IVcache: Defending Cache Side Channel Attacks via Invisible Accesses

Yanan Guo
University of Pittsburgh
Pittsburgh, PA, USA
yag45@pitt.edu

Andrew Zigerelli
University of Pittsburgh
Pittsburgh, PA, USA
anz37@pitt.edu

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

ABSTRACT

The sharing of last-level cache (LLC) among different CPU cores makes cache vulnerable to side channel attacks. An attacker can get private information about co-running applications (victims) by monitoring their accesses in LLC. Cache side channel attacks can be mitigated by partitioning cache between the victim and attacker. However, previous partition works either make weak assumptions about the attacker's strength or force their security mechanisms and thus overhead to every user on the system, regardless of their security requirement.

We argue that offering security protection as a service is a better choice for secure cache design. To achieve this, we propose Invisible-Victim cache (IVcache), a new cache partition design targeting both the original LLC attack and the new variant. IVcache classifies all security domains on the system as protected and unprotected. For LLC accesses from protected domains, IVcache handles cache state changes in a slightly different way to make those accesses invisible to any other security domains. We implement and evaluate IVcache in Gem5. The experimental results show that IVcache can defend against real-world attacks, and that it introduces negligible performance overhead to protected domains and no overhead to unprotected domains.

CCS CONCEPTS

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

KEYWORDS

Security; Side channel; Cache

ACM Reference Format:
Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2021. IVcache: Defending Cache Side Channel Attacks via Invisible Accesses. In *Proceedings of the Great Lakes Symposium on VLSI 2021 (GLSVLSI '21), June 22–25, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3453688.3461481

GLSVLSI '21, June 22–25, 2021, Virtual Event, USA.

ACM ISBN 978-1-4503-8393-6/21/06…$15.00
https://doi.org/10.1145/3453688.3461481

1 INTRODUCTION

Cache timing side channel attacks, or cache attacks for short, have been demonstrated to be extremely potent. Different cache behaviors, such as hits and misses, create significant execution timing differences. Attackers are thus able to use this timing information to monitor other processes' (victims') cache behavior and possibly infer secrets. Cache attacks work on both the private and shared caches. However, for the private caches (e.g., L1 cache), the victim and attacker must run on the same core. In contrast, shared cache attacks (e.g., last-level cache attacks) are much more powerful because the attacker can be on a different core than the victim. Many different architectural solutions have been proposed to mitigate timing attacks on last-level cache (LLC) [9, 10, 13, 15, 17], which can be classified into randomization-based solutions and partition-based solutions. A popular way to implement randomization-based defenses is to create a random mapping between cache lines and cache sets [13, 15]. Unfortunately, these designs have been proven insecure [12].

The root cause of cache attacks is that the victim's execution changes cache states which are visible to attackers. Thus, to make a rigorous defense, we should hide the victim's cache behavior from the attacker's view. From this standpoint, partition-based solutions seem to be more effective. Once the victim and attacker are completely separated in cache, the attacker cannot observe the victim's access patterns anymore. However, most prior partition designs [6, 14, 17] only focus on defending the original Prime+Probe attack [11]. With the new discovery of the attack based on replacement states [16], these defenses become insecure. Very recent partition designs [5, 10] cover this new attack; however, they either cause significant overhead, or force their security designs to every user, which is not fair to security-insensitive users.

A fair partition design should distinguish between security sensitive users (i.e. victims) and other users, protecting victims without generating performance overhead to other users, instead of forcing the security design and overhead to everyone. Thus, we propose Invisible-Victim cache (IVcache). IVcache is a novel cache partition design that prevents LLC attacks by making the victim's cache behavior "invisible" to all other processes. IVcache targets both

the original cache attacks and the new variants. We enumerate all situations a victim's access could face in LLC, and show how to make cache state changes caused by the victim invisible. Our design trades some performance for security, but we also develop two optimization mechanisms to salvage the performance of the victim, without affecting other users.

