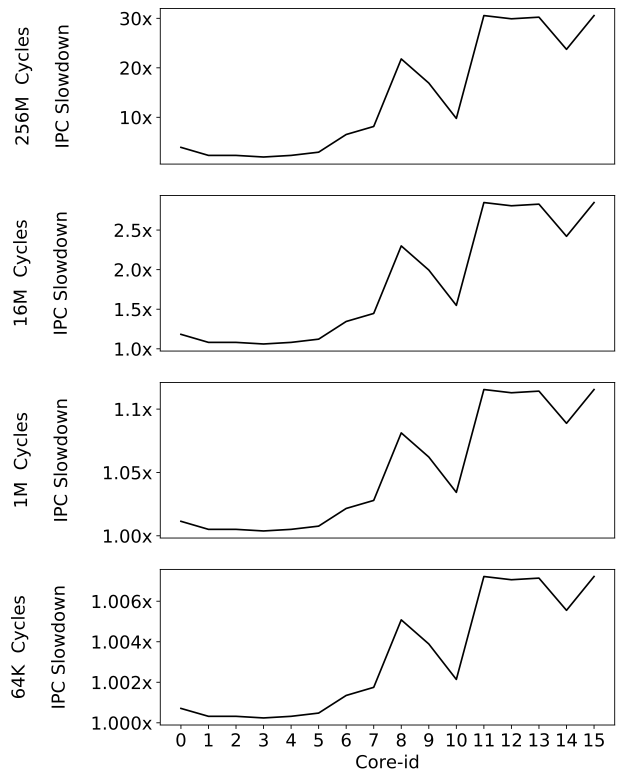

Figure 10: Slowdown with OS de-schedule and migration for 16-0 combination running on core id 0 to 15.

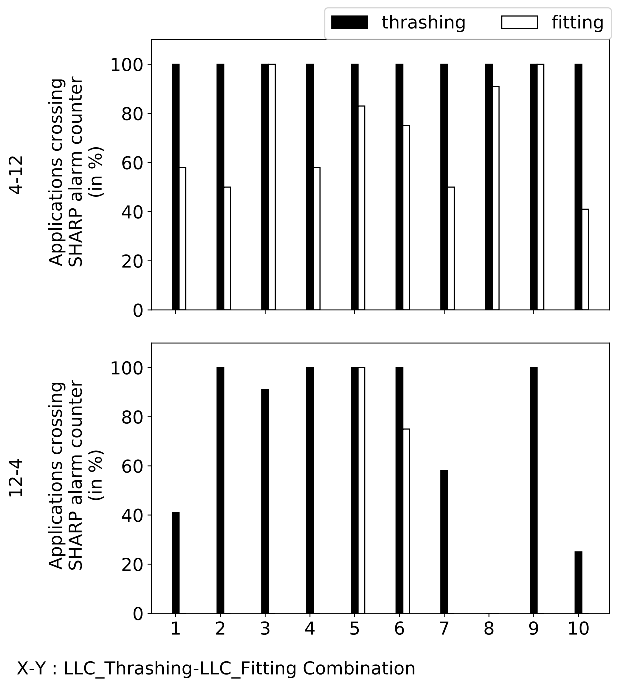

Figure 11: Fraction of applications (out of 16) that can get de-scheduled, migrated, or killed with SHARP.

7 SHARP and Information Leakage

To compare different micro-architecture techniques in terms of information leakage, metrics such as true positive rate (TPR), which is the ratio of true critical accesses observed by the attacker and the number of critical accesses of the victim and Cache side-channel vulnerability (CSV) [10] (Pearson's correlation coefficient between the victim and attacker traces at the LLC) are proposed. Recently, He and Lee proposed a nice and more generic model called Probabilistic information flow graph (PIFG) [7] to quantify the probability of attack success (PAS). A PAS value closer to 0 is better and secure.

PAS [7]: Table 3 shows conditional probabilities of interest through which the information flows from the victim to the attacker, for all various cross-core eviction based attacks at the LLC. For a detailed overview on PIFG, please refer [7].
p1: 1.00, conventional mapping in which a DRAM address mapped to a particular cache set with probability 1.00 and it is known to the attacker. If it is not known to the attacker, then it will be less than 1.00.
p2: 1.00, for a successful attack, the attacker should be able to replace the cache block(s) of interest before the victim reloads. For a w-way cache, the attacker should access a particular set at-least w times for LRU based policy and $\bar{w}$ ($\bar{w}$ can be less than equal to w or greater than w) times for RRIP [9] based eviction policies.
p3: 1.00, this probability will change if we prevent replacement of the block of interest.
p4: 1.00, the attacker observes an LLC miss/hit in miss/hit type attacks.
p5: 1.00, a direct correlation between miss/hit with the LLC access time. So the PAS of the baseline system is **1** ($p1 \times p2 \times p3 \times p4 \times p5$). Next, we show the PAS for cross-core miss-type attacks, which is easy to understand followed by the hit-type attacks.

