CacheRewinder: Revoking Speculative Cache Updates Exploiting Write-Back Buffer

Jongmin Lee
Korea University
Seoul, South Korea
flackekd@korea.ac.kr

Junyeon Lee
Samsung Electronics
Seoul, South Korea
junyeon2.lee@samsung.com

Taeweon Suh
Korea University
Seoul, South Korea
suhtw@korea.ac.kr

Gunjae Koo
Korea University
Seoul, South Korea
gunjaekoo@korea.ac.kr

***Abstract*—Transient execution attacks are critical security threats since those attacks exploit speculative execution which is an essential architectural solution that can improve the performance of out-of-order processors significantly. Such attacks change cache state by accessing secret data during speculative executions, then the attackers leak the secret information exploiting cache timing side-channels. Even though software patches against transient execution attacks have been proposed, the software solutions significantly slow down the performance of a system.**

In this paper, we propose CacheRewinder, an efficient hardware-based defense mechanism against transient execution attacks. CacheRewinder prevents leakage of secret information by revoking the cache updates done by speculative executions. To restore the cache state efficiently, CacheRewinder exploits the underutilized write-back buffer space as the temporary storage for victimized cache blocks evicted during speculative executions. Hence, when speculation fails CacheRewinder can quickly restore the cache state using the victim blocks held in the write-back buffer. Our evaluation exhibits CacheRewinder can effectively defend against transient execution attacks. The performance overhead by CacheRewinder is only 0.6%, which is negligible compared to the unprotected baseline processor. CacheRewinder also requires minimal storage cost since it exploits unused write-back buffer entries as storage for evicted cache blocks.

***Index Terms*—Secure Architecture, Transient Execution Attacks, Speculative Execution, Cache Side-Channels**

I. Introduction

Transient execution attacks are serious security threats since modern out-of-order (OoO) CPUs are vulnerable to such types of attacks. The attackers can easily access secret data in privileged regions and leak the secrets using cache timing side-channels. Since Spectre and Meltdown were first disclosed, several variants have been reported for a couple of years [1]–[4]. These attacks commonly exploit the speculative executions caused by various speculation sources such as branch prediction. The attackers access secret data using the loads executed on the predicted instruction paths (speculative loads). Although the executions of the speculative loads are discarded when the prediction fails, the microarchitectural changes (especially cache updates) done by the speculation loads remain in the processor. Then the attackers can leak the secrets by exploiting well-known side-channels. To mitigate the attacks we can apply several software patches that target the source of speculations. However, these software solutions result in significant performance degradation since speculative executions on OoO cores are essential features to improve the performance of modern CPUs. Hence, we require hardware-based solutions that can defend against the attacks.

As the transient execution attacks exploit the cache state updated by speculative loads, restoring the cache state can be an effective defense approach against the attacks. Note that the cache updates are *not discarded* even if the speculative loads are *canceled* in OoO processors. To restore the cache state to the original state before speculative executions, a processor core requires a roll-back mechanism using backup storage space for evicted or updated cache blocks. For instance, a cache block is removed from L1 data cache if a speculative load allocates new data. The data cache requires the evicted data to restore the cache state, however, the victim line is already removed from the cache. If a processor provisions extra storage space that can hold these victim lines temporarily, the cache state can be restored using the buffered victims. However, an extra buffer structure increases hardware cost. Besides, the performance of the OoO cores can decrease noticeably if the restore operations take many cycles.

In this paper, we propose CacheRewinder, an efficient hardware-based defense mechanism against transient execution attacks. CacheRewinder revokes the cache updates done by speculative loads when the speculation fails. CacheRewinder exploits the underutilized write-back buffer resources to temporarily store the victim blocks evicted by speculative loads. Thus CacheRewinder minimizes the additional hardware cost for implementing the proposed defense mechanism. Our evaluation reveals CacheRewinder can effectively defend against the transient execution attacks with negligible performance loss.

