New Cache Attacks and Defenses

by

Yanan Guo

Bachelor of Sciences, Xidian University, 2018

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Yanan Guo

It was defended on

February 11, 2020

and approved by

Jun Yang, Ph.D., Professor

Department of Electrical and Computer Engineering

Samuel Dickerson, Ph.D., Assistant Professor

Department of Electrical and Computer Engineering

Jingtong Hu, Ph.D., Assistant Professor

Department of Electrical and Computer Engineering

Thesis Advisor: Jun Yang, Ph.D., Professor

Department of Electrical and Computer Engineering

New Cache Attacks and Defenses

Yanan Guo, M.S.

University of Pittsburgh, 2020

The sharing of last-level cache (LLC) among different physical cores makes cache vulnerable to side channel attacks. An attacker can get private information about co-running applications (victims) by monitoring their accesses. Cache side channel attacks can be mitigated by partitioning cache between the victim and attacker. However, previous partition works make the incorrect assumption that only the victim's cache misses are visible to attackers.

In this work, we provide the key insight that both cache hits and cache misses from the victim are vulnerable. For a cache hit, although it does not affect the existence state, it can still change the replacement state and coherence state, which can also leak information to attackers. Based on this, we propose Invisible-Victim cache (IVcache), a new cache design that can mitigate both traditional LLC attacks and the new variants. IVcache classifies all processes as protected and unprotected. For accesses from protected processes, IVcache handles state changes in a slightly different way to make those accesses absolutely invisible to any other processes. We evaluate the defense effectiveness and performance of IVcache in the gem5 simulator. We show that IVcache can defend against real-world attacks, and that it introduces negligible performance effect to both protected and unprotected processes.

Table of Contents

List of Tables

List of Figures

Preface

This thesis is based on one of my paper work, submitted to Usenix ATC 2020. Cache side channel attacks are becoming more and more potent and researchers have proposed multiple secure cache designs. However, recently discovered leakages in cache can be utilized by attackers to defeat previous secure cache designs. Following the recommendation of my advisor Dr. Yang, I proposed a new design to provide strong security in cache which can defend both traditional cache attacks and the new variants.

Here I really want to give my appreciation to my advisor Dr. Yang and my lab-mate Andrew Zigerelli. My initial design had multiple flaws and they spent a lot of time helping me fixing and improving the design. Dr. Yang also provided me a chance to attend a top conference so I could get some advise about my design from other researchers and further improve it. The most difficult part of this work is implementing the design in the simulator and evaluating both performance and security, which costed almost 2 months. During that time, Dr. Yang and Andrew helped me figure out the best way to implement the design and fix a lot of errors in the code. Without their help, I couldn't finish this work.

1.0 Introduction

Side channels give attackers the opportunity to learn private information without accessing it directly. Recently, cache timing side channel attacks, or cache attacks for short, have been demonstrated to be extremely potent [3, 4, 8, 16, 19, 20, 24, 31, 52]. Different cache behaviors, such as hits and misses create significant timing differences to the execution of an instruction. Attackers are able to use this side channel information to monitor other processes' cache behavior and then infer private information (cryptographic keys for example).

For several reasons, cache is a popular target for attackers. First, the last-level cache (LLC) is shared among multiple physical cores, which is more convenient for the attacker to leverage than other hardware units such as execution port [5], line-fill buffer [41], etc. Even though those hardware units are also vulnerable to side channel attacks, they are only shared between processes that are co-running on the same physical core (e.g. using Hyper-Threading). Second, caches can leak information at very fine memory granularity, attackers can obtain information at cache line level, which is typically only 64 bytes. Third, different cache behaviors are easily distinguishable to attackers. Even for LLC, the time difference between a cache hit and miss can still be more than 50 cycles, which yields clear information to attackers.

Initial cache attacks only focus on leaking keys of cryptographic algorithms [32, 23, 53]. More recently, cache attacks are deployed to leak more general information. For example, cache template attack [18] leverages cache timing leakage to implement a keystroke logger; Flush+Reload attack [16, 9] is utilized to recover web search history [21] and to break address-space-layout-randomization (ASLR) [14, 15, 25].

Cache attacks can work on both the private cache and shared cache. However, working on the private cache (e.g. L1 cache) requires the victim and attacker to run on the same core. In contrast, attacks on the shared cache (e.g LLC) are much more powerful since the attacker can be on a different core than the victim. Many different architectural solutions have been proposed to mitigate timing attacks on LLC [45, 10, 28, 34, 46, 30, 49, 26], which can be classified into randomization-based solutions and partition-based solutions. A popular way to implement a randomization-based defense is creating a random mapping between cache lines and cache sets [45, 30, 43, 36]. Unfortunately, such mechanisms only work for conflict-based attacks, and some are proven insecure [37, 35].

The root cause of cache attacks is that the victim's execution makes changes to cache states, which are visible to attackers. Thus, to make a rigorous defense mechanism, we should endeavor to 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 separate in cache, the attacker cannot observe the victim's access patterns anymore. However, previous partition designs make weak assumptions about the attacker's strength [45, 49, 10]. It was falsely claimed that only the victim's cache misses are dangerous. Under this assumption, as long as the victim's access hits in cache, the victim is safe. However, cache state not only comprises existence state, but also coherence state and replacement state. Although the victim's cache hit does not affect the cache line's existence state, it can affect the line's coherence and replacement state, which also leaks secrets to a "smart" attacker.

In this paper, we demonstrate how replacement state and coherence state can be utilized to leak information. Based on this, we propose Invisible-Victim cache (IVcache). IVcache is a novel cache partition design that can completely prevent LLC attacks, by making the victim's cache behavior "invisible" to all other processes. IVcache targets both traditional cache attacks and the new variants. We enumerate all situations a victim's access could face in cache, and design how IVcache handles the changes on existence, replacement and coherence states in each situation to make the access invisible. Our design trades some performance for security, but we also develop two optimization mechanisms to salvage the performance of the victim. We aim to keep a performance balance between the victim and all other concurrent processes. We implement IVcache in gem5 simulator [6] and test it against real-world attacks. The result shows that IVcache effectively prevents these attacks. In addition, we run workloads derived from SPEC2017 benchmark to evaluate IVcache's impact on performance. We show that IVcache causes negligible overhead to the victim and and slight performance benefit to the co-running non-victim processes.

Our contributions can be summarized as follows:

- The insight that, not only the existence state of cache lines can be used to leak information, but also the coherence state and replacement state.
- The design of IVcache, which is an effective secure cache design that targets both traditional and the new variants of LLC attacks.
- A new proof-of-concept cache attack based on transient coherence states.
- The implementation of IVcache in the gem5 simulator, and the security and performance evaluation of IVcache.

2.0 Background and Related Work

In this section, we provide the necessary background on cache architecture, cache side channel attacks, and previous defense mechanisms.

2.1 Caches

In modern CPUs, memory hierarchy typically consists of cache, main memory, and disk storage. Cache is used to keep the data and instructions that are likely to be used in the future near CPU. Using cache can efficiently avoid the long access latency and huge energy consumption of off-chip main memory. Cache is a hierarchical architecture which usually consists of 2 to 4 layers. Each CPU core has it's private one or two cache layers which are the lowest levels in cache hierarchy. Each core's private cache is further split to instruction cache and data cache, and both of them are indexed by virtual address. LLC is much larger than private cache and is usually shared among all physical cores. In most modern CPUs, LLC is inclusive, i.e., any data stored in lower level caches have to be present in LLC. LLC is indexed by physical address which is easier for different CPU cores to share data.

