Received September 11, 2020, accepted September 25, 2020, date of publication September 30, 2020,
date of current version October 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3027946

Defending Against Flush+Reload Attack With DRAM Cache by Bypassing Shared SRAM Cache

MINWOO JANG, SEUNGKYU LEE, JAEHA KUNG, (Member, IEEE), AND DAEHOON KIM, (Member, IEEE)

Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu 42988, South Korea

Corresponding author: Daehoon Kim (dkim@dgist.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Korea Government Ministry of Science and ICT (MSIP) under Grant NRF-2020R1C1C1013315 and Grant NRF-2018R1A5A1060031, and in part by the Institute for Information and Communications Technology Promotion (IITP) funded by the Korea Government (MSIP), through Resilient Cyber-Physical Systems Research, under Grant 2014-0-00065.

ABSTRACT Cache side-channel attack is one of the critical security threats to modern computing systems. As a representative cache side-channel attack, Flush+Reload attack allows an attacker to steal confidential information (e.g., private encryption key) by monitoring a victim's cache access patterns while generating the confidential values. Meanwhile, for providing high performance with memory-intensive applications that do not fit in the on-chip SRAM-based last-level cache (e.g., L3 cache), modern computing systems start to deploy DRAM cache between the SRAM-based last-level cache and the main memory DRAM, which can provide low latency and/or high bandwidth. However, in this work, we propose an approach that exploits the DRAM cache for security rather than performance, called `ByCA`. `ByCA` bypasses the L3 shared cache when accessing cache blocks suspected as target blocks of an attacker. Consequently, `ByCA` eliminates the timing difference when the attacker accesses the target cache blocks, nullifying the Flush+Reload attacks. To this end, `ByCA` keeps cache blocks suspected as target blocks of the attacker and stores their states (i.e., flushed by `clflush` or not) in the L4 DRAM cache even with `clflush` instruction; `ByCA` re-defines and re-implements `clflush` instruction not to flush cache blocks from the L4 DRAM cache while flushing the blocks from other level caches (i.e., L1, L2, and L3 caches). In addition, `ByCA` bypasses L3 cache when the attacker or the victim accesses the target blocks flushed by `clflush`, making the attacker always obtain the blocks from L4 DRAM cache regardless of the victim's access patterns. Consequently, `ByCA` eliminates the timing difference, thus the attacker cannot monitor the victim's cache access patterns. For L4 DRAM cache, we implement Alloy Cache design and use an unused bit in a tag entry for each block to store its state. `ByCA` only requires a single bit extension to cache blocks in L1 and L2 private caches, and a tag entry for each block in the L4 DRAM cache. Our experimental results show that `ByCA` completely eliminates the timing differences when the attacker reloads the target blocks. Furthermore, `ByCA` does not show the performance degradation for the victim while co-running with the attacker that flushes and reloads target blocks temporally and repetitively.

INDEX TERMS Flush+Reload attack, DRAM cache architecture, cache side channel attack, cache architecture.

I. INTRODUCTION

Cache side-channel attack that can steal confidential information (e.g., private encryption key) becomes one of the critical threats to modern computing systems. As a representative cache side-channel attack, Flush+Reload attack [1] enables an attacker to steal the private encryption key of a victim by monitoring an execution flow of the victim represented by access patterns to shared cache (e.g., L3 cache). Furthermore, speculative attacks that attempt to figure out confidential data where the attacker does not have access permission (e.g., Meltdown [2], Spectre [3]) exploit the Flush+Reload attack. To figure out victim's access pattern to the shared cache, a Flush+Reload attacker first flushes target blocks, and waits

The associate editor coordinating the review of this manuscript and approving it for publication was Cong Pu.

for victim's accesses, and reloads the target cache blocks. When reloading the target blocks, the attacker can distinguish whether target blocks are accessed by the victim or not by measuring access latency to the target blocks. The attacker obtains the target block with shorter latency with a cache hit if the victim accesses the target block while obtaining the block, or with longer latency with a cache miss if the victim does not access the block. Consequently, exploiting the timing differences, Flush+Reload attacks can steal the confidential information that can be inferred by monitoring access pattern to the cache memory. Meanwhile, for high performance with memory-intensive applications that do not fit in the SRAM-based Last-level Cache (LLC), which is typically L3 cache and shared by all cores, modern computer systems, such as Intel's Knights' Landing [4], start to deploy an L4 DRAM cache based on emerging 3D stacked memory technology (e.g., HBM [5], HMC [6]) between the SRAM LLC and the main memory DRAM. Such DRAM cache can provide low latency and/or high bandwidth compared with the main memory, and much larger capacity than the SRAM cache according to the type of DRAM and its implementation [7]. To maximize its benefits regarding performance, most of prior work propose various techniques for the DRAM cache [8]–[12].