II. Background and Related Work

A. Transient Execution Attacks

Transient execution attacks are the recently disclosed security attacks that exploit microarchitectural vulnerabilities of modern OoO CPUs [1], [2]. To access secret data using *transient* loads executed speculatively, the attackers initiate speculative executions by manipulating the known prediction targets. For instance, Spectre-NG exploits the speculative store bypass to trigger speculative executions [4]. Then the attackers can decode the secret information that remains in the cache using cache side-channels. Prime+Probe approach detects the *misses* of the preloaded cache blocks replaced by secret data [5]. On the other hand, Flush+Reload approach identifies the cache *hits* installed by speculative loads that access secret data [6]. Consequently, most transient execution attacks make use of cache state changes caused by speculative loads.

B. Defense Mechanisms

1) Software approaches: Software-based defense mechanisms have been proposed to alleviate data leakages by

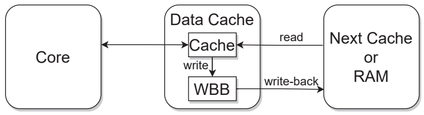

Fig. 1: Write-back buffer (WBB) on a write-back cache

transient execution attacks. The software mitigations focus on sidestepping the speculative executions that can be exploited by the attacks. For instance, the software patch for x86 CPUs against Spectre attacks employs *LFENCE* instructions to enforce execution orders [7]. This software patch degrades the performance of a system significantly. Another exemplar software-based solution against Meltdown-type attacks is to separate the kernel space from the page table of normal processes [8]. This approach also exhibits significant performance overhead since it can cause more frequent TLB flushes. To summarize, software defense mechanisms may result in significant performance loss since those approaches rely on restricting speculative executions.

2) Hardware approaches: As transient execution attacks exploit microarchitectural vulnerabilities, architectural solutions can be efficient approaches against the attacks. Note that the attackers rely on monitoring the cache state changed by speculative loads to decode secret data. Hence, the architectural approaches focus on *hiding* the cache state changes done by speculative loads. InvisiSpec employs the additional buffer called SpecBuffer to postpone the cache updates until speculation is resolved [9]. InvisiSpec first allocates the data requested by speculative loads in SpecBuffer then updates the data cache by transferring the data in SpecBuffer only when the speculation is hit. The performance overhead by InvisiSpec is not negligible since the loaded data are not visible until the speculation is resolved and additional cycles are taken to move data from SpecBuffer when the speculation is hit.

Another architectural solution is an *undo*-based mechanism proposed by ClecnupSpec [10]. CleanupSpec first allows cache updates by speculative loads and restores the cache to the original state only if the speculation fails. To minimize storage overhead, CleanupSpec exploits L2 cache space as *restore buffer* which preserves the data evicted from L1 cache. However, CleanupSpec's restoration process from L2 cache leads to performance degradation since transferring data from L2 cache takes a relatively large number of cycles. To reduce the performance loss by the restoration process, ReViCe includes the additional restore buffer that works like a victim cache between L1 and L2 caches [11]. ReViCe reduces the data transfer cycles for the restoration process using additional storage. However, such an approach is still vulnerable to the modified PRIME+PROBE type attacks that create many victims to make the fixed-sized restore buffer overflow.

III. Underutilized Write-Back Buffer

A write-back buffer (WBB) is a data buffer structure that temporarily holds the transactions to be updated to a lower-level cache as depicted in Figure 1. Writes to the lower-level cache can block the operation of the cache without WBB if the lower-level cache is not ready to process the write requests. Note that the lower-level cache cannot service write requests immediately if the lower-level cache is busy processing other

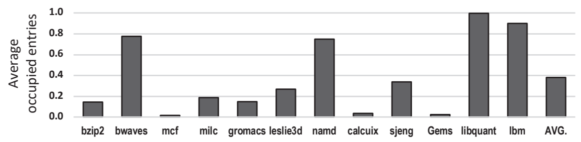

Fig. 2: The average number of occupied entries in WBB

requests. WBB stores write data temporarily to allow non-blocking write operation of the cache. In a write-back cache write requests are placed to WBB if a cache line is newly allocated and a *dirty* cache line is evicted from the cache. In the commodity CPUs, WBB is implemented as various data buffer structures such as a write-combining buffer [12] and an integrated line-fill buffer [13].