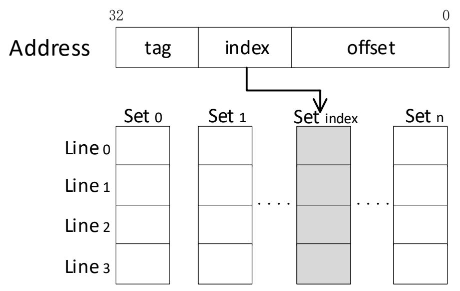

Figure 1: An example of 4-way set associative cache.

Cache is typically organized in cache sets and each cache set consists of several cache lines (i.e., cache ways), which are the units of allocation and transfer down the cache hierarchy.

As shown in Figure 1, cache sets are indexed by some specific bits of data address. And for cache lines that are mapped to to the same set, each of them can take any free space in the set. When a set is full, replacement policy is used to decide which line will be removed when a new line is coming into the set (e.g. LRU, MRU). This is also an important handle for cache side channel attacks. Each line in cache has a coherence state(e.g. shared, modified) which is used for the data sharing among multiple cores.

2.2 Traditional LLC Attacks

Traditional LLC attacks use the existence state changes of cache lines to leak secrets. The primary principle is that attackers can deduce whether a specific cache line is in cache or not, based on the access latency. These attacks can be further classified in conflict-based attacks and flush-based attacks.

2.2.1 Conflict-based Attacks

In conflict-based attacks, attackers must take the following steps:

1. Fill the target LLC set with his own cache lines (*probe lines*) to ensure the target victim line is evicted.
2. Wait (allow the victim to possibly load the target line).
3. Access each probe line to determine whether it is still resident in cache.

If any probe lines are removed, the victim likely loaded the target line back to LLC during the waiting period. If all probe lines are still resident, the victim didn't access his line (e.g. Prime+Probe attack [31]). In Step 3, the attacker may instead directly access the victim's line if it's shared with the attacker. Then, if the line is resident, the victim most likely accessed this line during the waiting period. If it's not, then the victim didn't access it. (e.g. Evict+Reload [18]).

By repeating the previous steps, the attacker may learn the victim's access patterns and infer some related secrets. Algorithm 1 shows an example victim code. Each key bit is related to the taken branch direction in each *for* loop iteration. To leak the key, the attacker monitors the LLC sets for instruction A and B. If the attacker detects an access to instruction A's set, the key bit equals 0; if there's an access to instruction B's set, the key bit is 1. By repeating these steps the attacker can leak the whole key.

Algorithm 1 Victim Code Example

```
Input: private key $key[128]$
1: for $i$ from 127 down to 0 do
2:     if $key_i == 0$ then
3:         instruction A;
4:     else
5:         instruction B;
```

2.2.2 Flush-based Attacks

One representative of flush-based attack is Flush+Reload attack [16]. Similar with Evict+Reload, in Flush+Reload the attacker also directly accesses the victim line in Step 3; however, in Step 1, the attacker removes the victim's line from cache by using the `clflush` instruction instead of building conflicts. [1]

2.3 Prior Secure Cache Architectures

Prior works proposed defenses on LLC attacks. However, they have one or more deficiencies, which we briefly discuss here.

2.3.1 Randomization-based Mitigations

Randomizing the address mapping in caches is a popular way to defend conflict-based cache attacks. **RPCache** [45] and **Newcache** [30] are table-based random mapping mech-

[1] We exclude attacks that can be fixed by subtle changes (e.g. Flush+Flush can be prevented by making the execution time of `clflush` constant [17]).

anisms. These work well on small private caches. However, for defending LLC attacks, the table size becomes several mega bytes, which is impractically large. In **Time-secure Caches** [43], **CEASER** [36] and **ScatterCache** [47], new mapping functions are used instead of a table. Time-secure Caches uses the memory address and process id as the input to its mapping function. CEASER uses a dynamic feistel network to create randomized set indices. ScatterCache provides a security-domain dependent cache mapping based on cryptography. However, these mechanisms either are proven to be insecure [37, 35], or introduce significant overhead [30, 45].

Thus, randomization methods are not yet both secure and keeping low cost. Further, pure randomization does not distinguish between security sensitive and insensitive applications, so the overhead is felt by each user on the platform. This is not fair to security insensitive users. In this work, we instead offer security as a service that the hardware can provide if the user requests it, rather than forcing it to every user.

2.3.2 Partition-based Mitigations

Partition-based methods are considered to be more effective than randomization-based methods. Once there's absolute isolation between processes, attackers cannot gain information about other processes. **PLCache** [45] uses cache line locking to keep the lines of protected processes in cache. However, this method penalizes unprotected processes' performance. **NoMo** [11] realizes way-level partition by changing the replacement policy, and **Catalyst** [29] utilizes Intel CAT [1] to implement isolation between security domains. However, these are not flexible. **DAWG** [27] proposes strict isolation in cache, but it requires significant OS modification. **SHARP** [49] prevents conflict-based attacks by avoiding the creation of "inclusion victims". SHARP causes negligible performance overhead; however, it makes weak assumptions, such that the attacker's probe addresses should always be in his private cache (since the time between consecutive eviction steps is very short). This assumption is not true because a "smart" attacker can always build conflicts to remove all probe lines from his private cache but still keep them in LLC. This renders SHARP insecure.

2.4 New LLC Attacks

As mentioned, it is very important for a partition-based mechanism to provide absolute isolation between the victim and attacker. Most previous partition works [45, 49, 11, 29] use the assumption that as long as the victim's accesses do not miss in LLC and change the existence state of the target cache line by bringing it into LLC, it is secure. This is true only for traditional LLC attacks (Section 2.2), since traditional attackers only monitor the existence state. However, cache state not only consists of existence state, but also replacement and coherence states. Changes that the victim leaves in any of these states may leak information. Thus, only making existence state assumptions weakens the security guarantee; "smart" attackers working outside the assumptions can defeat the previous defenses. Next, we show how attackers can utilize the replacement and coherence state to leak information.

2.4.1 Replacement State-based LLC Attack

Cache hits can change the replacement state (e.g. LRU state) of the accessed line. LRU state is a relative ordering among all cache lines in a set. Thus, the attacker can know that the victim accessed a victim-owned line in the set by checking the changes on the LRU ordering of the attacker-owned probe lines. Very recently, LRU state changes were used in cache attacks [48, 27]. LRU-based attacks require the attacker to own $(w-1)$ lines in the cache set and victim to own 1 line, where w is the associativity. The attacker can accomplish this if he has enough information about the LLC mapping. The basic attack repeats the following steps:

1. Victim accesses his line in LLC, and then the attacker accesses his $(w-1)$ *initial lines.* After this, the victim's line will become the oldest one in this set.
2. Attacker waits for a period of time.
3. Attacker loads a new line (disjoint from the initial lines in Step 1) into the set, which removes the oldest line.
4. Attacker accesses the *(w-1)* initial lines again. One of the following may occur:

a. All initial lines are still resident in LLC. This means the victim didn't access his line during the waiting period. Thus, in Step 3, the victim line was evicted because it was the oldest line in LRU order.

b. One of the initial lines is not in LLC. This means the victim accessed his line during the waiting period. Thus, in Step 3, one of the attacker's initial lines became the oldest one and was evicted[2].

2.4.2 Coherence State-based LLC Attack

Here we use the MESI coherence protocol [39] as an example to explain the coherence-based attack in inclusive caches. In MESI, a read access which hits in LLC may actually trigger different cache behaviors, depending on the cache line's coherence state. For example, if a requesting CPU reads a line and hits Shared (S) state in LLC (this means this line is read-only in LLC and multiple private caches), the data in LLC will be sent to the requesting CPU. However, if the target cache line in LLC is in Exclusive (E) state (this means there's an exclusive copy of this line in one private cache), the data in LLC is potentially stale. In this case, LLC will fetch the data (guaranteed to be up-to-date) from the owner private cache before responding to CPU. Previous research [51] found that the LLC hit latencies of a line in E and S state are distinguishable. Thus, by monitoring different LLC hit latencies, the attacker can learn the victim's access patterns. The attack is shown in Figure 2. The attacker controls two threads, termed spy and trojan, which are launched to different cores. Further, the target cache line must be shared among the victim, spy, and trojan threads. Then, the attacker monitors the victim's accesses by repeating the following steps:

1. Trojan evicts the target cache line to memory by building conflicts, and then fetches this line back to cache. After this, the state of this line will be E in LLC.
2. Attacker waits for the victim's behavior. If the victim accesses this line, the state will be changed from E to S.
3. Spy accesses the target line and measures the access latency, and then determines the state is S or E.

