$\mathcal{F}lush{+}early\mathcal{R}$ELOAD: Covert Channels Attack on Shared LLC Using MSHR Merging

Aditya S. Gangwar
Dept. of CSE, IIT Ropar
Punjab, India
2021csm1001@iitrpr.ac.in

Prathamesh N. Tanksale
Dept. of CSE, IIT Ropar
Punjab, India
prathamesh.21csz0003@iitrpr.ac.in

Shirshendu Das
Dept. of CSE, IIT Hyderabad
Telangana, India
shirshendu@cse.iith.ac.in

Sudeepta Mishra
Dept. of CSE, IIT Ropar
Punjab, India
sudeepta@iitrpr.ac.in

***Abstract*—Modern multiprocessors include multiple cores, all of them share a large Last Level Cache (LLC). Because of the shared nature of LLC, different timing channel attacks can be built by exploiting LLC behaviors. Covert Channel and Side Channel are two well-known attacks used to steal sensitive information from a secure application. While several countermeasures have already been proposed to prevent these attacks, the possibility of discovering new variants cannot be overlooked. In this paper, we propose a covert channel attack designed to circumvent the state-of-the-art countermeasure for preventing the Flush+Reload attack. Experimental results indicate that the proposed attack renders the current state-of-the-art countermeasure futile and ineffective.**

***Index Terms*—Covert Channel Attack, Side Channel Attack, Shared Memory, Last Level Cache, Flush+Reload.**

I. Introduction

The LLC in current multicore processors is shared by all cores, and each core has its own upper-level private caches. Because of the shared nature of LLC, numerous timing channel attacks target it in order to leak sensitive information. The side channel attack (SCA) and the covert channel attack (CCA) are two examples of such attacks [1]. In SCA, a malicious application acquires confidential information from a victim application. Contrastingly, in CCA, two malicious programs are involved: a spy and a Trojan. The Trojan runs within a secure core and possesses access to sensitive information. As per the security protocol, direct communication between the Trojan and the spy is restricted. Instead, the Trojan establishes a communication route through CCA to transmit confidential information to the spy. SCA and CCA are both calculated using the temporal difference between LLC hits and misses. Prime+Probe (P+P) [2] and Flush+Reload (F+R) [3] are two prominent methods for launching such attacks. The P+P requires no shared memory address between the spy (attacker) and the Trojan (victim), whereas the F+R requires at least one page of data that must be physically shared by the spy (attacker) and the Trojan (victim).

This work is based on the F+R attack and its variations. The F+R-based CCA attacks require a common block b between the spy and the Trojan. First, the spy issues the flush command, which clears b from all cache levels. The Trojan then either accesses or does not access b dependent on the bit that has to be sent. Assume the Trojan accesses b and gets it back to the LLC for bit-1. In the case of bit-0, it has no effect, and b is still absent in LLC. The spy then requests to reload b. This is known as the reload phase. During the reload, it is a cache hit if Trojan already loads b into the LLC. Otherwise, the reload request faces a cache miss at the LLC. The spy can now anticipate whether the Trojan wants to send bit-0 or bit-1 by observing the time gap between an LLC hit and an LLC miss. Trojan can convey the entire secret information to the spy by repeating this operation numerous times. In the case of SCA, the attacker can use this attack to determine whether or not the victim has accessed b. The primary emphasis of this work is CCA. However, the work is also relevant to SCA.

Cache randomization [4], [5] or cache partitioning [6]–[8] can be used to prevent P+P attacks. The prevention of shared memory-based reuse prediction attacks is more crucial. The F+R attack, for example, is conceivable by targeting only a single shared cache block. Cache randomization approaches are ineffective against such attacks. This is due to the fact that the attack does not need to know which set the block is mapping to. Furthermore, partitioning-based solutions are only suitable to multi-programmed applications that do not use address sharing. The flush can be implemented via the *cflush* instruction or by exploiting the replacement policy's features to evict the blocks from the LLC. There could be different ways a flush can be implemented in modern CPUs. We used existing works [3], [9] as a foundation for this.