WBB is an essential data buffer structure that enables non-blocking cache operations. However, we observe that the resources in WBB are usually underutilized. Figure 2 exhibits the average number of occupied entries in WBB. For this experiment, we use Gem5 configured with the processor configurations listed in Table I and SPEC2006 benchmarks used for the evaluations in Section V. We count the number of occupied entries in WBB whenever a cache request misses in the data cache. We calculate the occupation of WBB by averaging the total count of the occupied WBB entries with the total cache miss count for each application. As shown in the figure the average occupation of WBB is around 0.38 and the number of the occupied entries is less than one for most applications. This observation motivates that we can exploit the underutilized data buffer resources in WBB for restoring the cache state during speculative executions. Namely, the empty entries in WBB can temporarily hold *clean* cache lines as well as dirty blocks evicted by speculative loads. Then the evicted cache blocks held in WBB can be returned to the cache to restore the cache state if speculation fails.

IV. CacheRewinder

In this paper, we propose CacheRewinder, an efficient and lightweight defense mechanism against transient execution attacks. CacheRewinder exploits the underutilized WBB entries to temporarily hold evicted cache blocks during speculative executions. Then CacheRewinder restores the cache state using the data stored in WBB if speculation fails. Note that the attackers can leak secret data by detecting the access time differences between the original cache state and the changed state resulting from speculative executions. As CacheRewinder makes use of the underutilized WBB as *restore buffer* space, CacheRewinder exhibits the following advantages. First CacheRewinder can minimize the performance overhead caused by the additional cycles for the restoration process since WBB is close to the data cache. Furthermore, CacheRewinder does not require additional buffer structures for storing victims evicted by speculative loads.

A. Overall Architecture

Figure 3 depicts the architecture of CacheRewinder implemented on a typical OoO processor. In the figure, we represent the modified or the newly added components as yellow boxes. Red-colored solid and dashed lines respectively denote data movement and control signals by CacheRewinder. As WBB works as a restore buffer, we add a data path

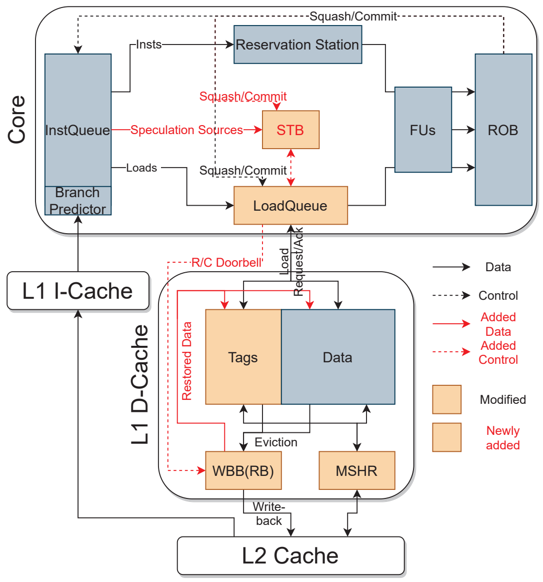

Fig. 3: Overview of CacheRewinder

from WBB to the data cache. The cache restoration process is controlled by a control signal called *R/C doorbell* which initiates *commit/restore* operations for the evicted cache blocks held in WBB. To generate the precise R/C doorbell signal, CacheRewinder tracks the status of speculative executions using a speculation tracking buffer (STB).

Now we describe the *restore* and *commit* operations by CacheRewinder. Modern OoO processors can execute instructions speculatively using various prediction techniques (e.g. branch prediction). If a prediction is missed, the speculative executions by the following instructions are canceled. In that case, CacheRewinder revokes the cache updates done by the speculative loads. On the other hand, CacheRewinder performs a commit operation immediately when the prediction succeeds.