In this work, we propose `ByCA` that exploits the L4 DRAM cache for security rather than for performance, assuming SRAM and shared LLC is at L3 as most of modern processor architecture; note that `ByCA` also can exploit the DRAM cache at any level, but right below shared SRAM LLC targeted by attacker. In particular, by keeping cache blocks suspected as attacker's target blocks and storing their state in L4 DRAM cache while not loading them to shared L3 cache, `ByCA` eliminates the timing differences seen by the attacker, when the attacker accesses the target cache blocks, nullifying Flush+Reload attack. To this end, we implement the L4 DRAM cache based on Alloy Cache [10], and implement `bypass bit` using a single bit of a tag entry for each block to record whether the block was flushed or not. By bypassing the shared LLC when the attacker/victim accesses target blocks with `bypass bit` set, the attacker always obtains the blocks from the L4 DRAM cache when private cache misses occur regardless of victim's accesses to the blocks.

For leaving the flushed blocks in the L4 DRAM cache, we re-define and re-implement `clflush` instruction, provided by x86 ISA, which flushes cache blocks from all cache memories in the entire hierarchy. The re-defined `clflush` does not flush the target blocks in L4 DRAM cache while flushing the blocks loaded in other level caches (e.g., L1, L2, and L3 caches), which makes the flushed blocks be written back to the L4 DRAM cache rather than main memory. Leaving the flushed blocks in the L4 DRAM cache while flushing the blocks in other level caches also can defend the system against Flush+Reload attacks targeting the L4 cache by negating the effect of `clflush` in the L4 DRAM cache.

To the best of our knowledge, this is the first work that exploits L4 DRAM cache for defending against cache side-channel attacks rather than for improving performance. With very simple hardware modification, `ByCA` fundamentally eliminates both L3 shared SRAM cache and L4 DRAM cache from the attack surface of Flush+Reload. Our experimental results show that `ByCA` can eliminate timing differences, completely nullifying Flush+Reload attacks with simple hardware modifications without performance degradation. The major contributions of our work are as follows:

1) We propose a simple L3 bypass architecture and modified `clflush` instruction, which nullifies Flush+Reload attack by eliminating the timing differences.
2) `ByCA` fundamentally eliminates both L3 and L4 DRAM cache from the attack surface.
3) `ByCA` does not require complex hardware modification while just requiring single bit extension to cache blocks in L1 and L2 caches and tag entries in L4 DRAM cache.
4) We demonstrate that our approach eliminates the timing differences when accessing flushed cache blocks, nullifying Flush+Reload attack using trace-driven simulator.

The rest of this paper is organized as follows. Section II explains background and related work. After describing threat model of this paper in Section III, we introduce `ByCA` to nullify Flush+Reload attacks exploiting L4 DRAM cache in Section IV. Section VI discusses issues of `ByCA` and potential extensions of `ByCA`. Section V describes our experimental methodology and results. Section VII concludes this paper.

II. BACKGROUND AND RELATED WORK

A. CACHE SIDE-CHANNEL ATTACK AND FLUSH+RELOAD ATTACK

The cache side-channel attacks (e.g., Flush+Reload [1], Prime+Probe [13], Evict+Reload [14] or other variants [15], [16]) can steal confidential information by inferring an execution flow represented by cache access patterns of a victim process; those attacks can also directly infer confidential data (e.g., encryption key) by accessing data where the attacker does not have access permission (e.g., Meltdown [2], Spectre [3]). To infer the execution flow, the attacker keeps monitoring the access patterns by exploiting timing difference when accessing a particular cache block in the cache memory. Such cache side-channel attacks repeat following steps. First, the attacker initializes the target cache memory to its desired state by flushing or filling cache blocks, and then waits for a while to let the victim access the cache memory. Next, the attacker reloads or refills cache blocks, and figures out which cache blocks are accessed by the victim process by exploiting the timing difference when reloading or refilling blocks. For example, the attacker experiences much shorter latency with a cache hit when reloading the flushed block if the victim accesses and loads the block to the cache memory

from the main memory while the attacker waits. By repeating the initialization, wait, and reload, the attacker can monitor the access patterns of the victim at the cache memory.