[2]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.

a. It's S, the victim accessed this line (Figure 2(a)).

b. It's E, the victim did not (Figure 2(b)).

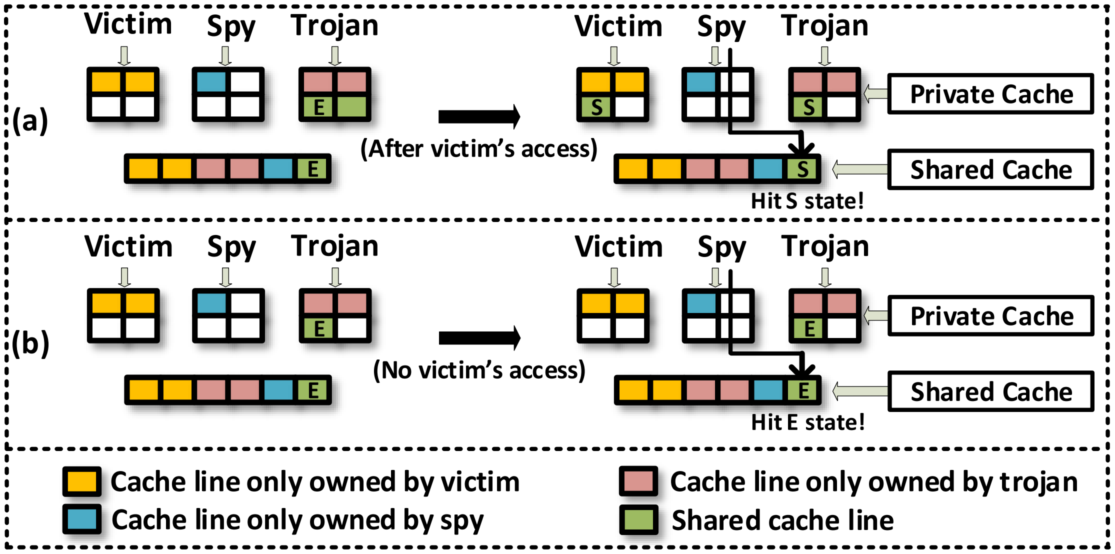

Figure 2: State changes of shared line, w.r.t. victim's behavior.

2.5 Challenges and Our Goal

In this work, we aim for an efficient partition-based mitigation method to absolutely prevent both traditional existence state-based LLC attacks and the new variants. Additionally, because security usually comes at a performance cost, we must not penalize the performance for users who do not need security. Thus, we should offer security as a service to users. Still, this security should have as little overhead as possible.

The main challenge of building the partition is to maintain a balanced and fair resource allocation between the protected and unprotected processes. Kiriansky et al. proposed DAWG [27] which claimed to provide strong isolation between security domains in cache. However, it ignores questions about allocating resources efficiently. Ferraiuolow et al. proposed Lattice Priority Scheduling (for secure memory controllers). This design gives higher priority to unprotected processes for fairness, which is similar to our design principle. However, our design is more flexible and efficient.

2.6 Threat Model

The focus of this work is defending against both traditional LLC attacks and the new variants. The attacker is an unprivileged user that can launch himself on the same processor as the victim. We assume that Hyper-Threading (which allows the attacker and victim to co-locate on the same CPU core) is turned off or unavailable; thus, the attacker is only sharing the LLC portion of cache with the victim. We assume that neither the attacker nor the victim must be pinned to a specific physical core; context switch may cause the attacker to run on the core that the victim used before, and vice versa. We assume an unmodified OS: we do not require OS changes but we require the OS to be always functionally correct.

We assume that for conflict-based attacks, attackers can efficiently build set conflicts. Most LLCs are indexed by physical address. Although the attacker cannot typically control the physical address of his data, there are workarounds. For instance, the OS may allow users to request using 2MB pages instead of 4KB pages to reduce TLB misses. Using large pages enables cache set indexing to be more controllable by the attacker. This is because in the virtual address for large pages, set index bits likely overlap with page offset bits. Thus, the attacker can influence the set index by just choosing an appropriate virtual address. Further, the attacker should have some knowledge about the relation between the victim's secrets and access patterns, so he can utilize the victim's cache behavior to infer secrets such as cryptographic keys.

We exclude minor leakages caused by bank conflict and MSHR contention since they are efficiently defended by previous research [7].

3.0 IVcache Design

Our design, IVcache, assumes an inclusive cache for simplicity: all data in a core's private cache are also in LLC. However, our design can be easily modified to defend against non-inclusive cache attacks [50] 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.

IVcache allows a process to require protected status in cache. This can be achieved by extending the ISA with an appropriate instruction, which informs cache to tag this process as protected. Then, protected processes' cache accesses will use IVcache's security mechanisms. Naturally, IVcache attaches the process id (PCID) to LLC accesses. This mechanism has been previously used, for both performance balance and security [43]. PCID can be obtained from supported TLBs during address translation.

IVcache requires tagging LLC lines that are owned exclusively by each victim, using the victim's PCID. This tagging behavior happens when the victim brings a line into LLC. Tagging victims' lines needs extra storage in LLC. However, in Section 5, we show that this storage overhead is negligible.

IVCache's philosophy is that in LLC, protected processes (victims) only make state changes that an attacker cannot monitor. Next, we discuss how to handle the victim's changes on existence, LRU, and coherence states respectively to make the victim invisible.

3.1 Existence State Changes

3.1.1 Victim's Access Misses When the LLC Set has Vacancy

If a victim's access misses in both the private cache and LLC, but the target set in LLC has space for a new line, then IVcache works as a normal inclusive cache: IVcache loads the data from memory, fills it to LLC and the private cache. As mentioned, this line will be tagged in LLC using the victim's PCID. However, this access changes the existence state of the line, thus it can be detected by a Flush+Reload attacker. To handle this, IVcache disallows other processes to hit any *tagged* victim lines: when a process is accessing an LLC line tagged and exclusively owned by another process (a victim), IVcache invalidates this line and triggers a memory access as an LLC miss. After this, this line is in LLC again and any process is allowed to hit it since it's not a tagged line anymore. In this way, although the victim changed the existence state of the line, by disallowing the hit from other processes, the victim's state change is invisible to any other process.