We implement IVcache in Gem5 [2] and test it against a real-world attack, showing that we can effectively prevent it. In addition, we run workloads derived from SPEC2017 [1] to test the performance of IVcache. The experimental results show that IVcache causes negligible overhead to the victims and slight performance benefit to the co-running non-victim applications.

2 BACKGROUND AND RELATED WORK

2.1 LLC attacks

There are two general approaches for LLC attacks, according to whether or not the attack requires data sharing between the attacker and victim. In sharing-based attacks, including Flush+Reload [20], Evict+Reload [8], and the coherence attack [19], the attacker observes the victim's access pattern to a shared cache line to infer the victim's secrets. These attacks can be stopped by disallowing the sharing of sensitive data.

Prime+Probe [11] is a more general LLC attack as it does not require data sharing. In Prime+Probe, the attacker fills a target LLC set with his own cache lines during the prime stage. He later probes this LLC set with the same cache lines and times the accesses: if a cache miss is detected, it indicates that the victim accessed this LLC set, exposing the victim's access.

In Prime+Probe, the attacker forces the victim to have cache **misses** because a cache miss from the victim evicts the attacker's cache line and thus causes leakage. In fact, when the victim's access **hits** in LLC, it changes the replacement state (e.g., LRU state) of the attacker's cache line (in the same LLC set); this state change can also be observed by the attacker. In 2020, Xiong et al. proposed an attack based on LRU state changes [16]. The basic attack repeats the following steps:

a) Victim accesses his line in an LLC set, and then the attacker accesses his $(w-1)$ *initial lines* in this set, where w is the set associativity. After this, the victim's line will become the oldest one in this set.
b) Attacker waits for a period of time.
c) Attacker loads a new line (disjoint from the initial lines in Step a)) into this set, which removes the oldest line.
d) Attacker accesses the $(w-1)$ initial lines again. One of the following may occur:
   - All initial lines are still resident in LLC. This means the victim didn't access his line during the waiting period. Thus, in Step c), the victim's line was evicted because it was the oldest line in LRU order.
   - One of the initial lines is not in LLC. This means the victim accessed his line during the waiting period. Thus, in Step c), one of the attacker's initial lines became the oldest one and was evicted[1].

[1]Monitoring LRU state changes in LLC requires removing the line from private cache since private cache hits do not update the LRU state in LLC.

2.2 Prior defenses and their limitations

Randomizing the address mapping in LLC is a popular way to defend LLC attacks. Many randomization works have been proposed [13–15]. However, most of them have already been proven to be insecure or having high overhead [12].

Partition-based methods are more effective than randomization-based methods. **PLCache** [14] uses cache line locking to keep the lines of protected processes always in cache; **NoMo** [6] realizes way-level partition by changing the replacement policy. However, these designs have high overhead and do not scale. **SHARP** [17] prevents LLC attacks by avoiding the creation of "inclusion victims". SHARP is most related to our work. However, it makes weak assumptions, such that the victim's security-sensitive cache lines should always be in his private cache. Our design avoids these assumptions to be more secure. More importantly, most of these early partition works did not consider the LRU attack, and are thus not secure.

Newer cache partition designs such as DAWG [10] cover both Prime+Probe and the LRU attack. However, in most of these works, the security design is applied to all users including security-insensitive users (e.g., DAWG requires reserving cache ways for each security domain).

3 IVCACHE DESIGN

3.1 Threat model

In this work, the targeted attacker is an unprivileged user that can launch himself on the same processor with the victim. Additionally, the attacker should be able to build set conflicts in LLC (e.g., by using the method in [11]). We assume that Hyper-Threading (which allows the attacker and victim to co-locate on the same CPU core) is turned off or unavailable. Then the attacker is only sharing the LLC portion of cache with the victim. We also assume a trusted operating system (OS) as in [10, 14, 17].

We exclude the LLC attacks relying on data sharing, because these attacks can be stopped by disabling data sharing across security domains, or by maintaining multiple copies of a cache line in LLC, as done in [10]. We also exclude leakages caused by bank conflict, memory bus contention, and other minor leakages discussed in [4], since they can be defended by simple hardware changes [3]. Speculative execution-based side channels are not considered as they can be solved by orthogonal works such as [21].