To counter F+R and related attacks, the authors of [10] introduced a concept called TimeCache. The same concept is applied to preventing coherence-based attacks in [11]. According to TimeCache, the first access to a block in LLC for any core is considered a miss, and the request is forwarded to main memory. If the fetched block is already present in the LLC, it is dropped, and the current LLC block is transmitted to the core. The fundamental goal of a forceful miss in TimeCache is to force the attacker to observe equal time for the first accesses of all blocks. Whether or not the Trojan has accessed the block, the attacker experiences large access latency during the reload phase. The concept eliminates F+R with 100% accuracy with minimal performance impact. TimeCache is a crucial contribution to making the LLC safe because other strategies such as randomization and cache partitioning cannot be utilized to avoid F+R.

In this work, we investigated the possibility of a F+R attack on top of TimeCache [10], [11]. We call this attack

979-8-3503-4859-0/DATE24/© 2024 EDAA

Flush+*early*Reload (F+*e*R). The main idea behind this attack is merging the block requests in the Miss Status Holding Register (MSHR) of LLC. In its current state, the MSHR does not send multiple requests (with the same block address) to main memory. Only the first request is sent to main memory, and the rest are merged and wait for the block from main memory. According to TimeCache, the spy always has an access time greater than T_{mm} during reload. T_{mm} is the shortest time required to fetch a block from main memory to LLC. However, an early reload request from the spy may merge in MSHR with the Trojan's request. In such a circumstance, the spy observes access latency that is much lower than T_{mm}. This is due to the fact that merging occurs after the first request has already been sent to main memory. If, on the other hand, a block is not requested by the Trojan, there is no merging in MSHR for the spy's request, and the request is delivered to main memory. As a result, a calculated reload by the spy, with an MSHR merging as late as feasible, makes the time difference significant in establishing CCA. Changing the MSHR logic to enable multiple requests will not prevent this attack but shift it to DRAM queue where also the requests can merge. We strongly believe merely shifting the problem from LLC MSHR to DRAM queue is not the right way to go ahead and build secure systems.

The remainder of this paper is organized as follows. The background and related works are discussed in section II. The proposed attack and the threat model are discussed in Section III. Experimental analysis are shown in Section IV. A countermeasure is discussed in Section V. Section VI finally concludes the paper.

II. Background

A. Two existing F+R defense techniques

TimeCache, as described in Section I, causes an LLC miss on the initial access of a block by a core. TimeCache uses an *s-bit*, which is initially set to 0, to verify whether a cache block has been previously accessed by a core. One *s-bit* per core is required for each block in LLC. The *s-bit* is set when the core first accesses the cache block. If a block's *s-bit* for core C_x is 0, the LLC hit is ignored and the request (from C_x) is forwarded to DRAM. The LLC destroys the dummy copy of the block after getting it from DRAM and deliverers the already-present copy to C_x. The *s-bit* for C_x is only set once the dummy block is dropped. This action requires the spy to observe long access times for each access during the reload phase. After issuing the flush instruction, the spy became ignorant of the Trojan's actions. This is how TimeCache protects against cross-core shared memory attacks, such as F+R attacks. According to TimeCache, the first cache miss produces only a 1.13% decrease in performance.

The authors have enhanced the TimeCache idea in [11] to defend against coherence-based attacks. The coherence-based attack is also a version of the F+R attack. The authors only discussed the attack in terms of SCA. Two different attacker applications must run in two different cores. Consider that the first and second attackers are running in core C_x and C_y respectively. The first attacker flushes the cache before reloading the shared block to measure latency. Following the flush, the second attacker always gets the block in exclusive (E) mode, indicating that the block is exclusive in LLC. The latency seen by the first attacker to access the block (during reload) is dependent on whether or not the victim accesses the block. Take into account that the victim application is executing on core C_v. In the event of victim access, the coherence protocol forwards the access request to C_y's private cache before making the block shared by both C_y and C_v. When the victim does not access the block, it is solely controlled by C_y's private cache. If the status of the block in LLC is exclusive during the reload phase, the first attacker experiences high access latency. This is because the coherence protocol must pass C_x's request to C_y's private cache. These temporal disparities can be used to launch a F+R attack. The authors demonstrated that this attack can also be avoided by using TimeCache.