Restore operation: When the processor squashes a speculative load from the reorder buffer (ROB) due to misprediction, the load queue (LQ) raises a *restore* doorbell signal along with the address of the speculative load. The restore signal and the address information are provided to WBB and the data cache. Then the cache line allocated by the speculative load is invalidated in the data cache and WBB transfers the evicted block to fill the invalidated line.

Commit operation: In order to minimize the performance loss by delayed updates of the victims, CacheRewinder applies a commit operation immediately when a speculation is resolved. For instance, if the prediction of the preceding branch is resolved, speculative loads become non-speculative and then the evicted cache blocks held in WBB can be committed safely. Hence if a prediction is resolved and hit, such information is provided to STB and LQ to create a *commit* doorbell signal. Then WBB receives the commit signal and the address information of the *resolved* speculative load. The victim blocks held in WBB are updated to the lower-level cache if the blocks are marked as dirty. Otherwise, the victim blocks are simply discarded. The detailed architecture of CacheRewinder will be described in the next sections.

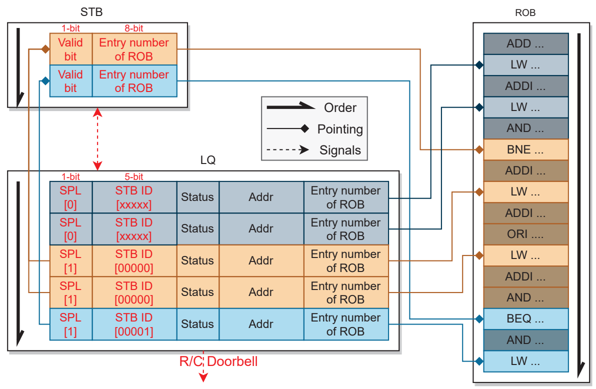

Fig. 4: An example of speculation handling by CacheRewinder

B. Restore/Commit Control

CacheRewinder creates the R/C doorbell signal to initiate restore/commit operations when a speculative execution is resolved. To raise the control signal at the proper time, CacheRewinder tracks the status of speculative executions. Figure 4 shows an example of how CacheRewinder monitors the speculation status using STB which interacts with ROB and LQ. STB employs a circular buffer structure to track the sources of speculations (e.g. branch instructions). Each entry of STB includes a valid flag and an index number of a ROB entry holding an instruction that provokes a speculative execution. When an instruction is dispatched and registered in ROB, the ROB entry number is registered in the STB entry pointed by the head pointer. In the example of Figure 4, *BNE* is a source of a speculative execution and the ROB entry number of *BNE* is registered in STB. We define a *speculation block* as a group of instructions executed speculatively by a prediction source, and in the figure, the instructions in the same speculation block are painted with the same color. Among the instructions executed speculatively due to *BNE* (painted with yellow color), loads are registered in both ROB and LQ at dispatch. We modify an LQ entry to include a speculation (SPL) flag, which indicates whether the load is a speculative load or not, and STB ID that points to a speculation source. In the example, the first and the second LQ entries include non-speculative loads thus SPL flags are reset. On the other hand, the third and the fourth loads are speculative loads following *BNE*, thus SPL flags are set and the entry number of the first STB entry is logged in STB ID fields. CacheRewinder creates restore/commit control signals using the data structures in STB, ROB, and LQ.

Restore control: CacheRewinder performs a restore operation by raising a restore doorbell signal. When a speculation fails ROB flushes the instructions within the corresponding speculation block. In that case, CacheRewinder invalidates the speculative loads in LQ and generates the restore doorbell signal. In the above example let us assume that *BEQ* is mispredicted. Then the corresponding entry (i.e. blue colored) in STB is invalidated and ROB flushes the instructions in the speculation block. The speculative load (*LW*) in the fifth entry of LQ is also invalidated when the load is squashed from ROB. Since the SPL flag for the load is set, LQ also creates the restore doorbell signal and provides the address of the