3.2 Basic design

IVcache assumes an inclusive cache for simplicity. However, our design can be slightly modified to defend against non-inclusive cache attacks [18] by instead changing the cache directory structures. Our design principle is the same regardless: we modify how state changes propagate so that the victim's cache accesses are invisible.

We use cache security domains (SD), introduced in [10]. Each SD is a group of one or several processes with mutual trust. Instead of forcing every SD to use protection mechanisms, IVcache allows users to request protected status in cache and only protects these SDs; users which do not request protection are all grouped into a single "unprotected" SD. We achieve this by extending the ISA with an new instruction, which informs cache to tag an SD as protected. Then, cache accesses from protected SDs (i.e. victims) use IVcache's

security mechanisms. Naturally, IVcache attaches the SD id to all the LLC accesses. This mechanism has been previously used, for both performance and security. The SD id can be obtained from the thread/process context. IVcache also requires tagging LLC lines that are owned by a victim, using the SD id. This tagging occurs when a victim brings a line into LLC from memory.

IVcache's philosophy is to ensure that in LLC, a victim's cache access cannot be observed by any attackers. Next, we discuss how to hide the victim's LLC misses (from a Prime+Probe attacker), and the victim's LLC hits (from an LRU attacker), respectively.

3.2.1 ***Defending Prime+Probe attack.*** If a victim's access misses in both the private cache and LLC, and the target LLC set has space for a new cache line. Then the victim is safe and IVcache works same with a normal inclusive cache: load the data from memory, fill it into the private cache and LLC, and send the data to CPU.

If a victim's access misses in LLC, and the target LLC set is full, in traditional caches, the LRU line in the target set will be evicted. However, if the evicted line is an attacker's probe line, the attacker can detect this victim's access. Instead, IVcache makes the victim's access invisible from a Prime+Probe attacker as follows:

- **Step 1: Self-eviction**
  IVcache tries to evict a line which is owned by the victim, if possible (Figure 1(a)).
- **Step 2: Bypass**
  If Step 1 fails (the victim owns no line in the set), then we bypass the cache fill to avoid vulnerable evictions. For loads, the data is sent directly to CPU; for writes, it's directly written to memory (Figure 1(b)).

In this way, IVcache effectively prevents dangerous set conflicts and hides the victim's LLC misses. However, victims may gradually lose their occupancy in LLC due to our design. In Section 3.3 and 3.4 we discuss how to solve this problem.

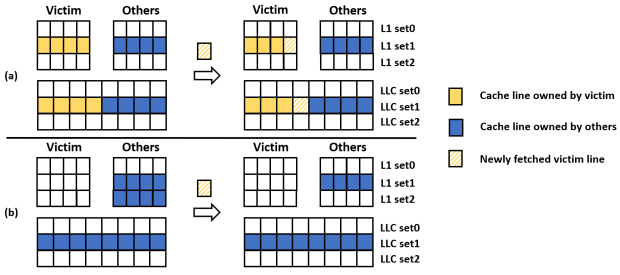


Figure 1: (a) Self-eviction and (b) Bypass mechanism; L1 cache is bypassed in (b) due to cache inclusion.

3.2.2 ***Defending the LRU attack.*** Modern LLCs use LRU policy and its variants (e.g., Tree-LRU), but we have shown that attackers can manipulate these policies to spy on victims. In contrast, uniformly random replacement is leakage free, because the attacker cannot associate the replacement state changes of his cache line with the victim's accesses. However, for some applications, random replacement causes significant performance degradation, e.g., it interacts negatively with the hardware prefetcher. In IVcache, we build a new replacement policy called Random-LRU (RLRU), which holds better performance than random replacement without sacrificing the victim's security.