Algorithm 1 Square-and-Multiply Algorithm of GnuPG

```
1:  INPUT: base b, exponent[] e, modulo m
2:  x ← 1
3:  for i ← |e| − 1 to 0 do
4:      x ← x ∗ x  (Square)
5:      x ← x mod m  (Reduce)
6:      if e[i] = 1 then
7:          x ← x ∗ b  (Multiply)
8:          x ← x mod m  (Reduce)
9:      end if
10: end for
11: return x
```

Such cache side-channel attacks mainly target processes generating private encryption keys using open source libraries (e.g., GnuPG [17], OpenSSL [18]) while showing the different cache access patterns depending on the generated key values. For example, when the victim process generates an encryption key bit by bit, it shows different execution flows depending on the generated bit while accessing different cache blocks in the shared cache. Algorithm 1 shows the pseudo-code of Square-and-Multiply exponentiation used by GnuPG version 1.4.13. At every iteration the algorithm refers the exponent bits, performing square and reduce operation. If the exponent bit is one, the algorithm calls multiply and reduce functions additionally while not calling those functions when the bit is zero. Consequently, since the exponent bit determines the control flow of GnuPG, the attacker can infer the value of the exponent bit of each iteration by monitoring the control flow of the victim process. Suppose the attacker observes that the victim calls square-reduce-multiply-reduce functions in order. In that case, they figure out that the exponent bit is one while figuring out the bit is zero if they observe that the victim calls square-reduce functions. Thus, monitoring the execution flow when the victim generates the private key allows the attacker to obtain the entire key value.

Flush+Reload attack [1] is a representative cache side-channel attack that infers the secret key by monitoring access patterns of the victim to shared cache. Meltdown [2] and Spectre [3] use such Flush+Reload attack as a part of their attacks inferring confidential information inside the cache memory stored by the speculative execution conducted by the attacker. The steps of Flush+Reload attack are as follows. To initialize the state of cache memory, the attacker flushes target cache blocks including function codes executed while generating the secret key in the shared library (e.g., GnuPG) using `clflush` instruction. Next, the attacker waits for a certain amount of time to let the victim access the flushed blocks while generating the key value. Lastly, the attacker reloads the target blocks, and measures the access latency to distinguish whether the victim accessed the target blocks or not. If the access latency is very short (i.e., cache hit latency) when reloading the blocks, it means that the victim has accessed and loaded the blocks to the shared cache before the attacker reloads the blocks. Otherwise, it means that the victim has not accessed the blocks, thus the attacker obtains blocks from the main memory with longer latency (i.e., main memory latency). Depending on a generated bit (i.e., 0 or 1) of the key, the attacker experiences cache hits at different cache blocks, which enables them to infer the value of the generated bits.

B. L4 DRAM CACHE ARCHITECTURE

Several studies propose a cache architecture based on emerging high-bandwidth DRAM (e.g., HBM [5], HMC [6]), called L4 DRAM cache, implemented between the SRAM-based LLC and main memory. Compared with the SRAM-based L3 cache, the L4 DRAM cache provides higher bandwidth and larger capacity at the cost of latency [7]. The DRAM cache based on the traditional cache architecture requires two separate accesses, for tag and data, which become the main sources of the access latency [19].

There have been several studies to improve the performance of the DRAM cache. Loh and Hill [12] propose the DRAM cache architecture that places tag and data in the same DRAM row, reducing the access latency by accessing both tag and data with a row buffer hit. They also provide MissMap that avoids extra DRAM cache accesses by identifying that the corresponding cache blocks do not exist in the DRAM cache. Huang and Nagarajan [11] use SRAM banks for storing tags, reducing tag lookup latency. The other prior works in [8], [9] propose an on-chip L4 DRAM cache based on eDRAM and 3D-stacked DRAM, respectively. By placing the L4 DRAM cache near processors or deploying private L4 cache for each core, we can benefit from the lower latency than the off-chip DRAM.

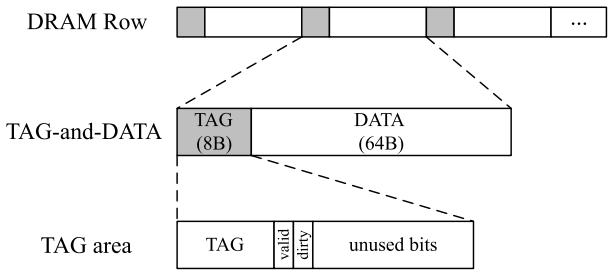


FIGURE 1. The architecture of alloy cache.

In this work, for the L4 DRAM cache that provides high bandwidth and large capacity at the cost of latency [7], we employ Alloy Cache [10] architecture. Figure 1 illustrates the architecture of Alloy cache, which places a tag next to data in the same DRAM row. Since the tag and data block are placed continuously in a DRAM row, Alloy Cache does not require an extra access to the DRAM cache for the tag lookup, and can even read the tag and data with a single burst of DRAM read operation. Alloy Cache has unused bits in each

tag entry, thus we can store additional information that can be used for further performance improvement. Young *et al.* [20] and Chou *et al.* [21] use the unused bits in the tag entries of Alloy Cache for supporting cache compression and cache coherence of multi-node systems. In this work, we also use the unused bits in the tag entries of Alloy Cache to defend against the Flush+Reload attack.