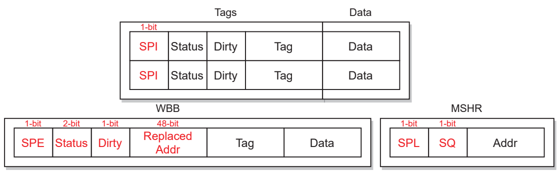

Fig. 5: Augmented fields in cache tags, MSHR and WBB

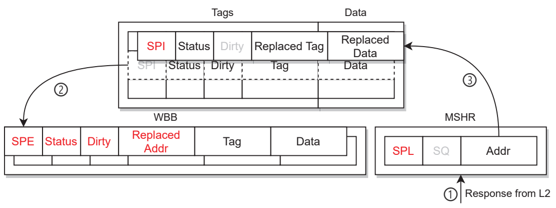

Fig. 6: Handling cache misses by speculation loads

speculative load. Then CacheRewinder performs the restore process. The detailed cache restoration mechanism will be explained in the next section.

Commit control: CacheRewinder raises the commit doorbell signal for the speculative loads as soon as the prediction of the corresponding speculative block is resolved. In the above example let us assume that the prediction for *BNE* is resolved and hit. Then ROB notifies the commit of *BNE* to STB using the ROB index number. STB looks up the entry number which contains the matched ROB index number. STB provides the entry number to LQ if the valid flag of the entry is set. LQ picks the speculative loads whose STB ID is matched with the provided STB entry number, then LQ provides the address of the speculative load and the commit doorbell signal to initiate the commit operation in WBB. Note that CacheRewinder does not need to wait until the speculative loads are committed in ROB since CacheRewinder can initiate the commit operation quickly using STB when a speculation source is resolved.

C. Cache and Write-Back Buffer Management

To support the operations by CacheRewinder we add more data fields on the existing cache tag and WBB entry as shown in Figure 5. The cache tag is augmented to include an SPI (speculative install) flag which indicates the corresponding cache line is allocated by a speculative load. A miss status handling register (MSHR) entry is expanded to include SPL (speculation load) and SQ (squashed) flags to handle the requests from the speculative loads. The SPL flag is set if a request by a speculation load is registered in MSHR. The SQ bit is used for nullifying the inflight request if the request was initiated by a speculation load but the request is canceled due to mis-speculation. Namely, MSHR ignores an incoming response if the SQ bit is set. A WBB entry is modified to support the restore operation. An SPE (speculative eviction) flag indicates the corresponding entry in WBB is evicted due to the cache line allocated by a speculation load. Cache tags and flags are also copied to a WBB entry since such information is required for restoring the cache state.

Now we describe how CacheRewinder handles the requests from speculative loads. We represent the sequences of cache events using the circled numbers in Figure 6. If a request is initiated by a speculative load and the request is missed in the data cache, the request is registered in one of the empty entries

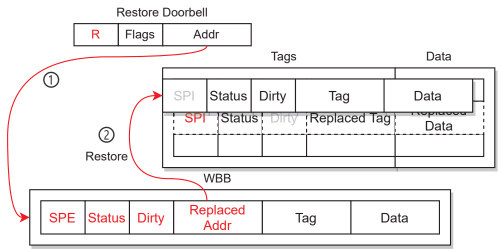

Fig. 7: The restore operation initiated by the restore doorbell

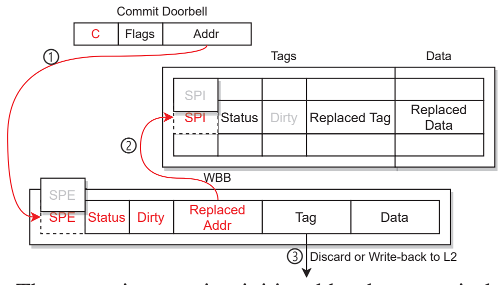

Fig. 8: The commit operation initiated by the commit doorbell

in MSHR. The SPL flag in the MSHR entry is set to indicate the request is made by a speculative load. ① Once the request is responded by the lower-level cache, the data cache picks a cache line to be evicted if all cache lines in the target set are already occupied. ② The victim cache line evicted from the data cache is transferred to WBB. Note that CacheRewinder places all *clean* and *dirty* victim lines in WBB if the eviction is caused by the *speculative* request. The tags and the flags of the victim line are also copied to the target entry of WBB. ③ Now the target cache line is occupied by the responded data. The SPI bit of the target cache line is set since this cache line is allocated by the speculative load.