In RLRU, we only track the ordering of $m \leq w$ newest lines (in LRU order) among w total lines in an LLC set. We refer to these m lines as the *active group*. For the oldest $(w-m)$ lines (*inactive group*), we use uniformly random eviction. The value m parameterizes RLRU between LRU ($m = w$) and total random replacement ($m = 0$). In RLRU, replacement state leakage is only possible when the victim's line is brought into/changed inside the active group. Hence, in IVcache we pin the victim's lines to the inactive group, preventing leakage.

When a victim's access hits in LLC, no replacement state change will be triggered to the target line, i.e. this line stays in the inactive group. This makes RLRU safe since whenever there's eviction, the eviction target is always randomly selected from the inactive group, and this selection is not related to the victim's previous behavior. When a victim's access misses in LLC, if the target line is brought into LLC, we force it into the inactive group. Note that this will not cause eviction and leak information since it only happens when the set is not full. (when the set is full, the LLC fill is bypassed according to Section 3.2.1).

When there is a non-victim LLC access, the replacement state update is similar with normal LRU: the LRU ordering of the active group is updated, and the target line is brought into the active group if not already present. This action may cause the oldest active line to be moved to the inactive group (which may further trigger an inactive group eviction)[2].

3.3 Keeping read-only copies in private cache

While our self-eviction and bypass mechanisms can effectively protect the victim from Prime+Probe, the victim tends to lose cache occupancy. With self-eviction, the victim owns fewer lines (over time) in each LLC set: he cannot obtain a higher residency percentage than he currently has, but his entries may be evicted by contending SDs, lowering his residency percentage.

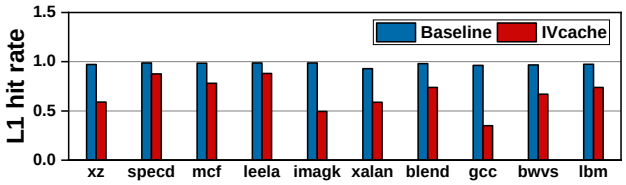


Figure 2: IVcache's effect on the victim's private cache hit rate for different victim benchmarks (shown in X-axis).

Consequently, once the victim is completely evicted from an LLC set, the bypass mechanism is triggered, this LLC set will never be filled again, and the victim enters permanent starvation in both LLC and the private cache (due to cache inclusion). As a result, the victim's private cache hit rate sharply decreases when using IVcache, which significantly degrades performance. To confirm this, we test IVcache with SPEC2017 benchmarks in a 2-core system; for each experiment, we run 2 benchmarks, setting one to be the victim. As shown in Figure 2, compared to the baseline insecure

[2]We evict from the inactive group on each non-victim LLC miss, evicting dummy lines if necessary, to prevent probabilistic Prime+Probe attacks.

cache, IVcache degrades the victim's hit rate by at best, 20%, and at worst, 70%.

To recover the victim's performance, we must relax the bypass mechanism to allow some private cache fills but still keep the victim invisible to the attacker. Our design is to ***still fill the private cache, when LLC needs to be bypassed***. This can help recover some private cache hit rate, even when the victim loses the LLC occupancy. However, filling the private cache but not LLC violates cache inclusion policy and thus creates coherence problems. To avoid this and simplify the design, we fill the private cache with just a read-only copy when LLC is bypassed. Then we only require some small additional logic in cache, as well as one fixed-size *record buffer* per victim (protected SD) near LLC. Each entry of the record buffer (RB) records an address of the cache line that is in private caches but not LLC and the corresponding ids of the private caches that have a copy of this line.

When a victim's access bypasses LLC, the victim's RB is checked: if there is an available slot, the cache line address will be recorded, and the requesting private cache is filled with a read-only copy; if the buffer is full, this access will bypass the private cache, and the data is sent directly to CPU.

For correctness, if *line l* (a line in the private cache) is recorded in the RB[3], a read access to this line can hit the read-only copy in the private cache. However, if there is a write to *line l*, our updated policy takes the following actions: (1) invalidate the copies of this line in all the private caches; (2) send message to LLC to reset the corresponding entry in the RB; (3) forward the write to LLC and LLC will handle the write securely as discussed before. A recorded line may have read-only copies in multiple private caches when it is a shared line among processes in the same SD. When a protected process misses in the private cache and LLC, LLC will first check this SD's RB for the requesting address. If found, LLC will add the id of the requesting private cache to the corresponding RB entry, and a copy of this line will be filled into the requesting private cache; if not, a memory access will be generated.