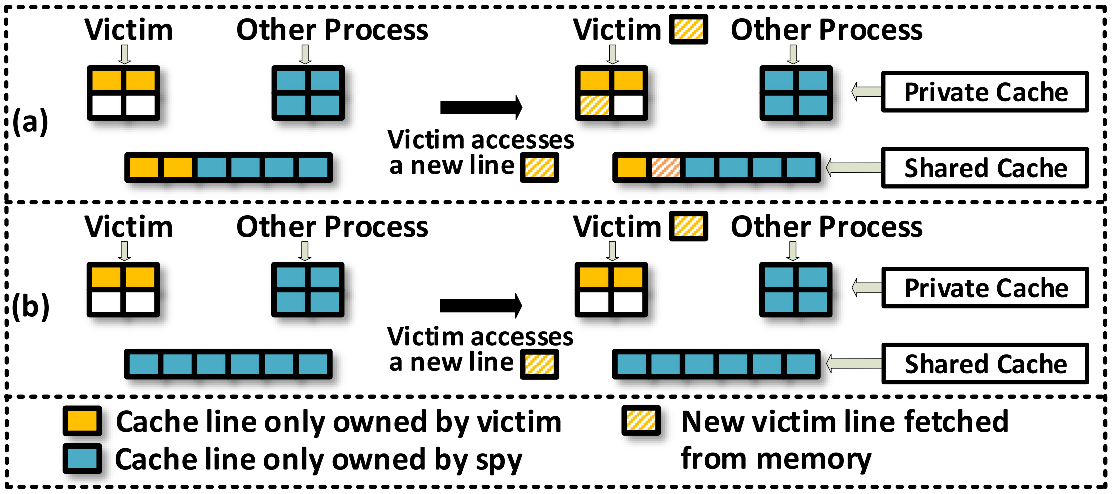
(a)
Victim
Other Process
Victim accesses a new line
Victim
Other Process
Private Cache
Shared Cache
(b)
Victim
Other Process
Victim accesses a new line
Victim
Other Process
Private Cache
Shared Cache
Cache line only owned by victim
Cache line only owned by spy
New victim line fetched from memory

Figure 3: Self-eviction and Bypass.

3.1.2 Data Fetch Misses When the LLC Set is Full

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

- **Step 1: Self Eviction**

  IVcache evicts a line owned by the victim (in LRU order), if possible (Figure 3(a)).

- **Step 2: Bypass**

  If Step 1 fails (the victim owns no line in the target set), then we bypass the cache fill to avoid vulnerable evictions. For loads, the data is sent directly to CPU. For writes, data is directly written to memory (Figure 3(b)).

In this way, IVcache effectively prevents set conflicts and hides victim's changes on existence state. However, victims may gradually lose their occupancy in LLC due to our design. In Section 3.4 and 3.5 we discuss how to solve this problem.

3.2 LRU State Changes

Modern LLCs use LRU policy and its variants (e.g. Tree-LRU), but we've shown that attackers can manipulate these policies to spy on victims. In contrast, uniformly random replacement is leakage free, because the attacker cannot associate his replacement state changes with the victim's accesses. However, in some applications, random replacement causes tremendous performance degradation, e.g. it interacts negatively with the hardware prefetcher. In IVcache, we build Random-LRU replacement policy (RLRU), which recovers performance 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 a set. We refer to these m lines as the *active group*. For the oldest $(w - m)$ lines (*inactive group*), we use uniformly random eviction[1]. 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 victims 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

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

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 3.1).

When there's 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).

3.3 Coherence State Changes

Because a victim's access to a shared line in LLC may cause coherence state changes (which can be observed by the attacker), IVcache avoids changing the coherence state in this scenario. Once a cache line is shared between the victim and the attacker, we assume that neither has write permission to this line. This assumption is reasonable, as we explain later in this section. Based on this, when the victim's access (read-only) hits in LLC, the state of this line can be either E or S. If it's S, as shown in Figure 4(a), IVcache just follows the MESI protocol i.e. keeps the state in S and fills the victim's private cache with a shared copy. If it's E (a.k.a LLC coherence hit), as shown in Figure 4(b), IVcache does not change the state of this line in LLC as done normally in Figure 2, since this behavior can be monitored by the attacker. Instead, IVcache keeps the state in E (does not update the LLC copy), fetches the data from the owner private cache and then returns it directly to the victim core without filling the victim's private cache. Thus, after the victim's access, there's no coherence state change to the target line. Not filling the victim's private cache may cause performance degradation due to lower private cache hit rate. However, prior research shows that LLC coherence hits happen very rarely [40]. In Section 5, we show that IVcache maintains acceptable performance. If the target cache line is not a shared line between processes, IVcache handles the coherence state changes normally.

When the victim's access hits S state, filling the victim's private cache increases the number of sharers (recorded in the LLC directory) by 1. This is a potential leakage if the attacker can learn that the number of sharers was incremented: when the attacker writes this line after the victim's access, the write will take longer than normal (more sharers to invalidate); he can thus learn that the victim accessed this line and increased the number of sharers. However, this case is not possible according to our assumption: the victim and attacker only share read-only data. A security conscious user will not explicitly give write access to other processes. However, the victim and attacker may still share read-only data passively due to OS optimizations. Examples include the attacker using the same shared library as the victim, and the attacker forcing the sharing by page deduplication [44, 22, 33]. For shared libraries, neither the victim nor attacker can write this data. For page deduplication, if the attacker or victim writes the data, a page fault will occur, triggering copy-on-write mechanism.

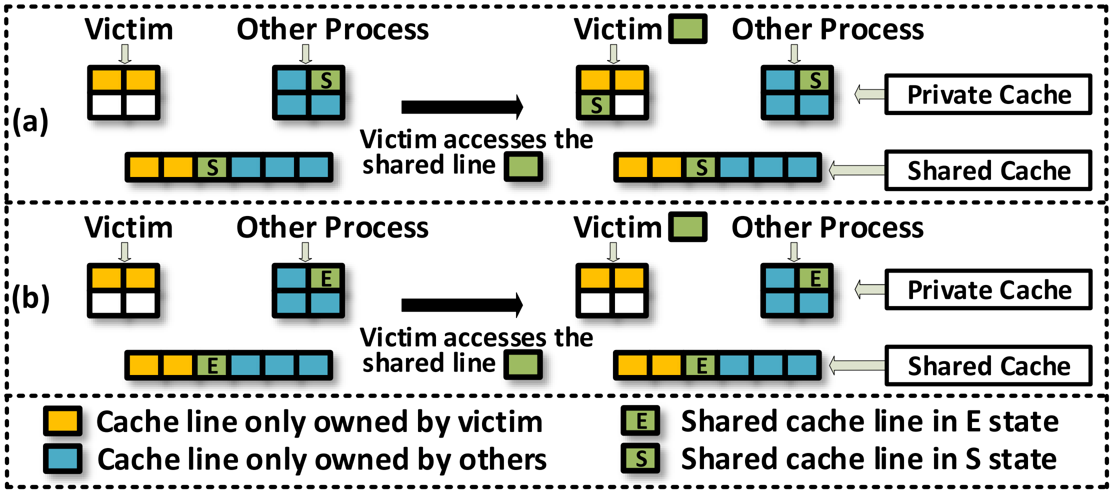
(a)
Victim
Other Process
Victim accesses the shared line
Victim
Other Process
Private Cache
Shared Cache
(b)
Victim
Other Process
Victim accesses the shared line
Victim
Other Process
Private Cache
Shared Cache
Cache line only owned by victim
Cache line only owned by others
Shared cache line in E state
Shared cache line in S state

Figure 4: IVcache's behavior when the victim's access hits S state and E state.

One natural question is: *Is this enough for mitigating coherence attacks?* Cache coherence states include not only M, S, E, and I, but also transient states such as "waiting for data from memory" and "waiting for ACK signal". It is possible for the victim's access to hit a transient state. **We found that "waiting for data from memory" is a typical transient state that can leak secrets.** When an access misses in LLC, as shown in Figure 5(Case 1), a memory access will be generated, and one entry in MSHR records this access's address and the requesting core. Once the memory controller receives the data from memory, the data will be filled into LLC (in E state) and the requesting private cache. However, if LLC receives another access to the same line from a different core during the "waiting for data from memory" state, as shown in Figure 5(Case 2), the new core will be recorded in the same MSHR entry. When the data arrives, it is filled to LLC (in S state), and both of the requesting private caches.

The attack then works as follows. First, the attacker issues an access, and change the state to "waiting for data from memory". If the victim accesses the same line during this period, the attacker and victim will both receive the data, and the LLC state will be S. If the victim does not access the line, when the line is filled into LLC, it will be in E state. Then the attacker can distinguish the states by triggering an access to this line from another core as shown in Section 2. Attackers can utilize this leakage to defeat our previous defense.