Restore operation: The restoration of the cache state is initiated by the restore doorbell signal. Figure 7 depicts the restore operation. CacheRewinder simply performs the restore operation by replacing the target cache line with the victim blocks held in WBB. ① When the restore doorbell is raised, CacheRewinder looks up the victim cache lines using the address of the speculation load provided by LQ. Note that the index and the location of the target cache line can be easily figured out using the provided speculative load address. ② Then the target cache line allocated by the speculative load is invalidated, and the victim cache line in WBB replaces the target cache block. The SPI flag is reset since this cache line is not speculative anymore.

The response doorbell can be raised even if the request by a speculative load does not complete yet. In that case, the victim line is not found in WBB since the requested data does not reach the data cache. CacheRewinder handles this situation by exploiting the SQ bit in MSHR. If the request by the speculation load is found in MSHR and the restore doorbell is raised for that load, the SQ bit in the corresponding MSHR entry is set. Then the response from the lower-level cache is simply ignored.

Commit operation: Figure 8 describes CacheRewinder's commit operation which is initiated by the commit doorbell signal. Once WBB receives the commit doorbell signal, ① CacheRewinder looks up the target victim block using the

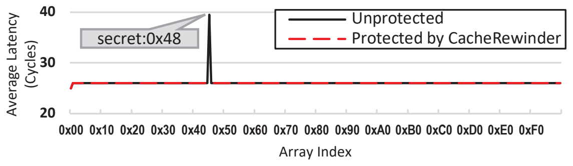

Fig. 9: Access latency for the primed data in L1-D cache

address of the speculative load. Then the SPE flags of the target victim block is reset since the evicted cache block is not speculative anymore and it can be handled by the normal operation of WBB. As such, if the target victim block is *dirty* the victim block is transferred to the lower-level cache. Otherwise, the WBB entry is simply invalidated since the victim is clean. ② Then the SPI flag in the cache block allocated by the speculative load is cleared.

WBB full: CacheRewinder simply disables the execution of speculative loads if all WBB entries are occupied. While WBB is full, the additional cache misses by the speculative loads can generate more *clean* or *dirty* victims which should be held in WBB. CacheRewinder does not allow the newly generated victims to bypass WBB or evict the existing *clean* victims in WBB. CacheRewinder's such policy makes the processor more secure against the potential PRIME+PROBE type attacks that make WBB overflow. Note that the data cache state cannot be restored from WBB if the victims are missing in WBB when the speculation fails. CacheRewinder may increase the chance of pipeline stalls during speculative executions since WBB can be occupied by *clean* as well as *dirty* victims. However, we reveal WBB is underutilized (see Section III) and our evaluation exhibits that CacheRewinder's such policy results in negligible performance loss (see Section V-C).

V. Evaluation

A. Experimental Setup

We implement CacheRewinder on the cycle-accurate Gem5 simulator [14]. We configure the processor core and the memory systems as listed in Table I, which is based on Intel Haswell architecture [15]. To evaluate CacheRewinder we run individual benchmarks in SPEC2006 using Gem5's system-call emulation mode. We simulate 50 million instructions after fast-forwarding one billion instructions.