III. Proposed F+eR Attack

A. Threat Model

In this work, we have assumed a multicore processor with four cores: C_0, C_1, C_2, and C_3. Each core has its own private L1 and L2 caches. All the cores share a common, large-sized L3 cache as LLC. The CCA attack based on F+R is implemented in this setup. We assume that spy and Trojan run in C_0 and C_1. The remaining two cores run innocent applications. The applications from the SPEC CPU 2006 benchmarks [12] are used to create different mixes to run along with the spy and Trojan. The purpose of running SPEC CPU applications with spy and Trojan is to assess their impact in the presence of various innocent applications. The implementation details of the spy and Trojan are elaborated in Section IV.

When LLC receives a read or write request from the upper cache level, it first places the request in LLC's request queue. The controller of the LLC responds to these requests in a certain order. If the requested block is not found in LLC, a request is sent to main memory to get it. At the same time, an entry is made in the MSHR table, and the status of the block is also added to the directory as a transient block. Transient signifies that the block cannot be accessed until it is retrieved from main memory. During this transient condition, the coherence protocol simply adds a new requester as a sharer in the directory for any further requests to the same block. After receiving the block from main memory, it is distributed to all cores using the sharer information stored in the directory. When the block is already existing in LLC and its state is shared (S), the request receives an LLC hit. As a result, in order to implement TimeCache, some logic in the coherence protocol must be changed. We assumed that such minimal changes in the controller are already in place to support TimeCache.

B. The Primary Attack Idea

The three stages (*flush, wait*, and *reload*) of an F+R attack are already discussed in Section I. In an F+R attack, both spy

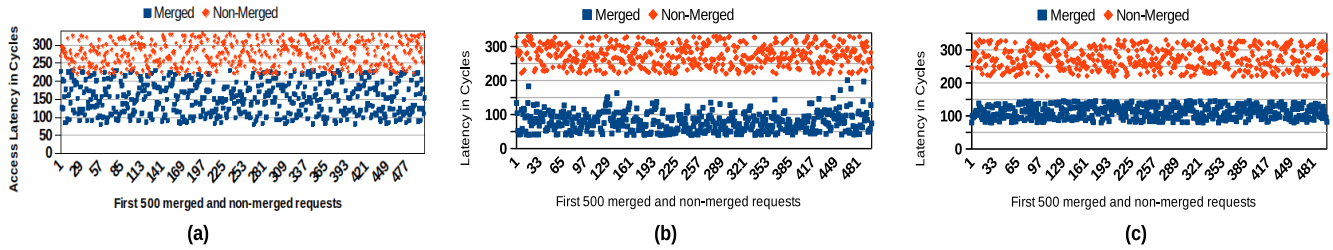

Fig. 1: Access latency observed by the spy during the reload phase both in case of MSHR merging and non-merging. The result shown for three SPEC CPU benchmark mixes: (a) *leslie+gobmk*, (b) *lbm+namd*, and (c) *milc+zeus*.

and the Trojan maintain a sufficient time gap between any two stages. The reload starts after a sufficient gap so that Trojan, if it wants, can fetch the block and place it in the LLC before the reload. TimeCache prevents F+R attacks by sending the first access request for a block from each core to the main memory. Forcing such a main memory request always shows high access latency during the reload phase of spy. Hence, the spy cannot guess the bit sent by the Trojan.