To further defend transient state-based attacks, we block the victim's access if the coherence state is transient. If the victim's access hits a line in a transient state, IVcache will not dequeue it from the request buffer and refuse to serve it until the state of this line is changed to a non-transient state. Then, we handle the access as we did before.

3.4 Keeping Read-only Copies in Private Caches

While our self-eviction and bypass mechanisms effectively hide the victim's cache state changes from the attacker, 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 processes, lowering his residency percentage. Consequently, once the victim is completely evicted from an LLC set, the bypass mechanism is triggered, both LLC and the private cache are never filled again, and the victim enters permanent starvation. Due to cache inclusion, starvation in LLC causes the starvation in the private cache. As a result, the victim's private cache hit rate sharply decreases, which significantly degrades performance. To confirm this, we run SPEC2017 benchmarks in a 2-core system in the gem5 simulator; our configuration is shown in Table 2. For each experiment, we run 2 benchmarks, setting one to be the victim. As shown in Figure 6, in the baseline inclusive cache, the private cache hit rate is generally above 90%. IVcache, in contrast, degrades the private cache hit rate of the victim to at best, 80%, and at worst, 30%.

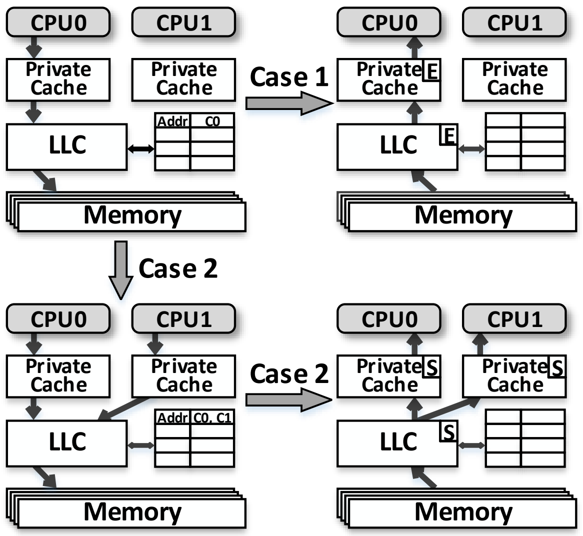

Figure 5: Coherence state transitions in different scenarios.

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 enables us to 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 then creates coherence problems. Thus, our design entails coherence policy modification to maintain correctness. To simplify the modification, we choose to fill the private cache with a read-only copy when LLC is bypassed. In this way, our modification only requires some small additional logic, as well as several fixed-size *record buffers* (one per victim) next to LLC. Each entry of the record buffer records an address of the cache line that is in the private cache but not LLC and the corresponding private cache id. When a victim's access bypasses LLC, the victim's record buffer is checked. If there is an available slot, the request address will be recorded, and the private cache is filled with a read-only copy. If the record buffer is full, this access will bypass filling the private cache, the data is sent directly to the CPU. In the full case, since the data resides neither in LLC nor the private cache, there's no coherence issue; issues only occur when there's a copy in the private cache but not LLC.

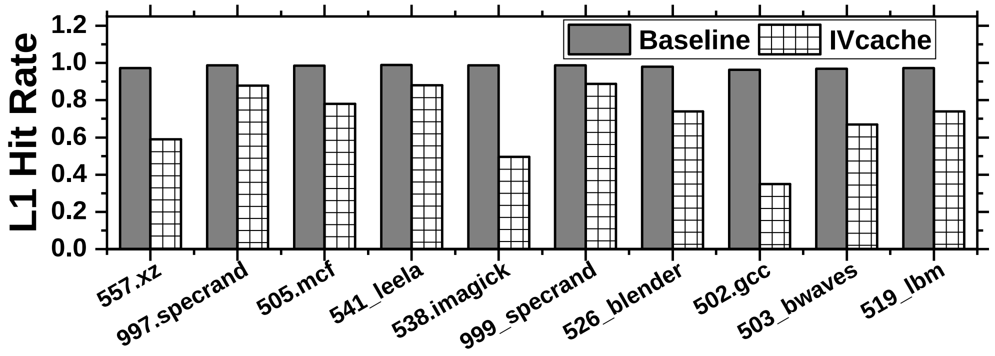

Figure 6: IVcache's effect on private cache hit rate for different victim benchmarks.

If *line l* (a private cache line) is recorded in the buffer[2], the victim's read access to this line can hit the read-only copy in the private cache. However, if the victim writes *line l*, our updated policy takes the following actions: (1) invalidates the copy in the private cache; (2) sends message to LLC to reset the corresponding entry in the record buffer; (3) forwards the write to LLC and LLC will handle the write securely as discussed before. Further, if another process (attacker or otherwise) misses in LLC, a memory access will be generated. Besides, LLC will check the record buffers for the requesting address. If found, LLC will signal the corresponding private cache to invalidate the read-only copy. This buffer check and signaling happen while LLC is waiting for the data from memory. Hence, it does not cause extra communication latency.

[2]We mark all the recorded private cache lines. This will be discussed in Section 3.6.3

3.5 LLC Active Invalidation

The record buffer only benefits the victim's read accesses, by allowing them to hit the read-only copies in the private cache. However, those read-only copies cannot serve writes. This is a severe problem: we observed memory traces and found that for most benchmarks, consecutive accesses to the same cache line in a short window occur more often for writes than reads (we conjecture this is due to compiler optimization of data structures, as well array writes).

Modern caches use the *write allocate* mechanism which benefits consecutive writes to the same line. When a write misses in cache, the data will be fetched from memory (taking potentially hundreds of cycles), and filled into the new allocated space in cache. Then, the following writes to this line will likely hit in the private cache, taking only a few cycles. Unfortunately, IVcache requires all the writes to go directly to memory (when the corresponding LLC set is bypassed), which can significantly hurt performance.

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 the line. Thus, for IVcache, once the victim does not own any lines in an LLC set (reaches bypass state), it's rare for the victim to recover entries (i.e. due to other processes' entries being flushed). To solve this problem, IVcache **actively invalidates a line from the inactive group in an LLC set**, writing it back to memory if necessary. In this way, IVcache actively gives the victim a chance to recover some LLC occupancy, allowing the victim to leave the bypass state and recover performance.

We classify all cache sets in LLC into *bypass* sets and *non-bypass* sets. Whenever a write from any victim bypasses LLC, the set in LLC will be marked as a bypass set; non-bypass sets are simply those not yet marked. For bypass sets, once per c_1 cycles, IVcache randomly chooses one set among all bypass sets, invalidate a line from the inactive group, and remove the set from bypass set list (setting c_1 is discussed in section 6). By doing this, bypass sets will have a chance to serve the victim again, recovering the victim from high write latency.

Non-bypass sets are not as critical to the victim's performance, but we still wish to use the invalidation mechanism to prevent them from becoming bypass sets. However, we want to be fair to non-victims since frequent invalidation may hurt performance (e.g. due to memory bus contention). So instead we dynamically invalidate to eliminate unnecessary invalidations. For each non-bypass set, we record the number of hits and misses from non-victims. By comparing the difference, we can determine if this is a good set to invalidate. When the hit rate of the set is high, replacement is occurring less frequently; non-victims are hitting in the set, not missing. Thus, after invalidating a line, it's less likely that non-victims will fill this space with a new line, instead the victim has a high chance to utilize this space. Therefore, in this case we make the set more likely to be invalidated. If the miss rate of the set is high, the non-victims are missing. After invalidation, non-victims will probably fill the space with a new line and the invalidation is less likely benefiting the victim so we reduce the probability for the set to be invalidated. We now exactly describe our parameters and how they update.