3.4 LLC active invalidation

Those read-only copies in private caches can only serve reads but not writes. When the victim is bypassing an LLC set, all the writes to this set need to go directly to memory, which can significantly hurt performance. To solve this problem, we need to ensure that when the victim is starving in an LLC set, the victim is able to recover entries and leave the starvation.

In IVcache, the victim's starvation occurs because cache is *reactive*. A cache line is evicted only when a new access causes replacement or the CPU flushes/invalidates the line. Thus, for IVcache, once the victim does not own any lines in an LLC set (reaches bypass state), it is rare for the victim to recover entries (e.g., due to other users' entries being flushed).

Thus, we propose to ***actively invalidate a line from the inactive group in an LLC set*** per c cycles, writing it back to memory if necessary. In this way, we actively give the victim a chance to recover some LLC occupancy, helping it quit the starvation and improve performance. The choice of c is left for the OS; when there are more users claiming to be a victim in the system, c should be set to be shorter. The OS could also increase c temporarily when a victim is joining the system to make some LLC space for it (if the victim's joining is not considered a leakage). Note that similar ideas have been proposed before (e.g., evicts dirty lines early) for improving performance. With a proper c, this design will not hurt non-victim users' performance.

[3]We mark all the recorded private cache lines. This will be discussed in Section 3.5

Table 1: Experiment Configuration.

Parameter	Description
ISA	X86_64
Processor type	8-way out-of-order, no SMT, 2.0GHz 128 ROB entries, 32 load store queue entries
L1-Icache, private	32kB, 64B line, 8-way associative
LI-Dcache, private	32kB, 64B line, 8-way associative
L2 cache, shared	2MB bank per core, 64B line, 16-way associative
Cache polices	LRU, Directory-based MESI Coherence Protocol
Memory	8GB, 1 channel, 8 ranks, 8 banks
Hardware prefetcher	Stride prefetcher

3.5 Storage Overhead

IVcache only requires negligible extra storage in cache: first of all, for each cache line in private caches, 1 bit is needed to track whether this line is a read-only copy or not. Besides, we also need to track the owner SD of each line in LLC to know if it belongs to a victim. However, there can be many SDs running in the same time, causing the SD id space to be relatively large. This can make it very expensive to record the owner's id of each cache line. Therefore, in IVcache, we limit the number of victims running in the system. An extra storage called id buffer is added near LLC to record the SD id of each victim. By limiting the number of victims, we only need to add several 1-bit flags in each cache line, and each flag is mapped to an entry in the id buffer, instead of storing SD ids in each line. We leave the quantitative limitation of victims configurable to the processor designer.

As mentioned, one RB is required for each victim. From the result of the sensitivity study on the buffer size, giving 40 entries for each victim is enough for keeping relatively good performance. Then the storage of the RBs is less than 1% of LLC, when the quantitative limitation of victims is set to be the number of physical cores.

Algorithm 1: Square-and-Multiply exponentiation

Input: base b, modulo m, exponent $e = (e_{n-1}...e_0)_2$
Output: $b^e \bmod m$
r $\leftarrow$ 1
for $i = n - 1; i >= 0; i - -$ **do**
  $r \leftarrow r^2 \mod n$
  if $e_i == 1$ **then**
    $r \leftarrow r * b \mod n$
return r