Somehow if the block requests of Trojan (wait phase) and spy (reload phase) merge in MSHR, then spy observes less access latency than Trojan for accessing the same block from main memory. This is because the request from the Trojan was already sent to memory when the request from the spy was merged with it in MSHR. When the block reaches LLC, it is sent to both Trojan and spy at the same time. But because of merging, the spy gets it in less time than the Trojan. On the other hand, if there is no MSHR merging, then the spy observes the normal longer latency because of fetching the block from main memory. TimeCache has assumed that spy and Trojan always leave enough gaps between two stages, and hence the MSHR merging concepts are ignored. However, with simple calculations, a spy can easily send reload requests such that they can merge with the Trojans request in MSHR. The most important point here is to make the request merged in the MSHR, and merging should be as late as possible. Such a delay can be calculated easily, as the spy and Trojan already sync to establish the F+R attack. The same synching logic can be used to start the reload early.

Consider that T_c is the time required to communicate a request from the core to the LLC. T_q is the average waiting time in the request queue for LLC. T_a is the LLC access time. T_{mm} is the minimum time required to get the block from main memory after the request is issued from the LLC. The minimum time required to access a block if hit in LLC is $T_{hit} = 2T_c + T_q + T_a$. In the case of a miss, the minimum time required is $T_{miss} = T_{hit} + T_{mm}$. In a modern processor like Intel Icelake sunny cove architecture [13], T_{mm} contributes $\sim 90\%$ in this delay. The value of T_a is known. The T_c is the average delay for on-chip communication between two nodes within the chip. The value of T_c does not vary drastically unless the on-chip interconnects are under any kind of DoS attack [14]. Hence, the value of T_c between two nodes is predictable. Similarly, T_q is also predictable. We have observed that the T_q value changes with different benchmarks. However, the variation is not significantly high. Since T_a is known and T_c is predictable, the spy can calculate T_q periodically. For a confirmed MSHR merge, the spy should issue the reload between $T_{hit} + p$ and $T_{hit} + (T_{mm} - (T_c + T_q))$, after finishing the flush. Here p can be considered a padding delay, and $p < (T_c + T_q)$. Any time before and after may miss the MSHR merging. Experimentally, we have observed 100% accurate merging with p between $0.5T_{mm}$ and $0.8T_{mm}$. Hence, spy has a sufficiently wide window to make a guaranteed MSHR merge.

Figure 1 shows the access latency observed by the spy during the reload phase, both in the case of MSHR merging and non-merging. The access latency shown in the figure is for the first 500 merged and non-merged reload requests of spy. For this experiment, the spy and Trojan run in C_0 and C_1, while the remaining two cores run benchmark mixes as discussed in Section III-A. The purpose of running spy and Trojan with benchmark mixes is already discussed in Section III-A. From the figure, the time differences between merging and non-merging requests are clearly observable. In the case of merging, the latency observed by the spy is less than 200 cycles for most of the time. In the case of non-merging, the latency is above 200 cycles in every case. However, Figure 1(a) and 1(b) show some overlapping between merging and non-merging times. This is because some of the merging happened much earlier. A late merging with a p value between $0.8T_{mm}$ and $0.9T_{mm}$ reduces all such overlapping as shown in Figure 1(c). Another interesting fact is that the time difference observed by the spy is almost similar when it runs with different benchmark mixes.

The step-by-step description of Flush+*early*Reload (F+*e*R) attack is given below:

- *Flush*: Attacker flushes the shared block from all cache levels.
- Trojan fetches the shared block in cache if it wants to send bit-1, otherwise no action taken from Trojan.
- *EarlyReload*: After the flush, the spy waits for a random cycles between $T_{hit} + p$ and $T_{hit} + (T_{mm} - (T_c + T_q))$ before sending a reload request for the shared block. Once the block is being accessed the attacker calculates the time. Access time less than a experimentally calculated threshold means Trojan sends bit-1, otherwise bit-0.