Algorithm 2 p Generation Algorithm

```
1: **Input:** $d_{base}$, $p_{base}$, $N_{miss}$, $N_{hit}$
2: **Output:** Invalidation probability,$p$
3: **Step 1:**
4:     $d = (N_{hit}$ - $N_{miss})$ - $d_{base}$
5: **Step 2:**
6:     $p' = d \, / \, \delta + p_{base}$
7:     **if** $p' > 1$ **then** $p = 1$
8:     **else if** $p' < 0$ **then** $p = 0$
9:     **else** $p = p'$
10: *End*
```

Once per c_2 cycles, IVcache randomly chooses one set among the non-bypass sets. However, after a set is chosen, the oldest line in the set will be invalidated only with a dynamic probability p. Algorithm 2 shows how p is calculated. After p is calculated, a random r between 0 and 1 is generated. If $r \leq p$, invalidate; else, don't. We first set a base probability p_{base} between 0 and 1, which is initially the same for each set, i.e. $p = p_{base}$ for all sets.

As mentioned, we update p dynamically, utilizing the history of misses and hits to this set to tune p_{base}. This history is cleaned (set $N_{miss} = N_{hit} = 0$) if this set is chosen to be invalidated. We do not record any victim's accesses in this history due to security issue: if the victim's accesses are counted, the attacker may get information about the victim's access pattern by monitoring the changes of the invalidation probability p.

As the formula indicates, if the difference between N_{hit} and N_{miss} is more than d_{base}, d is positive, and p increases in step 2; for the opposite case, p decreases. We tune d_{base} depending on how often we want to increase p. We choose δ so that $0 \leq p' \leq 1$ usually is true. In our experiments, this inequality is usually true; lines $10 - 12$ are for the rare false case.

3.6 Other IVcache Design Details

3.6.1 Effect of Context Switch

Context switch introduces security problems in some previous works, which use the core id for the partition boundary [49, 46]. Once the attacker is switched to the core that the victim was running on, the partition mechanism becomes invalid.

IVcache prevents this leakage by partitioning based on PCID instead of core id. Since PCID is constant during the execution, IVcache does not need to trust the OS to ensure the attacker will never use the core that the victim was using.

3.6.2 Effect of Hardware Prefetcher

Hardware prefetchers (e.g. stride prefetcher) have significant impact on the user's performance. However, prior research [42] has found that hardware prefetchers have powerful side channels (e.g. enabling the break of constant time cryptographic libraries). It's possible for attackers to manipulate the prefetcher to defeat IVcache's security guarantee. To prevent this, we modify the prefetcher to also follow IVcache's prior described security policies, i.e. prefetchers change cache states in the same way as demand accesses.

3.6.3 Hardware Extension

For any secure cache design, it is necessary to keep the simplicity of cache architecture. Here we show that IVcache only introduces negligible hardware extension. Firstly, for each cache line in the private cache, 1 bit is needed to track whether this line is a read-only copy or not. Secondly, for LLC, 1 bit is needed for each set to mark that this set is a bypass set or non-bypass set. 5 bits are needed to record $N_{hit} - N_{miss}$. 5 bits are used because during the experiment we found that over 99% of this value fits in the range of 0 - 20. When this value is going over 20 or less than 0, we could just stop updating the number stored in the set. This won't affect *active invalidation* since p' is already used to drop the effect of $N_{hit} - N_{miss}$ when the absolute value is large. We also track the owner process of each line in LLC to know if it belongs to a victim. However, there can be many processes running at the same time, causing the PCID space to be relatively large. This can make it very expensive to record the owner's PCID 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 PCID 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 the owner's PCID of each line. We leave the quantitative limitation of victims configurable to the processor designer.

Table 1: Storage Overhead of Record Buffer

Core Number	1	2	4	8
Overhead	0.1%	0.1%	0.1%	0.09%

As mentioned before, IVcache also requires a record buffer for each victim. The storage overhead of the record buffer is shown in Table 1, when the quantitative limitation of victims is set to be same with the number of physical cores.

4.0 Security Evaluation

In this section we evaluate the security guarantee of IVcache by both theoretical analysis and running real-world attacks on IVcache. The result shows that IVcache provides strict security guarantee.

4.1 Theoretical Analysis

The basic IVcache design effectively hides protected accesses in LLC by avoiding all visible state changes. Here we discuss the security effect of the optimization mechanisms: keeping read-only copies(Section 3.4) and active invalidation(Section 3.5).

Algorithm 3 Square-and-Multiply exponentiation

```
Input: base $b$, modulo $m$, exponent $e = (e_{n1} ...e_0)_2$
Output: $b^e$ mod $m$
1: r ← 1
2: for $i$ from $n-1$ down to 0 do
3:     $r \leftarrow r^2 \bmod n$
4:     if $e_i = 1$ then
5:         $r \leftarrow r * b \bmod n$
6: return $r$
```

If an attacker's access misses in LLC, but the line is in a record buffer entry (i.e. a victim has a read-only copy in the private cache), IVcache triggers a memory access instead of forwarding the data from the victim's private cache. Therefore, the attacker cannot determine if the victim loaded this line. If the attacker claims to be a "victim", he is allocated a new record buffer; buffers are not shared. Thus, the attacker gains no information about the victim's record buffer utilization.

For the active invalidation mechanism, the attacker may try to leverage the invalidation frequency to leak secrets. However, from the invalidation frequency, the attacker can only learn ① bypass sets and non-bypass sets, and ② determine the invalidation probability p of each non-bypass set.

These do not leak any private information from the victim. For ①, this only tells the attacker that the victim lost set residency, which is probably because the attacker is building conflicts. Thus this information is already known by the attacker. For ②, the invalidation probability is only determined by hits and misses of non-victims; it is decoupled from the victim. Thus, the attacker cannot monitor the victim's behavior from these two points.

sectionCase study on defending real attacks

We test IVcache against known real-world attacks in the gem5 simulator. Our system configuration is shown in Table 2. We use GnuPG [2] as our victim in each attack.

4.1.1 Prime+Probe Attack

The square-and-multiply algorithm [13] is found in GnuPG version 1.4.13 for both RSA [38] and ElGamal [12] ciphers; leaking the exponent e of this algorithm leaks the decryption key. As shown in Algorithm 3, in each iteration of the *for* loop, the executed instruction pattern is related to a 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 set that the square instruction is mapped to, the attacker can learn each bit of the exponent.

In Prime+Probe, the attacker doesn't know the physical address of the square instruction; he needs to profile LLC to learn each set's access pattern, and then monitor the set whose pattern is closest to the square instruction's pattern. To profile LLC, we probe each set every 5000 cycles. The top of Figure 7 shows the access pattern in 50 time slots of an LLC set (set_s) corresponding to the square instruction. Each dot represents an access to the square instruction, detected by the attacker.

Most distances between two consecutive dots are either one of two distinct distances, except two dots. The square 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 baseline shows the key pattern: 110001(10/01)01, where the parentheses mean uncertainty due to the noise.

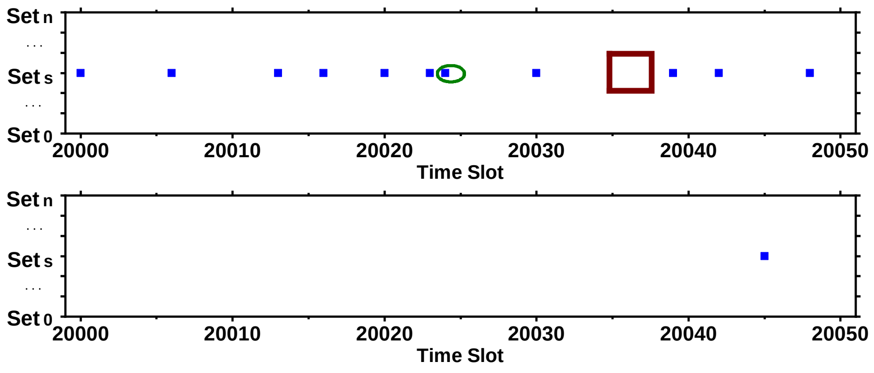

Figure 7: Prime+Probe attack results.