C. DEFENDING AGAINST FLUSH+RELOAD ATTACKS

A quite number of studies propose solutions to defend against cache-side channel attacks including Flush+Reload. Liu *et al.* [22], Wang *et al.* [23], Kiriansky *et al.* [24] propose software-based cache partitioning policies using Intel's Cache Allocation Technology (Intel CAT) to allocate separate cache partitions for attacker and victim. However, partitioning shared cache inherently degrades the overall cache utilization [25] while requiring a rich hardware implementation such as Intel CAT technology [26]. RIC [27] allows an exception for read-only blocks not to obey inclusion property in the inclusive cache since target blocks including program codes (instructions) are read-only. However, protecting read-only blocks is not enough for attacks that do not target such code blocks (e.g., Meltdown). SHARP [28] modifies `clflush` not to allow user space applications to flush non-writable cache blocks to disable Flush+Reload Attack. Therefore, SHARP can defend systems against Flush+Reload, but applications cannot use `clflush` instruction.

III. THREAT MODEL

We assume the victim and attacker co-run in a single physical machine while not sharing a physical core. We assume that a physical machine has a private L1 and L2 cache for each core, an L3 cache shared by all cores, and an L4 DRAM cache also shared by all cores, thus the attacker and victim can share the L3 and L4 cache. We do not consider other side-channel attacks such as Prime+Probe and Evict+Reload, or attacks exploiting speculative executions, (e.g., Meltdown, Spectre).

IV. ARCHITECTURE

In this section, we propose `ByCA`, **By**passing **C**ache **A**rchitecture, which defends against Flush+Reload attack by exploiting L4 DRAM cache. We first discuss how to bypass the shared L3 cache when loading cache blocks targeted by an attacker conducting Flush+Reload attack to eliminate the timing differences when accessing the target blocks in the conventional cache hierarchy without DRAM cache. Next, we discuss how we can exploit L4 DRAM cache to support for bypassing.

A. BYPASSING L3 CACHE IN CONVENTIONAL CACHE HIERARCHY

To prevent Flush+Reload attack inferring victim's confidential information by monitoring the shared L3 cache, we can simply bypass the L3 cache when loading cache blocks flushed by `clflush` instruction. With bypassing the L3, an attacker cannot exploit timing differences when reloading the flushed blocks since the L3 does not hold the blocks, and thus the attacker always obtain the blocks from the main memory. To support the bypassing, we can maintain an additional bit (i.e., `bypass bit`) in somewhere to mark whether the cache block was flushed by `clflush` instruction or not; when a block is flushed by `clflush` instruction, `bypass bit` is set. Assuming typical cache hierarchy with three level caches without L4 DRAM cache, the `bypass bit` can be implemented by extending data or hardware structure for metadata, such as page table, Translation Look-aside Buffer (TLB). However, an entry of TLB and page table has page information for a virtual page. Therefore, we must track and update all virtual pages mapped to a physical page including target cache blocks to maintain `bypass bit` correctly.

If the `bypass bit` is set when a process accesses a cache block, the cache controller loads the block just into private L1 and L2 caches, and bypasses the shared L3 cache. Thus, when an attacker running on another core reloads the flushed block with the `bypass bit` set, the block is obtained from the main memory along with a cache miss regardless of the behavior of a victim. Consequently, such bypassing approach nullifies the Flush+Reload attack by eliminating timing differences when the attacker accesses target blocks flushed by `clflush` instruction. To support bypassing L3 for cache blocks from L1 and L2 caches, the cache blocks in the caches also keep the `bypass bit`. For example, dirty blocks can be written back to L3 cache when they are evicted or invalidated at the L1 or L2 caches. If there is inclusion property between L1 and L2 caches, only the L2 cache needs to store `bypass bit` in the blocks. Otherwise, both L1 and L2 caches need to maintain the `bypass bit` in each block to bypass L3 when written back to the L3 cache.

B. BYPASSING L3 EXPLOITING L4 DRAM CACHE

Bypassing L3 in the conventional cache hierarchy with three levels faces an issue about metadata: where to store `bypass bit` when L1 or L2 misses occur. In this section, we propose `ByCA` that exploits the L4 DRAM cache supporting the bypassing L3 cache, nullifying Flush+Reload attacks. To this end, `ByCA` re-defines and re-implements `clflush` instruction not to flush cache blocks from the L4 cache while flushing the blocks from L1, L2, and L3 caches. `ByCA` stores the `bypass bit` in an unused bit of a tag entry for each block in L4 DRAM cache, and sets the bit when the block is flushed. When a victim/attacker accesses the flushed cache block, they obtain the block from the L4 cache. After the cache controller checks the `bypass bit`, if the bit is set, it loads the block to L1 and L2 cache while bypassing the L3 cache. Since the attacker always obtains flushed blocks from L4 DRAM cache, and the blocks are not loaded to L3 cache, `ByCA` completely nullifies Flush+Reload attack with a negligible space overhead (i.e., one bit per cache block in L1, L2, and DRAM cache).