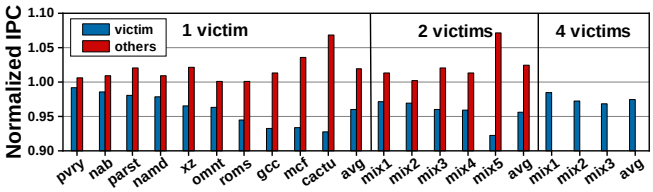
4 EVALUATION

4.1 Configuration

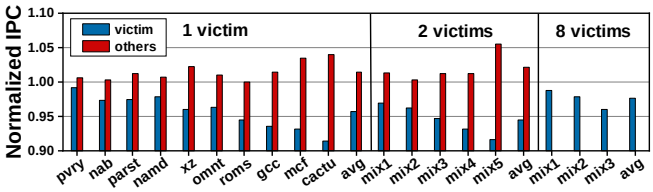
To evaluate IVcache, we modify the Gem5 [2] cycle-level full-system simulator. The system setup and the baseline cache parameters are shown in Table 1. Table 2 shows the related parameters of IVcache that are used in our evaluation. These parameters are the optimized ones from experimental results.

4.2 Defending real-world attacks

To evaluate the security of IVcache, we test it against a real-world Prime+Probe attack on GnuPG. We did not run the LRU attack



(a) Normalized IPC of 4-core systems



(b) Normalized IPC of 8-core systems

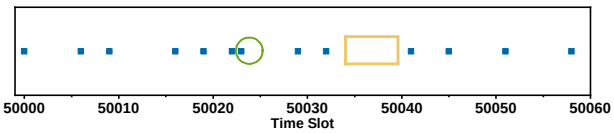
Figure 3: Normalized Performance overhead of IVcache when fixing 1, 2, or all running benchmark(s) as the victim(s).

Table 2: IVcache Parameters.

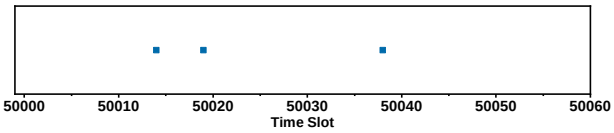
Record Buffer Entry	Invalidation Rate c	Size of Active Group
40	110 cycles	9

because there is no public code base for it and our defense on it is quite simple.

The square-and-multiply algorithm [7] is found in GnuPG version 1.4.13 for multiple ciphers; leaking the exponent e of this algorithm leaks the decryption key. As shown in Algorithm 1, in each iteration of the *for* loop, the executed instruction pattern is related to one bit of the exponent. When the current bit is 0, the executed instruction pattern is *square-reduce*; when it's 1, the pattern is *square-reduce-multiply-reduce*. Thus, for different bit values, the time between two consecutive accesses to the square instruction is different: if the current bit is 1, the distance is longer than when the current bit is 0. By tracing the victim's accesses to the LLC line that the square instruction is mapped to, the attacker can learn each bit of the exponent. Since the attacker does not know the physical address of the square instruction, he needs to trace the access pattern of each set in LLC to find the set with the access pattern that is close to the pattern of square instruction fetching. To achieve this, the attacker primes an LLC set every 5000 cycles (a time slot).



(a) Prime+Probe result on the baseline cache



(b) Prime+Probe result on IVcache

Figure 4: Prime+Probe attack results.

Figure 4(a) shows a fragment of an LLC set's access pattern in 60 time slots. This set is the only one whose pattern is similar with the pattern of square instruction fetching. Each dot represents an access to this set, detected by the attacker in a time slot. Most distances between two consecutive dots are either one of two distinct distances, except two dots. The rectangle highlights an access missed during the attack; the circled dot is a false access caused by the attack noise (e.g., OS noise, speculative execution, etc). Noise is usually eliminated by multiple runs. In this trace, the leaked key pattern is: 1010010(10/01)011, where the parentheses mean uncertainty due to the noise.

Figure 4(b) shows the result of IVcache. Once the attacker primes the target LLC set (fully occupies it), the victim will bypass this set. Thus, the attacker cannot cause conflicts with the victim and observe the victim's accesses. The detected spots in the trace are from the attack noise, as explained earlier.

