Restore Buffer Overflow Attacks: Breaking Undo-Based Defense Schemes

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

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

***Abstract*—Transient execution attacks have been severe security threats since such attacks exploit architectural vulnerabilities in out-of-order processors. Researchers proposed several architectural solutions to defend against transient execution attacks. By restoring the victim blocks stored temporarily in a restore buffer, the undo-based approaches can revoke the cache state changed by speculative loads. Thus it is known that the undo-based defense mechanisms can protect processors from transient execution attacks.**

In this paper, we reveal the undo-based protection scheme is still vulnerable to the elaborated Prime+Probe type attacks. Under the undo-based protection, the victim blocks by the speculative loads are stored in the restore buffer that has limited resources. Thus if the restore buffer is full, part of the victim blocks in the restore buffer can be evicted from the restore buffer. Then the cache state cannot be restored since the processor cannot find the victim blocks required for restoring the cache state. We design a restore buffer overflow attack that can leak secret data even if the processor is protected under the undo-based scheme. We evaluate the attack mechanism using the architectural simulator. Our evaluation exhibits that the attack can leak part of secret data successfully.

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

I. Introduction

Since the first disclosure of Spectre and Meltdown, the transient execution attacks have been severe security threats to modern computer systems because such attacks exploit the architectural vulnerabilities in out-of-order (OoO) processors [1], [2]. Namely, the transient execution attacks exploit speculative execution, which is an essential architectural solution to improve the performance of processors, to change cache states by accessing secret data. Then the attacks use cache timing side-channels to identify the cache states changed during the speculative executions. Several researchers proposed architectural solutions to protect the processor from transient execution attacks.

An undo-based protection scheme is one of the architectural solutions that can defend against the transient execution attacks [3], [4]. Under the undo-based scheme, the processor stores the victim blocks evicted from the data cache in the restore buffer during the speculative execution. When the speculation is resolved, the processor can restore the cache state to the original state before the speculative execution using the victims stored in the restore buffer. The researchers reveal that the proposed undo-based protection schemes can defend the processors against the popular transient execution attacks.

However, we reveal that the undo-based protection schemes are still vulnerable to the elaborated transient execution attacks. Since the restore buffer has limited resources, the victims in the restore buffer can be also discarded when the restore buffer is full. Thus if an attacker can make the restore buffer full of additional victims during the speculative execution, the data cache cannot be restored to the original state even though the processor is under the protection of the undo-based schemes. In this paper, we modify the Prime+Probe type transient execution attack to exhibit secret data can be leaked even if the undo-based architectural solution is applied to an OoO processor. We evaluate our attack mechanism using the popular architectural simulator that implements an OoO processor with the undo-based defense scheme.

II. Background

A. Transient Execution Attacks

Transient execution attacks such as Spectre and Meltdown have been serious security threats to modern OoO processors that employ speculative executions. To leak secrets such attacks exploit the architectural vulnerability in speculative executions. The attackers that employ the transient execution attacks can access secret information using the instructions executed speculatively (i.e. transient instructions). Note that the commit of the transient instructions is dependent on the results of prediction sources such as branch predictions and memory disambiguation. The modern OoO processors execute the transient executions speculatively even though the predictions are not resolved yet. The OoO processors cancel the execution of the transient instructions and roll back the architectural states if speculations fail. However, the cache states changed during the speculative executions are not revoked even if the speculation fails because it is harmless. The attack processes of the transient execution attacks change the cache state using the transient instructions to decode secret information by exploiting cache timing side-channels. Namely, the attackers change the cache state by accessing the secret data during the speculative execution. Then the attackers decode the secret information by comparing the cache states before/after the speculative execution.

To summarize, the attackers exploit the vulnerability in speculative executions to access secret data and the cache timing side-channel to decode the secret information from the cache state changes. Now we introduce the representative cache side-channel attacks exploited by the transient execution attacks.

1) Flush+Reload attacks: Flush+Reload type attacks detect cache blocks allocated during speculative executions that access secret data [5]. The attackers first flush the cache blocks in a target set from data cache using the cache flush instructions such as *clflush* of x86 ISA. The victim processes then access the secret data and one of the data blocks in the target set indexed by the secret data during the speculative execution. Note that the allocated cache blocks during the speculative execution remain in the data cache even though the speculation fails. Then the attackers access the cache blocks mapped to the data blocks in the target set. The data blocks indexed by the secret data are hit in the data cache thus this block can be accessed faster than other data blocks.