Figure 2a, Figure 2b, and Figure 2c illustrate when an attacker flushes target blocks, waits for victim's access, and

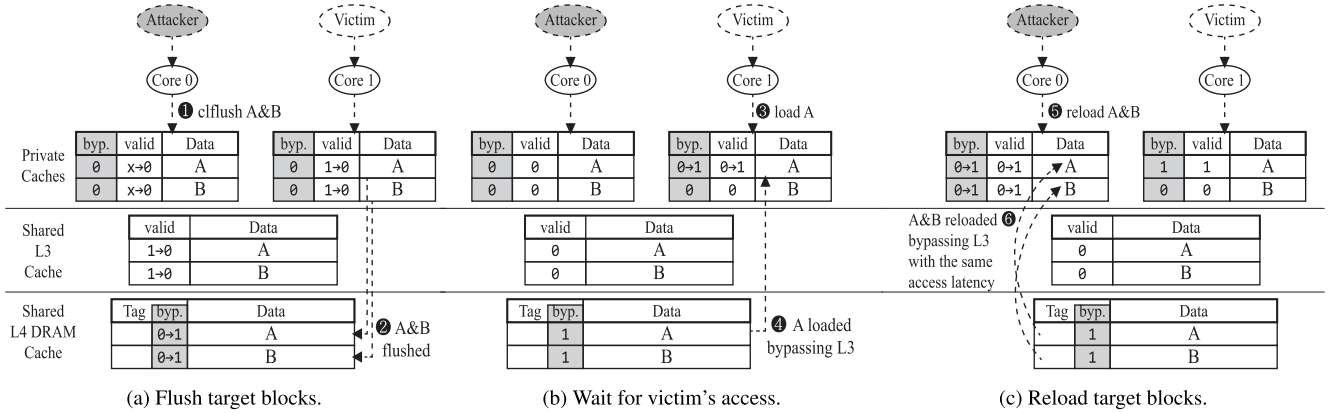


FIGURE 2. Nullifying Flush+Reload attack exploiting L4 DRAM cache.

reloads blocks, respectively, with ByCA. ❶ Attacker flushes target blocks, resulting in invalidation of block A and B in private caches, and shared L3 cache. Note that the bypass bit in shared L4 DRAM cache is set, which makes blocks bypass L3 cache on reloads. ❷ The block A and B in private caches of victim's core is also invalidated in accordance with the inclusive nature. ❸ The victim accesses block A, ❹ which is loaded from L4 cache bypassing L3 cache as the bypass bit in L4 cache is set. ❺ The attacker reloads both block A and B to distinguish the blocks accessed by the victim. ❻ Both blocks are loaded from L4 DRAM cache with the same access latency since the bypass bit is set for both blocks regardless of victim's accesses. Consequently, the attacker cannot exploit the timing differences, thus cannot infer confidential information by conducting Flush+Reload attack.

V. EVALUATION AND ANALYSIS

A. EXPERIMENTAL METHODOLOGY

We implement and evaluate ByCA with a trace-driven simulator, Sniper [29]. Table 1 shows hardware configurations that we modeled. We configure shorter latency for L4 DRAM cache than the main memory DRAM. As a base architecture of DRAM cache, we implement Alloy Cache and implement clflush instruction. We model a cache hierarchy with inclusion property while allowing the exception for flushed blocks by clflush instruction. We implement an attacker program that conducts Flush+Reload and use GnuPG 1.4.13 as a victim program while running them on separate cores. We evaluate with three scenarios, 1) baseline (not bypassing L3 + flushing blocks in L4), 2) bypassing L3 only (bypassing L3 + flushing blocks in L4), and 3) ByCA (bypassing L3, not flushing blocks in L4).

B. SECURITY EVALUATION

To demonstrate that ByCA nullifies Flush+Reload attack by eliminating the timing differences when accessing flushed blocks, we plot the access latency of every accesses to target blocks by the attacker in the middle of execution in Figure 3.

TABLE 1. Hardware configuration.

hardware	configuration
Processor	quad-core, 2.66GHz
L1-I/D	32KB, 4-way, 4 cycles, private
L2 cache	256KB, 8-way, 8 cycles, private
L3 cache	8MB, 16-way, 35 cycles, shared
L4 DRAM cache	1GB, direct-mapped, 70 cycles, shared
Main memory	infinite capacity, 110 cycles

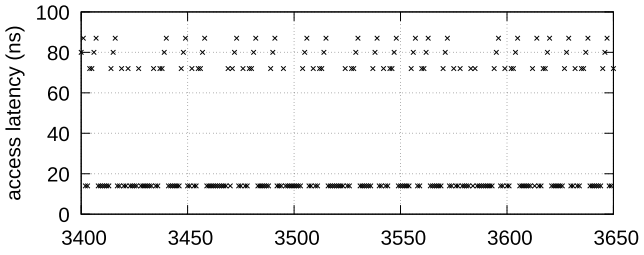
The x-axis shows the order of accesses to the target blocks while the y-axis shows the access latency of each access in nanosecond scale. Figure 3a shows results of baseline where clflush flushes cache blocks in all caches including L4 DRAM cache without bypassing. As plotted, after flushed, the attacker obtains the block from L3 cache if the victim accesses the target block while obtaining the block from the main memory if the victim does not access the block. Exploiting the difference in access latency between L3 and DRAM (i.e., 27ns and 80ns), the attacker can infer the access patterns of the victim's to the L3 cache.