4.3 Performance evaluation

We also test IVcache's performance in Gem5. We use 10 SPEC2017 benchmarks that have either high L1 hit rate, low L1 hit rate but high LLC hit rate, or low L1 and LLC hit rate. In each experiment, we first run 1 billion instructions to warm up the cache, then we run another 1 billion instructions to collect statistic data. We test IVcache on 4-core and 8-core systems. We run n benchmarks concurrently, where n is the number of CPU cores. In each test, we fix 1, 2, or all benchmark(s) as the victim(s). To choose the non-victim benchmarks, we randomly select 10 combinations so that the total number of running benchmarks is the core size, n. For example, on an 8-core system with 2 victims, say `mcf` and `gcc`, we run 10 tests where the other 6 benchmarks are randomly selected. Thus, in Figure 3, the bars are averages among the 10 tests; each group of bars refers to fixed victim(s), shown in the X-axis. We run each victim benchmark in a different protected SD, and run all the non-victim benchmarks in one unprotected SD. Figure 3 shows the performance of the victim and non-victim benchmarks, normalized to the baseline insecure cache.

As shown in Figure 3, IVcache has good performance in terms of IPC, giving very little performance penalty to victims and little performance benefit to other benchmarks. The max performance penalty is only 7.8% for 4-core systems, and 8.6% for 8-core systems, and the average is only 3.9% for 4-core systems, and 4.4% for 8-core systems. `cactuBSSN` and `mcf` have the worst performance, as a victim. This is because these workloads have relatively lower L1 cache hit rate; their performance highly relies on LLC utilization. However, as our optimization method actively makes LLC space for the victims, the overheads still stay acceptable. IVcache shows very subtle overhead when all the SDs on the system are victims. This is because 1) in this case all the SDs contend for LLC resource in a relatively balanced way, maintaining high LLC performance, and 2) the utilization of RBs helps keeping high L1 cache hit rate.

Figure 5 (left) shows the average normalized L1 cache hit rate in different systems. Comparing Figure 5 (left) with Figure 2, we see that our performance optimizations, keeping read-only copy and active invalidation, can almost recover victims' L1 cache hit rate. This is very important for IVcache: in a normal inclusive cache, after missing in L1 cache, the victim's access still has a high probability

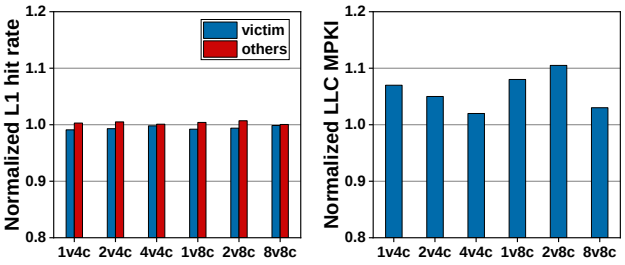


Figure 5: Average cache performance. N_1VN_2C means the systems with N_1 victim(s) and N_2 cores.

to hit in LLC. However, in IVcache, if LLC is bypassed, the access will go directly to memory, which results in over 100 times higher latency than hitting in L1 cache. Additionally, IVcache barely affects the L1 cache hit rate of unprotected benchmarks.

IVcache also affects the LLC performance in two ways: (1) self-eviction and bypass mechanisms increase the victim's LLC misses; (2) forced victim starvation gives unprotected SDs more chance to use LLC, reducing their LLC misses. As we can see in Figure 5 (right), although IVcache shows an increase on the LLC MPKI in each scenario, in most cases this increase is less than 10%. For all-victim systems, the increase is less than 5%.

4.4 Comparison with DAWG

DAWG is a cache partition design proposed in 2018. DAWG forces its security mechanism to every SD: it applies way-partitioning among SDs; given an LLC set, an SD can only use a portion of it. We modified Gem5 to evaluate the performance DAWG: as the associativity of our LLC is 16, we assign every SD 4 ways of each LLC set in 4-core systems, and 2 ways of each LLC set in 8-core systems. As shown in Table 3, the performance degradation for victims in DAWG and IVcache are very close; however, DAWG can also cause performance overhead for non-victim applications (up to 5.3%). This unfair overhead is avoided in IVcache.

Table 3: The normalized IPC of DAWG and IVcache; for each N_1/N_2, N_1 is the average IPC of the victims, and N_2 is the average IPC of the non-victim applications.