TABLE I: Processor configuration

Parameter	Value
Core	x86 ISA, out-of-order, no SMT, 64 IQ entries, 192 ROB entries, 32 LQ entries, 32 SQ entries, 32 STB entries
Branch Predictor	L-TAGE
L1-I cache	32 KB, 64B line, 8-way
L1-D cache	32 KB, 64B line, 8-way, 8 MSHR entries
L2 cache	256 KB, 64B line, 8-way
Write-back buffer	8 entries
Coherence protocol	MESI

B. Proof-of-Concept Defense

We test the defense by CacheRewinder using Spectre Prime+Probe proof-of-concept (POC) code [5]. We modify the original POC code to work on the L1 data cache of the Gem5 OoO processor model. Such an attack first fills cache sets with the attacker's data, then the attacker makes the load indexed by the *secret* data executed speculatively. This load

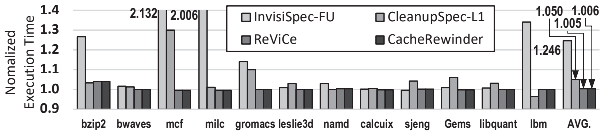

Fig. 10: Execution time normalized to the baseline

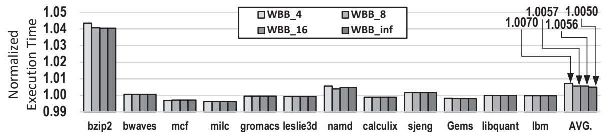

Fig. 11: Normalized execution time w.r.t WBB size

is squashed by the OoO core model when the speculation fails, however, one of the primed cache lines is evicted from the data cache by the speculative load. Then attacker probes the primed cache sets to detect the evicted cache line that exhibits longer access time. Figure 9 shows the access latency for the primed data after the Spectre Prime+Probe attack. The black solid line represents the access latency probed in the unprotected core model. One cache line is evicted by the speculative load indexed by the secret data (0x48) thus the evicted data exhibits a longer access time. Meanwhile, when CacheRewinder is employed, the access time for all primed data does not change since the cache state updated by the speculative load is revoked.

C. Performance Evaluation

We evaluate the performance change by CacheRewinder in Figure 10. We also compare the performance of CacheRewinder with InvisiSpec, CleanupSpec, and ReViCe [9]–[11]. InvisiSpec-FU represents InvisiSpec working under the futuristic conditions presented in [9]. CleanupSpec-L1 applies the CleanupSpec model for L1 data cache without the L2 invalidation mechanism. The execution time by each protection approach is normalized to the execution time of the unprotected baseline configured as listed in Table I. The performance evaluation results exhibit that the performance overhead by CacheRewinder is negligible, which is only 0.57% on average. The maximum performance degradation by CacheRewinder is 4.1% for bzip2. For some applications such as milc and calculix, the performance is improved when CacheRewinder is applied. It is because cache pollution by speculative loads is slightly alleviated by CacheRewinder. CacheRewinder also outperforms InvisiSpec and CleanupSpec by 24.0% and 4.4% respectively. InvisiSpec's *Redo* mechanism is inefficient for the applications (e.g. milc, lbm, and bzip2) that exhibit high branch misprediction rates. CleanupSpec's main source of performance loss is long restoring time since victims by speculative loads are provided by the slow L2 cache. On the other hand, CacheRewinder can minimize the restore time by exploiting the unused WBB entries as restore buffer spaces. ReViCe's performance overhead is slightly lower (0.50%) compared to CacheRewinder since ReViCe does not block the execution of speculation loads even if the restore buffer is full. However, the performance gap is extremely small.

We also evaluate the performance changes by the size of WBB as shown in Figure 11. As mentioned in Section IV-C

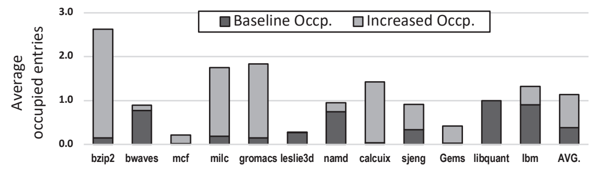

Fig. 12: Write-back buffer occupancy by CacheRewinder

CacheRewinder stalls the execution of speculative loads if all entries of WBB are occupied. This policy may incur additional pipeline stalls during speculative executions, which can degrade the performance. In Figure 11 the number after *WBB_* represents the number of WBB entries and *inf* means the infinite WBB size. Our evaluation shows the performance change with respect to WBB size is negligible.