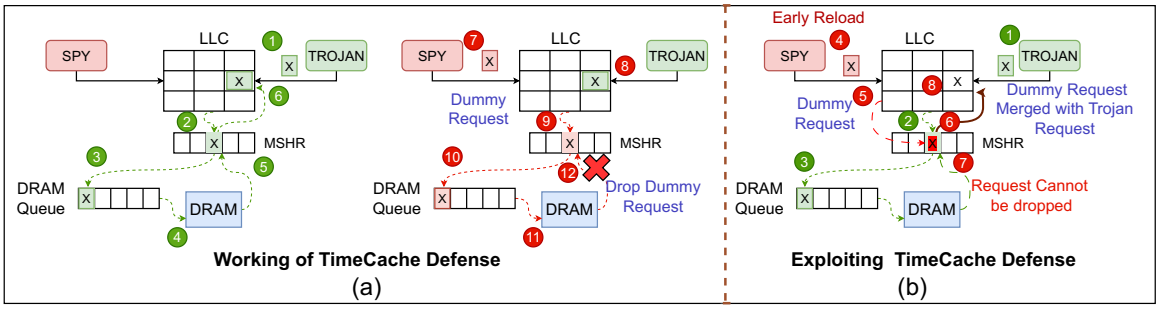

Fig. 2: Comparison of TimeCache defense and how F+eR exploits it.

The Figure 2(a) shows how TimeCache prevents the F+R attack. After the flush phase, first assume that the Trojan needs to send bit-0. Hence it has not requested for the block x. During the reload phase, spy requests for the block x. Since x is not present in LLC, a directory entry is made in LLC (1) and the request is sent to main memory through MSHR and DRAM queue (2), (3), and (4). When DRAM returns x, the MSHR entry is removed (5) and x is placed in LLC (6) and eventually sent to the spy. In this case spy observe a cache miss. In case, the Trojan wants to send bit-1, Trojan loads x before the reload phase of spy. When spy request for x (7) it is a hit in LLC (8). However, since spy is accessing the block first time the request is forwarded to the main memory through (9), (10) and (11). When the DRAM returns x, MSHR drops its entry as well as the block x received from DRAM (12). After that the original block present in the LLC is sent to the spy. Hence, in this case also, the spy observe a cache miss. Figure 2(b) shows how F+eR can exploit the TimeCache to perform attack. In case of bit-1, Trojan sends a request for x through (1), (2), and (3). Before receiving x by the MSHR from DRAM, spy initiates its reload phase and request for x (4). Now as per TimeCache, the request is also sent to MSHR (5). However, since an entry of x is already there in MSHR this request is merged with the old request (6). At (7) when the MSHR receives x from DRAM, the block cannot be dropped and written in LLC (8). In this scenario, spy observe less access time because of the MSHR merging.

Observation: If the duplicate requests may pass through the MSHR, their chances of being merged in the main memory queue cannot be ignored. Bypassing the security-related responsibilities to the main memory cannot be considered a good solution.

IV. Experimental Analysis

We simulate the CPU as well as the memory system, which consists of caches, TLBs, memory controller, and main memory, using a modified version of the Champsim simulator [15]. ChampSim is equipped with a detailed memory hierarchy, we modified the simulator to incorporate the proposed threat model. The system parameters used is mentioned in Table I.

TABLE I: Simulation parameters of the baseline system.

Core	Out-of-order, 4 GHz
	STLB: 2048 entries, 16-way, 8 cycles
L1I/L1D	32 KB, 8-way, 4 cycles/48 KB, 12-way, 5 cycles
L2	512 KB 8-way associative, 10 cycles, LRU
LLC	2 MB/core, 16-way, 20 cycles, LRU
MSHRs	8/16/32 at L1I/L1D/L2, 64/core at the LLC
DRAM	1 channel, 6400 MTPS, 64-entry RQ/WQ, 4 KB row-buffer, open page, t_{RP}:50 cycles, t_{RCD}: 50 cycles, t_{CAS}: 50 cycles

We have considered certain predetermined memory addresses that both the spy and the Trojan request for executing the F+R attack. We have a predetermined 32-bit number that is repeatedly sent by the Trojan to the spy.