	4-core			8-core		
	1-V	2-V	4-V	1-V	2-V	8-V
IVcache_avg	0.961/1.019	0.956/1.024	0.975	0.957/1.024	0.945/1.021	0.977
IVcache_min	0.928/1.002	0.922/1.000	0.968	0.914/1.002	0.916/1.001	0.960
DAWG_avg	0.953/0.959	0.956/0.959	0.953	0.942/0.947	0.940/0.947	0.945
DAWG_min	0.858/0.861	0.853/0.862	0.855	0.847/0.854	0.844/0.850	0.845

4.5 Sensitivity study on record buffer size

To learn how the size of RB affects the victim's IPC, we choose the same 10 benchmarks as in Section 4.3, and run each of them on a 4-core system with 3 other non-victim benchmarks. The result is shown in Figure 6: when each RB has less than 30 entries, increasing buffer size can clearly benefit the victim's IPC. However, when it has over 40 entries, the IPC becomes stable. Thus, we give each RB 40 entries, introducing less than 1% storage overhead to LLC.

5 CONCLUSION

We proposed IVcache, a comprehensive defense mechanism for both the original LLC attack and the new variant. IVcache makes

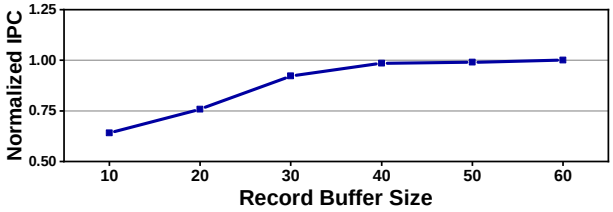


Figure 6: The normalized IPC with different record buffer sizes.

the victim's behavior invisible to attackers by modifying the way state changes are handled in cache. To make IVcache practical, we also provided two optimizations. We tested IVcache against a real-world attack, which showed that IVcache could effectively defend the attack. We also used SPEC2017 workloads to evaluate IVcache's performance degradation. The result showed that IVcache has negligible performance impact.

6 ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation #2011146, #1910413, #1725657, #1738783, and #1718080. The authors thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] 2017. SPEC CPU 2017, https://www.spec.org/cpu2017.
[2] Nathan Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (08 2011), 1–7.
[3] Thomas Bourgeat et al. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *MICRO '52*. 42–56.
[4] Shuwen Deng et al. 2020. A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks. In *ASPLOS'20*. 683–697.
[5] Ghada Dessouky et al. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security'20*. 451–468.
[6] Leonid Domnitser et al. 2012. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Trans. Archit. Code Optim.* 8, 4 (2012), 1–21.
[7] Daniel M. Gordon. 1998. A Survey of Fast Exponentiation Methods. *J. Algorithms* 27, 1 (1998), 129–146.
[8] Daniel Gruss et al. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security'15*. 897–912.
[9] Mehmet Kayaalp et al. 2017. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *DAC'17*. 1–6.
[10] Vladimir Kiriansky et al. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO'50*. 974–987.
[11] Fangfei Liu et al. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *S&P'15*. 605–622.
[12] Antoon Purnal et al. 2019. Advanced profiling for probabilistic Prime+ Probe attacks and covert channels in ScatterCache. *arXiv preprint arXiv:1908.03383* (2019).
[13] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO'51*. 775–787.
[14] Zhenghong Wang et al. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ISCA'07*. 494–505.
[15] Mario Werner et al. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security'19*. 675–692.
[16] Wenjie Xiong et al. 2020. Leaking information through cache LRU states. In *HPCA'20*. 139–152.
[17] Mengjia Yan et al. 2017. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *ISCA'17*. 347–360.
[18] Mengjia Yan et al. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *S&P'19*. 888–904.
[19] Fan Yao et al. 2018. Are coherence protocol states vulnerable to information leakage?. In *HPCA'18*. 168–179.
[20] Yuval Yarom et al. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security'14*. 719–732.
[21] Jiyong Yu et al. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO'51*. 954–968.