If SHARP is secure then the expected PAS with SHARP should be zero indicating no information leakage. However, we believe that it is not the case. With SHARP, an additional event of interest comes into the picture, which is the probability of an evicted block creating an cross-core back-invalidation hit. Theoretically, for an LRU replacement policy, the worst case probability will be $\frac{(n-1)\times L2size}{LLCsize}$. However, this number may come out small. If we change the replacement policy to say some variant of RRIP (like DRRIP) then the probability shoots up significantly.

We add an additional probability (p) that provides the probability of an LLC eviction resulting in cross-core inclusion victim. So, the baseline PAS will be extended to PAS$\times\mathrm{p}$. SHARP will be secure if p becomes zero. However, there are applications that can be created to exploit the cache hierarchy and the LLC replacement policy to have p value closer to **0.25**. So, with non-zero inter-core back-hit rate, which makes the security guarantee of SHARP probabilistic in nature. This is

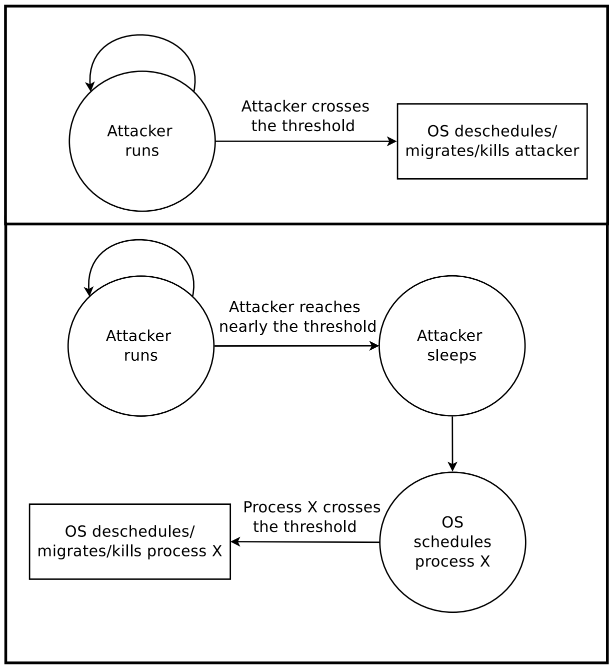
Attacker runs
Attacker crosses the threshold
OS deschedules/ migrates/kills attacker
Attacker runs
Attacker reaches nearly the threshold
Attacker sleeps
OS schedules process X
Process X crosses the threshold
OS deschedules/ migrates/kills process X

Figure 12: SHARP threshold ware attack.

a serious concern, because for applications like GnuPG, if an attacker can leak a few bits then rest can be extracted through a brute-force approach.

8 Conclusion

Through this paper, we discussed and debated a few subtle issues that were unanswered in the original SHARP paper. We discussed new possible attacks, issues with the alarm threshold, and the role of operating system. We believe, these issues are important and the community should look at these issues and try to find out better solutions. Fixing these subtle issues will make SHARP sharper in mitigating cross-core last-level cache based side-channel attacks.

9 ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers for their helpful comments and suggestions. We would also like to thank members of CARS research group (especially Upasana Singh) for their feedback on the initial draft. This work is supported by the SRC grant SRC-2853.001.

References

[1] Champsim, https://github.com/ChampSim/ChampSim.

[2] Gnupg, https://www.gnupg.org/software/index.html.

[3] Poppler, https://poppler.freedesktop.org/.

[4] Spec 2017 traces, http://hpca23.cse.tamu.edu/champsim-traces/speccpu/.

[5] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 279–299, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[6] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 897–912, 2015.

[7] Zecheng He and Ruby B. Lee. How secure is your cache against side-channel attacks? In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 341–353, New York, NY, USA, 2017. ACM.

[8] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604, May 2015.

[9] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 60–71, 2010.

[10] Ruby B. Lee and Weidong Shi. HASP 2013, the second workshop on hardware and architectural support for security and privacy, tel-aviv, israel, june 23-24, 2013. ACM, 2013.

[11] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016.

[12] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 203–215, Washington, DC, USA, 2014. IEEE Computer Society.

[13] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[14] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[15] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[17] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 74:1–74:6, New York, NY, USA, 2016. ACM.

[18] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[19] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 347–360, New York, NY, USA, 2017. ACM.

[20] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.

[21] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 871–882, New York, NY, USA, 2016. ACM.