The bottom of Figure 7 shows the result of IVcache. Once the attacker primes the cache set (fully occupies it), the victim will bypass this LLC set. Thus, the attacker cannot cause conflicts with the victim. The only spot the attacker caught in the trace is from the attack noise, as explained earlier.

4.1.2 Coherence-based Attack

In this experiment, we force the sharing between the attacker and GnuPG by using `mmap()` to map the page that consists of the square instruction as in line 3 of Algorithm 3 into the attacker's address space. As the prerequisite of the experiment, we tested the LLC hit latency in the gem5 simulator, which is about 105 CPU cycles for S state, and 122 CPU cycles for E state.

As mentioned in Section 2, the attacker learns the victim's access patterns by determining the coherence state of the shared line, based on the access latency in Step 3. Figure 8 shows the probability distributions of the access latency of the square instruction line recorded by the attacker. As shown, for the attack on the baseline cache, the attacker's LLC hits consist of both S state hits and E state hits. However, for the attack on IVcache, the attacker's LLC hits to the square instruction line only have E state hits, because IVcache does not change the coherence state of a shared line for victim's accesses. Hence, the attacker is unable to learn the victim's access pattern by monitoring the coherence changes.

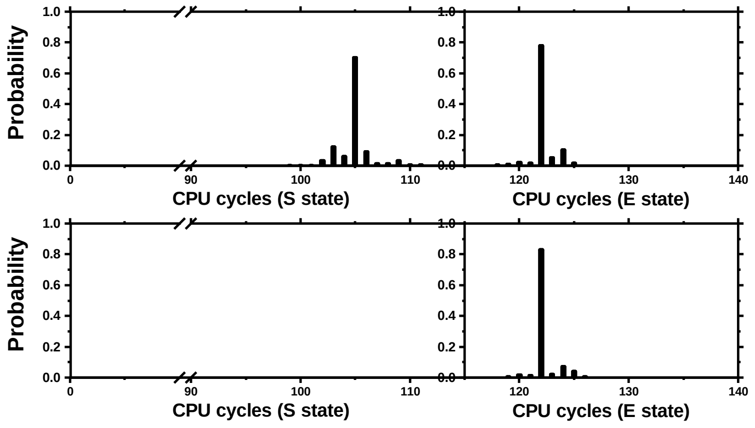

Figure 8: Probability distributions of attacker's access latency.

5.0 Experiment

We implement IVcache on a cycle-accurate simulator, gem5 [6]. To evaluate IVcache, we run SPEC2017 benchmarks for 1 billion instructions in each experiment. Our baseline cache parameters are shown in Table 2.

5.1 Performance Evaluation

As mentioned in Section 3, there are several parameters we need to set for IVcache. Table 3 gives an example setting of IVcache. In this section, we only show the experiment results under this fixed setting. The configuration of the parameters will be discussed in next section.

Table 2: Experiment Configuration

Parameter	**Description**
ISA	X86
processor type	8-way out-of-order, no SMT
L1-Icache, private	32kB, 8 way associative, 1 cycle latency
LI-Dcache, private	32kB, 8 way associative, 1 cycle latency
L2 cache, shared	2MB bank per core, 16 way associative
cache polices	LRU, Directory-based MESI Coherency

We test the performance of IVcache on 2-core, 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 or 2 benchmarks as victims. 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 blender, we run 10 tests where the other 6 benchmarks are randomly selected. Thus, in Figure 9 and Figure 10, the bars are averages among the 10 tests; each bar refers to fixed victims. These figures show the IPC and private cache hit rate, normalized to the baseline insecure cache.

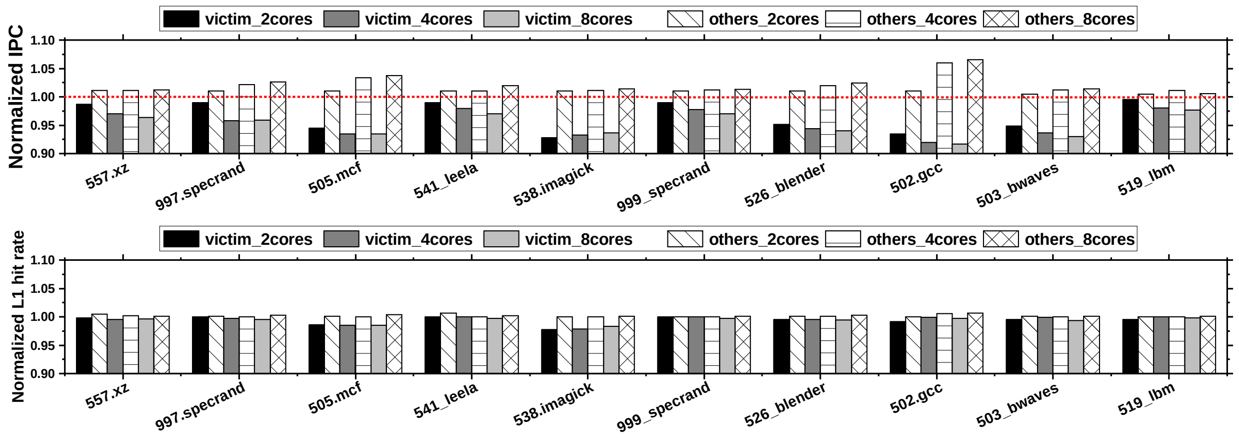

Figure 9: Normalized Performance overhead of IVcache when there's one victim in the system.

As shown in Figure 9 and Figure 10, 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 8.4% for 1-victim systems, and 7.3% for 2-victim systems, and the average is only 4.7% for 1-victim systems, and 5.5% for 2-victim systems, which are negligible. There is always a performance gap between victims and other processes. When the system has more CPU cores, the gap increases. This increase of performance gap is because the invalidation rate is the same for all experiments, regardless of the number of cores. However, in our LLC, the number of banks is same with the number of CPU cores, which is consistent with modern processors. Thus, with more cores, the number of LLC sets increases; if we keep invalidating 1 set per 100 cycles, the time gap between two consecutive invalidations to the same set becomes longer, hurting the victim's performance.

Comparing Figure 9 and Figure 10 with Figure 6, we see that our performance optimizations, keeping read-only copy and active invalidation, almost recover victims' private cache hit rate. This is very important for our design: in a normal inclusive cache, after missing in the private cache, the victim's access still has a high probability to hit in LLC. However, in IVcache, if the LLC is bypassed, the access goes directly to memory, which results in over 100 times higher latency, compared to hitting in the private cache. Additionally, IVcache barely affects the private cache hit rate of unprotected processes.