Figure 3b shows bypassing L3 only is not enough to defend against Flush+Reload attack without re-implementation of clflush. As baseline, the attacker obtains the target block from the main memory when reloading it if the victim does not access the block after flushed. However, if the victim accesses the block, bypassing only obtains the block from L4 DRAM cache. Consequently, depending on victim's access, the attacker obtains the block with the main memory or L4 DRAM cache, generating the timing differences.

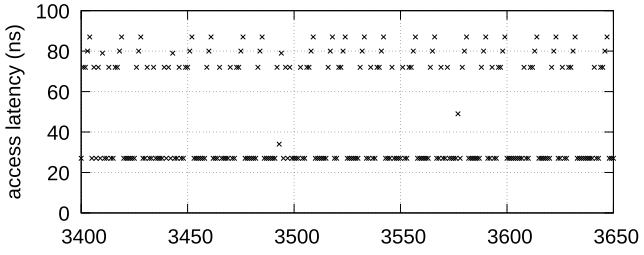
However, ByCA eliminates the timing differences by re-implementing clflush to not flush the block in L4 DRAM cache, as plotted in Figure 3c. Regardless of the victim's behavior, the attacker always obtains the target block from the DRAM cache. Therefore, the attacker cannot figure out the cache access pattern of the victim at all.

C. PERFORMANCE EVALUATION

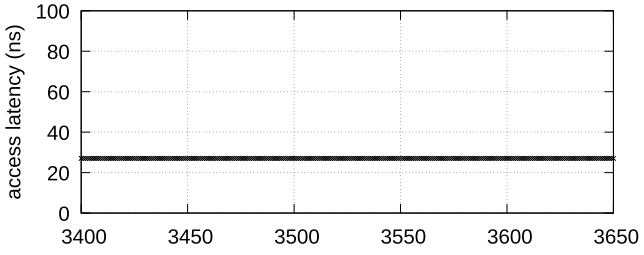
To investigate impact of our proposed architecture on performance, we compare ByCA with baseline in terms of



(a) `baseline` (Not bypassing L3, flushing blocks in L4 DRAM cache).



(b) `bypass only` (Bypassing L3, flushing blocks in L4 DRAM cache).



(c) `ByCA` (Bypassing L3, not flushing blocks in L4 DRAM cache).

FIGURE 3. Comparison of access latency.

TABLE 2. Victim's IPC.

	`Baseline`	`ByCA`
Repetitive attack	1.97	2.09
Temporal attack	2.17	2.17

Instruction Per Cycle (IPC) in Table 2. For `baseline`, we model L4 DRAM cache without bypassing implementation of `ByCA`. We run two different attack scenarios, `repetitive attack` and `temporal attack`. We repeat Flush+Reload for `repetitive attack` while conducting Flush+Reload only once at the beginning for `temporal attack`.

As `ByCA` replaces main memory accesses for flushed lines with L4 cache accesses, the IPC of victim rather slightly improves with `ByCA` even though the attacker conducts Flush+Reload repetitively. Furthermore, `baseline` and `ByCA` show the same IPC with temporal attack since the victim mostly obtains flushed cache blocks from private caches. Therefore, with these attack scenarios, `ByCA` does not need to clear `bypass bit`. However, if applications flush a quite number of cache blocks using `clflush` instruction and reuse the blocks, bypassing L3 cache may degrade performance. We remain policies to clear `bypass bit` as our future work.

D. STORAGE OVERHEAD

`ByCA` requires one bit (i.e., bypass bit) for cache blocks placed on L1, L2, and L4 DRAM caches while not requiring the extra bit for cache blocks placed on L3 cache. Therefore, with 64B size cache blocks, `ByCA` requires about 0.2% extra storage for caches that maintain `bypass bit`. In our evaluation, since we use 32KB L1 and 256KB L2 caches, `ByCA` additionally requires 64B and 512B for L1 and L2 caches, respectively. Since `ByCA` maintains `bypass bit` only at L1 and L2 caches, it requires about 0.025% extra storage in our evaluation scenario where four cores, each of which is with a private L1 (32KB) and L2 (256KB) cache, shares an 8MB L3 cache. Furthermore, if L1 and L2 caches are inclusive, `ByCA` does not need to maintain `bypass bit` for L1 caches.