Figure 3 shows the access latency observed by spy during the entire execution. The access time shown in y-axis is the latency observed (in cycles) by spy for accessing the block during its reload request. The bit shown over x-axis is the bits the Trojan is willing to transfer in that attack round. Each row in the figure is for one benchmark mix, where the attack runs along with the benchmark. Because of the space limitation we could not show the results of all the benchmark mixes in the figure. However, the observations are almost similar in all the experiments. The first column shows the latency observed during the F+R attack and the second column is for the TimeCache. The third column is for our proposed F+eR attack. It can be seen that the F+R attack has a substantial timing gap between bit-0 and 1. The spy always detects an access latency of less than 40 cycles for bit-1, whereas the latency for bit-0 is always greater than 200 cycles. When the attack is run with different benchmark mixtures, a similar disparity is seen. As a result, the F+R attack is successful in establishing a covert channel to secretly transfer information in our experiments. The attack established by F+R can be dismantled using TimeCache, as seen in the second column of the figure. The delay observed by the spy for both bit-0 and bit-1 is highly comparable. Consequently, the spy cannot predict the bit sent by the Trojan based on access latency.

The third column of the figure shows that the F+*e*R attack implemented on top of TimeCache can reestablish the covert channel for successfully transfer of the secret information from Trojan to spy. In case of F+*e*R, spy always observes an access latency less than 140 cycles for bit-0 and higher than 200