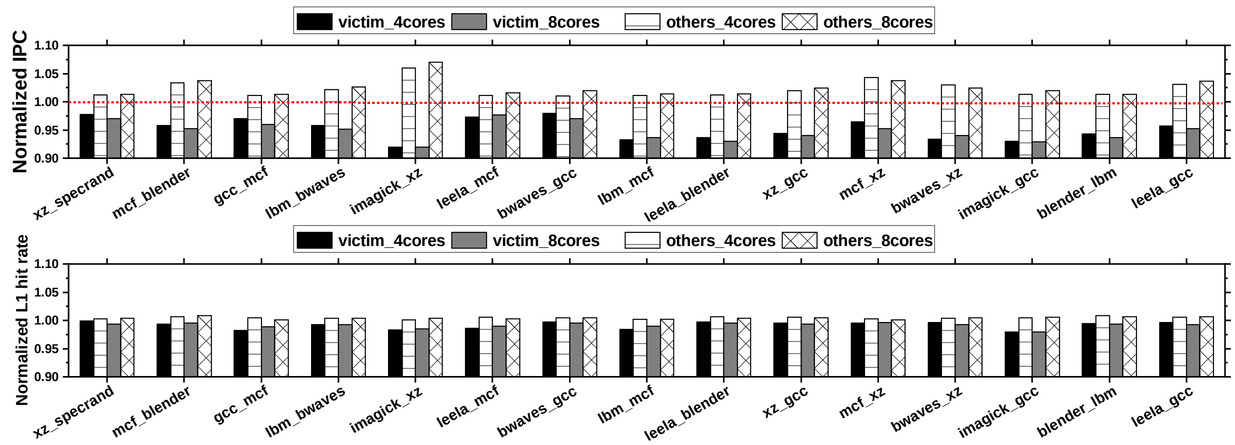

Figure 10: Normalized Performance overhead of IVcache when there are two victims in the system.

5.2 Parameter Configuration

5.2.1 Record Buffer Size Sensitivity Study

The record buffer is used to recover the victim's private cache hit rate (and thus performance). It is necessary to find a balance between the storage overhead of the record buffer and the performance benefit the buffer offers to the victim. Here we show the average IPC change of 10 victim benchmarks derived from SPEC2017. For each victim benchmark, we run it on a 4-core system with 3 other non-victim benchmarks. From Figure 11, we see that when the buffer has lower than 30 entries, increasing buffer size can clearly benefit the victim's IPC. However, when the buffer has near 40 entries, the performance result becomes stable. Thus, we give each buffer 40 entries, which only introduces 1% storage overhead to LLC.

Table 3: IVcache Parameters

Record Buffer Entry	40
Invalidation Rate c_1, c_2	100 cycles
p_{base}	0.5
d_{base}	50
Size of Active Group	9

5.2.2 Other Parameters

While the active invalidation mechanism benefits victims by preventing starvation, a high invalidation rate can also hurt non-victims' performance, since a high invalidation rate will cause contention on the memory bus and also decrease the LLC hit rate. We test the performance effect of different invalidation rates on the same system as the record buffer experiment, and we found that, on average, when the invalidation rate is higher than once per 110 cycles, our design starts to reduce non-victim processes' performance. Thus, in Section 5.1, we set both $c1$ and $c2$ to once per 100 cycles. However, as we mentioned before, when there's more physical cores in the system, $c1$ and $c2$ need to be changed correspondingly. We leave this configurable for processor designers.

In the active invalidation mechanism, the difference between the number of recorded cache hits and misses is checked when a non-bypass set is chosen to be invalidated. We print this record and find that for 16-way associative LLC, about 50% recorded number is over 10, and most less than 20. Thus, we simply set d_{base} to 10 and set p_{base} to 0.5 to keep a balance for tuning the probability up and down.

The size of the active group in our secure replacement design can affect performance, depending on the application. We test the IPC of different victim benchmarks with the size of the active group reducing from 16 (associativity) to 0. We found that when the size is larger than 9, IVcache gives less than 10% IPC degradation to the victim. Thus, we set this value to be 9 to keep generally acceptable performance. Finding better parameters is left as future work even though the experiment result shows that our configuration is very reasonable.

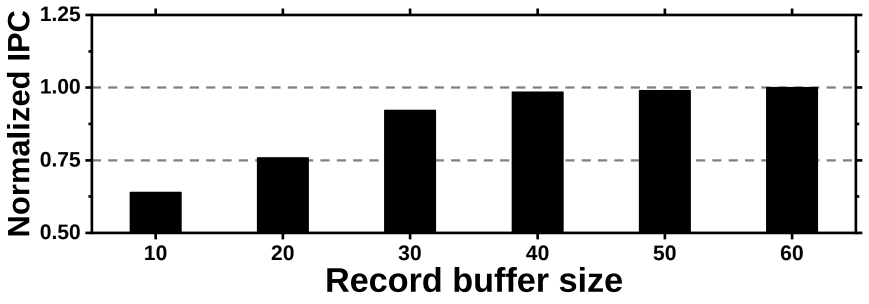

Figure 11: The IPC effect under different record buffer sizes.

6.0 Conclusions

This paper gave an important insight that not only the existence state of cache lines can be utilized to leak information, but also the replacement state and coherence state. Based on this insight, we proposed IVcache, which is a defense mechanism for both traditional LLC attacks and also the new variants. IVcache makes the victim's behavior absolutely invisible to attackers by modifying the way that state changes are handled. Besides, we provided two optimization mechanisms for IVcache to provide good performance to users.

We implemented IVcache in the gem5 simulator. We tested IVcache against real-world attacks, which showed that IVcache could effectively defend these attacks. Besides, we used benchmarks from SPEC2017 to evaluate IVcache's performance degradation. The result showed that IVcache provides negligible performance effect to all workloads.

Bibliography

[1] Intel 64 and ia-32 architectures software developer's manual. Intel Corporation, 2013.

[2] Using the gnu privacy guard. 2019.

[3] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *RAID*, 2014.

[4] Daniel J. Bernstein. Cache-timing attacks on aes. 2005.

[5] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. *CoRR*, abs/1903.01843, 2019.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 667–684, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 667–684, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4):35:1–35:21, January 2012.

[11] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[12] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theor.*, 31(4):469–472, September 2006.

[13] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, April 1998.

[14] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, 2017.

[15] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[16] 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, Berlin, Heidelberg, 2016. Springer-Verlag.

[17] 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, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag.

[18] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 897–912, Berkeley, CA, USA, 2015. USENIX Association.

[19] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium*

on Security and Privacy, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+reload attack on aes. In *COSADE*, 2015.

[21] Taylor Hornby. Side-channel attacks on everyday applications: distinguishing inputs with flush+ reload, 2017.

[22] Hailiang Huang, Chenggang Yan, Bingtao Liu, and Licheng Chen. A survey of memory deduplication approaches for intelligent urban computing. *Mach. Vision Appl.*, 28(7):705–714, October 2017.

[23] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 591–604, Washington, DC, USA, 2015. IEEE Computer Society.

[24] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 353–364, New York, NY, USA, 2016. ACM.

[25] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[26] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 7:1–7:6, New York, NY, USA, 2017. ACM.

[27] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 974–987, Piscataway, NJ, USA, 2018. IEEE Press.

[28] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *IEEE Transactions on Computers*, 62(7):1276–1288, 2012.

[29] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[30] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, September 2016.

[31] 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.

[32] 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.

[33] Rodney Owens and Weichao Wang. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, PCCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[34] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005. page@cs.bris.ac.uk 13017 received 22 Aug 2005.

[35] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+ probe attacks and covert channels in scattercache. *arXiv preprint arXiv:1908.03383*, 2019.

[36] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[37] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 360–371, New York, NY, USA, 2019. Association for Computing Machinery.

[38] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[39] Mark S. Papamarcos and Janak Patel. A low-overhead coherence solution for multiprocessors with private cache memories. volume 12, pages 284–290, 01 1998.

[40] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv:1905.05726*, 2019.

[42] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 131–145, New York, NY, USA, 2018. Association for Computing Machinery.

[43] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 98:1–98:6, New York, NY, USA, 2018. ACM.

[44] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[45] 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.

[46] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[47] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, August 2019. USENIX Association.

[48] Wenjie Xiong and Jakub Szefer. Leaking information through cache LRU states. *CoRR*, abs/1905.08348, 2019.

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

[50] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE S&P 2019*, 2019.

[51] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[52] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[53] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7:99–112, 2016.