As discussed in Section IV, `ByCA` uses a bit among unused bits in a tag field. As our experimental configuration, with a 1GB DRAM cache with 48-bit physical address, each block uses 18 bits for the tag and 2 bits for the valid bit and dirty bit. Since the size of the tag field is 8B (64 bits), each DRAM cache block has 44 unused bits in the tag field. `ByCA` uses a single bit as `bypass bit` among the 44 bits. Therefore, `ByCA` does not require additional storage to maintain `bypass bit` for the L4 DRAM cache.

VI. DISCUSSION

A. CLEARING `BYPASS BIT` AND PERFORMANCE DEGRADATION

Although we do not discuss about clearing `bypass bit` in the previous sections, bypassing L3 cache may have impact on the performance. Inherently, compared with the conventional `clflush` implementation and cache hierarchy without L4 DRAM cache, our approach without clearing the `bypass bit` does not notably degrade performance of the victim while co-running with the attacker conducting Flush+Reload attacks since the victim mostly obtains the flushed cache blocks from the main memory; note that our approach just requires simple hardware modification for bypassing and maintaining the history for the flushed blocks. Our approach rather improves the victim's performance while co-running with the attacker since the victim obtains the flushed blocks from L4 DRAM cache with lower latency than the main memory.

Although we show that `ByCA` does not degrade the performance notable even with the temporal attack, if the number of target blocks flushed by the attacker increases considerably, since the flushed blocks will permanently bypass the L3 cache even without any attacker, `ByCA` may suffer from performance degradation without clearing `bypass bit` when the victim does not co-run with the attacker that flush target blocks repetitively. For example, although the attacker flushes the cache blocks once and does not conduct the attack repeatedly, the victim always obtains the blocks from L4 DRAM caches not from the L3 cache; the victim will obtain the block from the L3 cache without the attacker. Therefore, to avoid

the performance degradation by bypassing L3, we need to clear the `bypass bit` periodically. The length of the clearing period should be carefully determined since too long period can degrade performance while too short period can load the target blocks to L3 cache when the attacker co-runs. However, in most cases, the number of target blocks will not be large; the number of target blocks for GnuPG is three.

B. DEFENDING AGAINST OTHER CACHE SIDE-CHANNEL ATTACKS EXPLOITING L4 DRAM CACHE

In this work, we propose an approach that can defend against Flush+Reload attack while not considering other cache side-channel attacks, such as Prime+Probe [13] and Evict+Reload [14]. While Flush+Reload attack flushes the target block, other attacks evict target blocks by filling other cache blocks in the cache memory. Compared with monitoring flushed blocks, monitoring evicted blocks is challenging since all blocks can be evicted while a small number of blocks is targeted by `clflush` instruction. Therefore, we need to predict target blocks based on access patterns or eviction frequency of blocks, and selectively monitor only the blocks predicted as target blocks of the attacker. After setting the `bypass bit` for the blocks predicted as target blocks of attacker, bypassing L3 cache for the blocks exploiting DRAM cache can defend against the eviction-based attacks in the almost same way. We remain supporting other cache side-channel attacks along with target block prediction as our future work.

C. DEFENDING AGAINST SPECULATIVE ATTACKS EXPLOITING FLUSH+RELOAD ATTACK

Speculative attacks such as Meltdown [2] and Spectre [3] exploit cache side-channel attacks to access privileged data. For example, Meltdown intentionally raises an exception by accessing the region where the attacker cannot access, such as kernel memory region. While handling the exception, the core executes instructions speculatively that must not be executed. As the attacker cannot directly access the privileged region, the attacker creates side-channel with the speculatively executed instructions.

However, since the attacker exploits its own cache blocks for Flush+Reload attack, the attacker flushes/reloads the target block from/to its private cache. Therefore, it requires techniques to defend against Flush+Reload attack targeting private caches (e.g., L1 cache) along with techniques for shared caches. We remain an approach that defend against such speculative attacks as our future work.

VII. CONCLUSION

We propose `ByCA` that nullifies a representative cache side-channel attack, Flush+Reload, exploiting L4 DRAM cache. `ByCA` re-defines and re-implements `clflush` instruction to not flush a cache block from the L4 DRAM cache while flushing blocks from other level caches. After that, `ByCA` bypasses the L3 cache when accessing flushed cache blocks that can be targeted by the attacker. Consequently, `ByCA` eliminates timing differences when the attacker accesses the target blocks since the attacker always obtains the target blocks from the L4 DRAM cache regardless of the victim's accesses. To distinguish between flushed blocks and other blocks, `ByCA` stores whether the cache block was flushed by `clflush` or not in an unused bit, i.e., `bypass bit` in each tag entry of L4 DRAM cache, which adds negligible overheads when used with Alloy Cache architecture. Using the simulated environment, we demonstrate that `ByCA` completely eliminates the timing differences, nullifying Flush+Reload attack. Furthermore, we show that `ByCA` does not degrade victim's performance due to the L3 bypassing when the victim co-runs with the attacker.