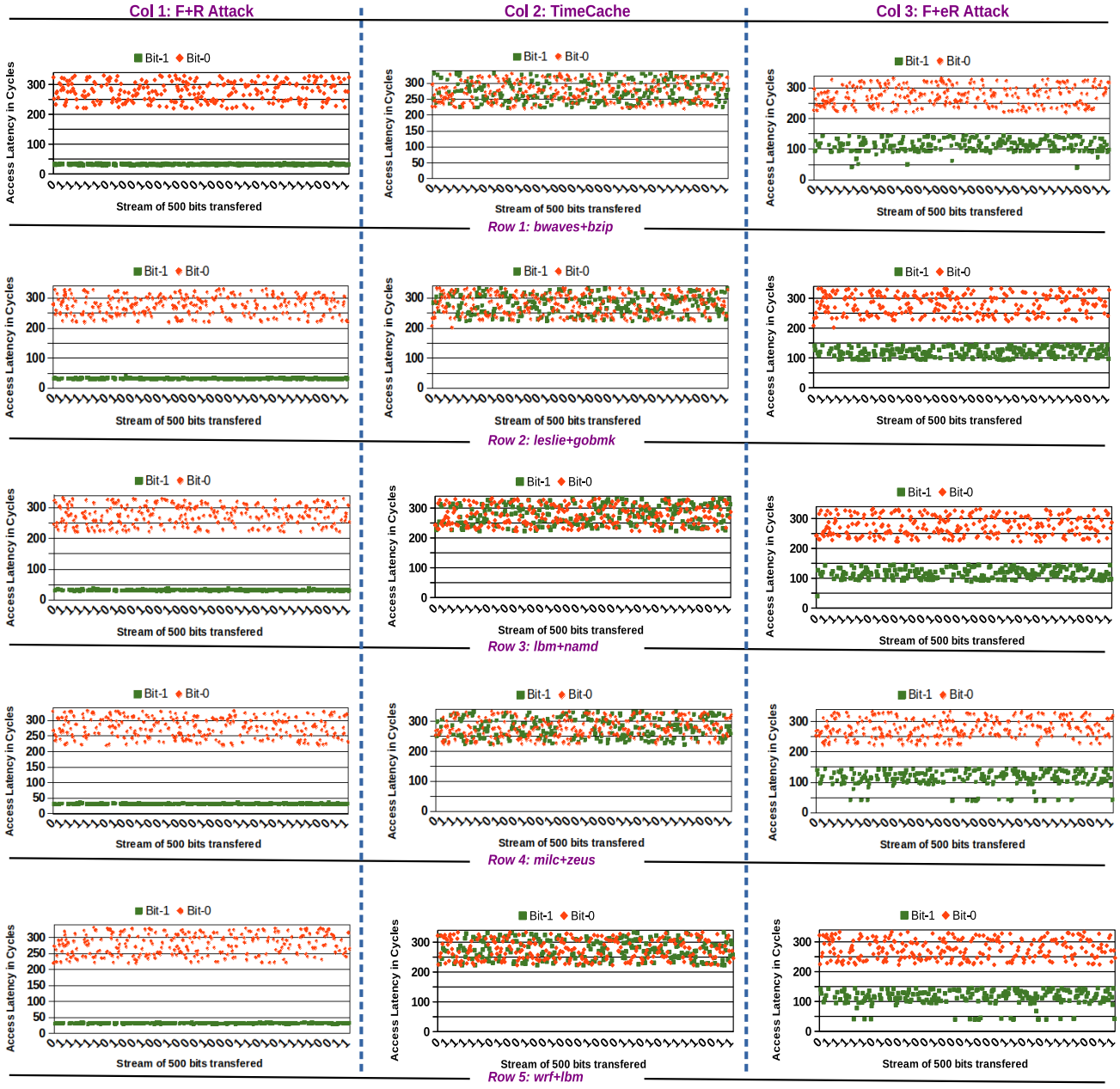

Fig. 3: The access latency observed by spy during the reload phase.

cycles for bit-1. These 60 cycles gap is significant enough to distinguish the bit which Trojan is willing to send to the spy. The access latency for bit-0 in F+*e*R is almost 100 cycles higher than the latency observed during F+R attack (col 1). This is because of MSHR merging. The reload request has to wait in the MSHR (after merging) until the block is fetched to the LLC. This wait adds an extra 100 cycles delay in F+*e*R. However, even after this additional latency, the time gap is still significant. However, as discussed in Section III-B, the spy has to send its reload request between $T_{hit}+p$ and $T_{hit}+(T_{mm}-(T_c+T_q))$ cycles after the flush operation. For this experiment, the spy sends the reload request after a random time gap between 150 and 170 cycles after the flush. The experiments shows that, a minimum 60 cycles gap is always possible if the spy sends the reload request within this window.

V. Countermeasure to prevent F+eR

The goal is to avoid MSHR merging by not sending the first access request to main memory if the block is already in LLC. We propose holding these initial access requests in a buffer for a random duration of time ranging from T_{mm} to $T_{mm}+d$. The padding time is denoted by d. This buffer is known as

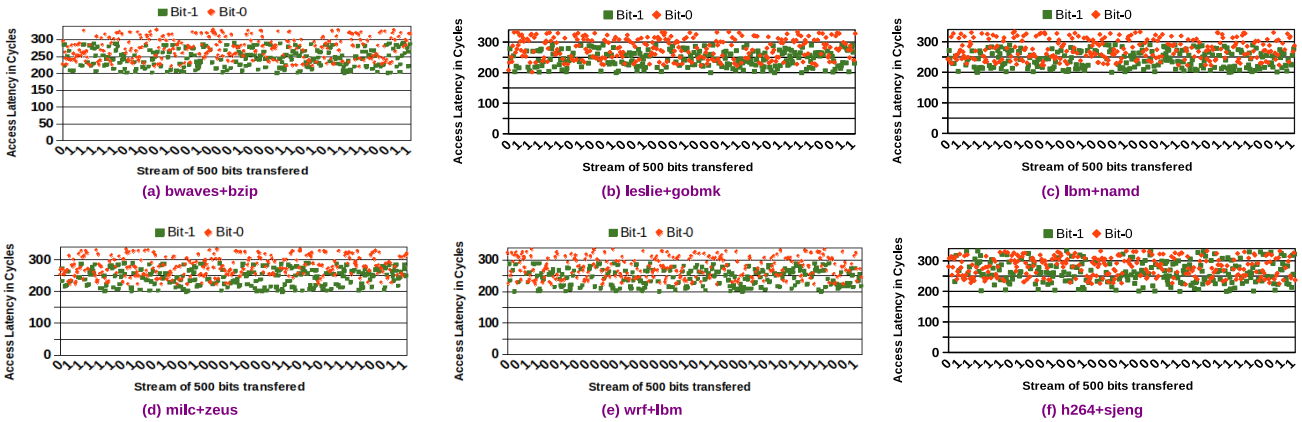

Fig. 4: The access latency observed by spy during the reload phase when countermeasure is implemented.