D. Utilization of Write-Back Buffer

With CacheRewinder the utilization of WBB increases since the WBB entries are additionally occupied by *clean* victims evicted by speculation loads. Figure 12 exhibits the occupancy of WBB increased by CacheRewinder. We measure the occupancy of WBB as described in Section III. The original occupancy by *dirty* victims is represented as black bar graphs. The light grey bars represent the average number of the WBB entries occupied by *clean* victims during speculative executions. The evaluation results exhibit the average occupancy of WBB increases by 0.75 when CacheRewinder is applied.

E. Storage Overhead

The additional hardware cost for CacheRewinder is minimal since CacheRewinder exploits the existing WBB entries as a buffer space for holding victim blocks evicted by speculative loads. CacheRewinder includes STB to generate the commit control signal quickly. We configure the number of STB entries as 32 that is half of the IQ entry count. Cache tags, MSHR entries, and WBB entries are augmented to hold additional information as described in Section IV-C. In total CacheRewinder requires additional 191 bytes of storage.

TABLE II: Storage overhead

Solutions	Core	L1-D Cache	Total
InvisiSpec	28 B	2072 B	2100 B
CleanupSpec	224 B	56 B	280 B
ReViCe	24 B	2500 B	2524 B
CacheRewinder	60 B	131 B	191 B

The storage overhead by CacheRewinder and other approaches are summarized in Table II. InvisiSpec requires the additional SpecBuffer to temporarily store the data requested by speculative loads. ReViCe also requires additional storage space to store victims. CleanupSpec uses the existing L2 space to hold victim blocks during speculative executions, however, the restore operation takes long due to the slow L2 cache.

VI. Conclusion

In this paper we present CacheRewinder. CacheRewinder revokes the cache updates by speculative loads thus the attackers cannot leak secret data through cache timing side-channels. CacheRewinder exploits the unused write-back buffer entries as the temporary storage for victims evicted by speculative loads. When speculation fails CacheRewinder relocates the evicted blocks held in the write-back buffer to the data cache, thus the cache state can be restored. CacheRewinder implements efficient restore/commit operations that can minimize performance loss. Our evaluation exhibits CacheRewinder can effectively defend against transient execution attacks with only 0.6% of performance overhead. CacheRewinder requires minimal storage cost and hardware modifications compared to the previous architectural defense solutions.

Acknowledgement

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation grants funded by the Korea government (MSIT) (No. 2019-0-00533, Research on CPU Vulnerability Detection and Validation / No. 2019-0-01343, Regional Strategic Industry Convergence Security Core Talent Training Business / IITP-2021-2020-0-01819, ICT Creative Consilience Program).

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.
[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.
[3] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, “A systematic evaluation of transient execution attacks and defenses,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.
[4] M. Sternberger, “Spectre-ng: An avalanche of attacks,” in *Proceedings of the 4th Wiesbaden Workshop on Advanced Microkernel Operating Systems (WAMOS’18)*, 2018, pp. 21–26.
[5] 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*. IEEE, 2015, pp. 605–622.
[6] 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)*, 2014, pp. 719–732.
[7] Intel, “Intel analysis of speculative execution side channels,” *URL https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf*, 2018.
[8] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, “Kaslr is dead: long live kaslr,” in *International Symposium on Engineering Secure Software and Systems*, 2017, pp. 161–176.
[9] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.
[10] G. Saileshwar and M. K. Qureshi, “Cleanupspec: An” undo” approach to safe speculation,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.
[11] S. Kim, F. Mahmud, J. Huang, P. Majumder, N. Christou, A. Muzahid, C.-C. Tsai, and E. J. Kim, “Revice: Reusing victim cache to prevent speculative cache leakage,” in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 96–107.
[12] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev, “An intel ixp1200-based network interface,” in *Proceedings of the workshop on novel uses of system area networks at HPCA (SAN-2 2003)*, 2003.
[13] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “Ridl: Rogue in-flight data load,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.
[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.
[15] A. Akram and L. Sawalha, “Validation of the gem5 simulator for x86 architectures,” in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 53–58.