In this work, we propose `ByCA` for systems that allow user applications to flush cache blocks in the unprivileged mode. However, an attacker can conduct other types of cache-side channel attacks rather than Flush+Reload, such as Prime+Probe and Evict+Reload attacks, against systems that do not allow such flush instruction (e.g., ARMv7) in the unprivileged mode; ARMv8 allows user applications to flush cache blocks only when it is specifically enabled. We remain extensions to defend systems against Prime+Probe and Evict+Reload attacks exploiting the L4 DRAM cache as our future work.

REFERENCES

[1] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2014, pp. 719–732.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, and D. Genkin, "Meltdown: Reading kernel memory from user space," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2018, pp. 973–990.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[4] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[5] J. Kim and Y. Kim, "HBM: Memory solution for bandwidth-hungry processors," in *Proc. IEEE Hot Chips 26 Symp. (HCS)*, Aug. 2014, pp. 1–24.

[6] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proc. IEEE Hot Chips 23 Symp. (HCS)*, Aug. 2011, pp. 1–24.

[7] S. Mittal and J. S. Vetter, "A survey of techniques for architecting DRAM caches," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1852–1863, Jun. 2016.

[8] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for Web search," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 643–656.

[9] A. Shahab, M. Zhu, A. Margaritov, and B. Grot, "Farewell my shared LLC! A case for private die-stacked DRAM caches for servers," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 559–572.

[10] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 235–246.

[11] C.-C. Huang and V. Nagarajan, "ATCache: Reducing DRAM cache latency via a small SRAM tag cache," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation (PACT)*, 2014, pp. 51–60.

[12] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 454–464.

[13] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers' Track RSA Conf. (CT-RSA)*, 2006, pp. 1–20.
[14] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2015, pp. 897–912.
[15] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 51–67.
[16] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*. Cham, Switzerland: Springer, 2016, pp. 279–299.
[17] W. Koch *et al.*, "The GNU privacy guard," OpenSSL Softw. Found., Tech. Rep., 2003. [Online]. Available: http://www.gnupg.org
[18] M. J. Cox, R. S. Engelschall, S. Henson, and B. Laurie, "The OpenSSL cryptography and SSL/TLS toolkit," OpenSSL Softw. Found., 2011. [Online]. Available: https://www.openssl.org
[19] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *Proc. 25th Int. Conf. Comput. Design*, Oct. 2007, pp. 55–62.
[20] V. Young, P. J. Nair, and M. K. Qureshi, "Dice: Compressing dram caches for bandwidth and capacity," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 627–638.
[21] C. Chou, A. Jaleel, and M. K. Qureshi, "CANDY: Enabling coherent DRAM caches for multi-node systems," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 35–47.
[22] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.
[23] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure dynamic cache partitioning for efficient timing channel protection," in *Proc. 53rd ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2016, pp. 74–79.
[24] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 974–987.
[25] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, p. 27, 2017.
[26] C. Intel, "Improving real-time performance by utilizing cache allocation technology," Intel Corp., Tech. Rep., Apr. 2015.
[27] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "RIC: Relaxed inclusion caches for mitigating llc side-channel attacks," in *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.
[28] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 347–360.
[29] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–12.

MINWOO JANG received the B.S. degree in convergence engineering from the Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, South Korea, in 2018, where he is currently pursuing the combined Ph.D. degree with the Department of Information and Communication Engineering. His current research interests include computer architecture, process in memory systems, and mitigating cache attacks such as cache side channel attacks, meltdown, and spectre.

SEUNGKYU LEE received the B.S. degree in convergence engineering from the Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, South Korea, in 2018, where he is currently pursuing the combined Ph.D. degree with the Department of Information and Communication Engineering.

His current research interests include network I/O optimization in virtual environments, mitigation against cache side-channel attacks, and mobile systems.

JAEHA KUNG (Member, IEEE) received the B.S. degree in electrical engineering from Korea University, Seoul, South Korea, in 2010, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2012, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2017.

He is currently an Assistant Professor with the Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, South Korea. His current research interests include energy-efficient digital accelerators for deep learning, distributed learning systems, hardware architecture for machine intelligence, and high-performance solver for dynamical systems.

DAEHOON KIM (Member, IEEE) received the B.S. degree in computer science from Yonsei University and the Ph.D. degree in computer science from the KAIST, in 2008 and 2014, respectively. He is currently an Assistant Professor with the Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology (DGIST). His research interests include computer architecture, operating systems, system virtualization, and cloud computing.