the delay buffer D_{bf}. After the waiting period, the request is reinserted to the LLC request queue. If the buffer becomes full, the FIFO policy is used to remove a request from the buffer. The ejected request is added to the LLC request queue. When the request returns to the LLC after exiting D_{bf}, the controller must identify that the request has already waited in D_{bf}. This is necessary since the request should not be entered into the buffer a second time. In our experiments, we consider d to be 50 cycles. The temporal diagram of the suggested countermeasure under an active F+*e*R attack is shown in Figure 4. The figure shows that the suggested countermeasure stops the attack by mixing the access latencies of bit-0 and bit-1. The countermeasure has a 1.7% performance overhead when compared to the baseline, which does not use TimeCache.

The buffer's size should theoretically be equal to the number of entries in L2's MSHR: the private cache just above the LLC. This is because the number of entries permitted in the MSHR between L2 and LLC is equal to the maximum number of requests from an L2 waiting at LLC. As a result, the buffer size for an N-core processor should be $N \times |MSHR_{L2}| \times s$. In this case, $|MSHR_{L2}|$ denotes the number of entries in L2's MSHR, and s is the amount of bytes necessary to store one request in the buffer. The number of L2-level MSHR entries considered for this paper is 32, as shown in table I. The buffer requires 0.75KB of additional storage.

VI. Conclusion

The paper introduces a compelling security attack F+*e*R, which nullifies the effectiveness of state-of-the-art countermeasure designed for F+R attacks. This attack capitalizes on an inherent micro-architecture feature that involves merging a request at the LLC MSHR, initially implemented for performance enhancement. Though the proposed countermeasure for F+*e*R attack by using a delay buffer is able to mitigate the attack to some extend, we strongly push for finding pertinent ways to handle request merges at LLC MSHR and DRAM Queues which become primary source of exploitation for attackers.

References

[1] J. Kaur and S. Das, "A survey on cache timing channel attacks for multicore processors," *Journal of Hardware and Systems Security*, vol. 5, no. 2, pp. 169–189, 2021.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*, pp. 605–622, IEEE, 2015.

[3] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 719–732, 2014.

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

[5] G. Saileshwar and M. Qureshi, "{MIRAGE}: Mitigating conflict-based cache attacks with a practical fully-associative design," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[6] J. Kaur and S. Das, "TPPD: Targeted Pseudo Partitioning based Defence for cross-core covert channel attacks," *Journal of Systems Architecture*, vol. 135, p. 102805, 2023.

[7] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–21, 2012.

[8] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient caches for trusted execution environments," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 451–468, USENIX Association, Aug. 2020.

[9] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: A Fast and Stealthy Cache Attack," in *Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299, 2016.

[10] D. Ojha and S. Dwarkadas, "Timecache: using time to eliminate cache side channels when sharing software," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 375–387, IEEE, 2021.

[11] D. Ojha and S. Dwarkadas, "Preventing coherence state side channel leaks using timecache," *IEEE Transactions on Computers*, vol. 72, no. 2, pp. 374–385, 2022.

[12] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[13] *SunnyCove microarhcitecture, May 2018. [Online]. Available:*. https://en.wikichip.org/wiki/intel/microarchitectures/sunnycove.

[14] B. Bisht and S. Das, "BHT-NoC: Blaming Hardware Trojans in NoC Routers," *IEEE Design & Test*, vol. 39, no. 6, pp. 39–47, 2022.

[15] "Champsim simulator." Online. Available:. https://github.com/ChampSim/ChampSim.