2) Prime+Probe attacks: To leak secret data Prime+Probe type attacks exploit the timing differences in cache accesses to identify the cache blocks evicted during the speculative executions. [6]. Unlike the Flush+Reload type attacks, the Prime+Probe type attacks do not require special cache flush instructions. The attackers first prime the data cache using an eviction set. Then victim processes access to the data cache using the array data indexed by the secret information. Note that the array data shares the same cache space already primed by the eviction set. Them one of the data blocks in the eviction set is evicted from the data cache. The attackers can identify the evicted cache lines indexed by the secret data by proving the cache blocks in the eviction set.

Algorithm 1 Prime+Probe Spectre Attack

```
1: prime the cache with eviction set data
2: if x is in bounds then
3:     secret = A[x];
4:     temp = array[secret]; // evicts an entry
5: end if
6: probe eviction sets
```

Algorithm 1 shows the algorithmic flow of the Prime+Probe type Spectre attack. The attacker prepares the attack by filling the cache lines with the data block in the eviction set. The victim statements (lines 3 and 4) are executed speculatively as the attacker has trained the conditional branch (*if* statement) is predicted as *true*. Note that the attacker tries to extend the attack window (i.e. cycles taken for the speculative execution) by flushing the required operand (i.e. *x*) of the branch. The cache access by the array data indexed by *secret* evicts one of the cache blocks in the eviction set, thus the attacker can identify the secret data.

We examine the above attack code on the Gem5 architecture simulator configured with the typical OoO processor settings as listed in Table I [7]. The attack processor successfully retrieves 80 out of 80 secret characters using the Prime+Probe type Spectre attack.

B. Undo-Based Defense Mechanism

To defend against the transient execution attacks, several researchers proposed undo-based architectural defense mechanisms [3], [4]. The undo-based defense approaches revoke the data cache state to the original state before the speculative execution if the speculation fails. For this purpose, the undo approach requires a buffer structure, called a *restore buffer*. The restore buffer is a temporary storage space that stores the victim cache blocks evicted by loads instructions executed speculatively. Note that in the conventional processor the victim blocks are simply discarded from the data cache if the victim is not dirty. The undo approaches revoke the cache state by transferring the victim blocks stored in the restore buffer to the original cache lines when the speculation loads are squashed due to the speculation miss. Thus the undo-based defenses are effective to defend against the transient execution attacks that exploit the cache timing side-channels. The undo mechanisms that use the restore buffer are essential for defending against the Prime+Probe type attacks since the attacks exploit the eviction data set as explained in the previous section.

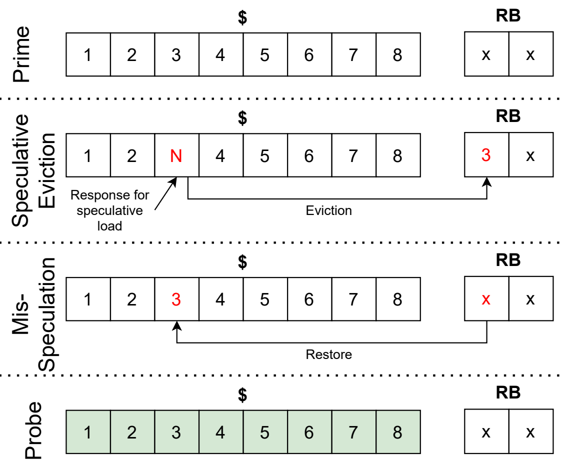

Fig. 1: Undo-based defense against Prime+Probe attacks

Figure 1 depicts how the undo-based approaches can neutralize the cache state changes by the Prime+Probe attacks. First, the attacker primes the cache with the eviction set data. When a speculative load allocates a cache block, one of the blocks in the eviction set is evicted from the data cache. Note that this victim block is temporarily stored in the restore buffer. If speculation fails the processor transfers the blocks stored in the restore buffer to the original lines of the data cache to restore the cache state. Now the attacker cannot identify the cache changes using the cache timing side-channel since all original blocks in the eviction set remain in the data cache.

The undo-based defense mechanisms can implement the restore buffer in various ways. CleanupSpec makes use of the existing L2 cache space as restore buffer space [3]. CleanupSpec can implement a large restore buffer using the L2 cache since the L2 cache size is usually larger than the L1 data cache. However frequent restore operations may degrade the performance of a processor due to long L2 access time. ReViCe adds buffer structure nearby L1 data cache to minimize the performance overhead from the restore operations [4].

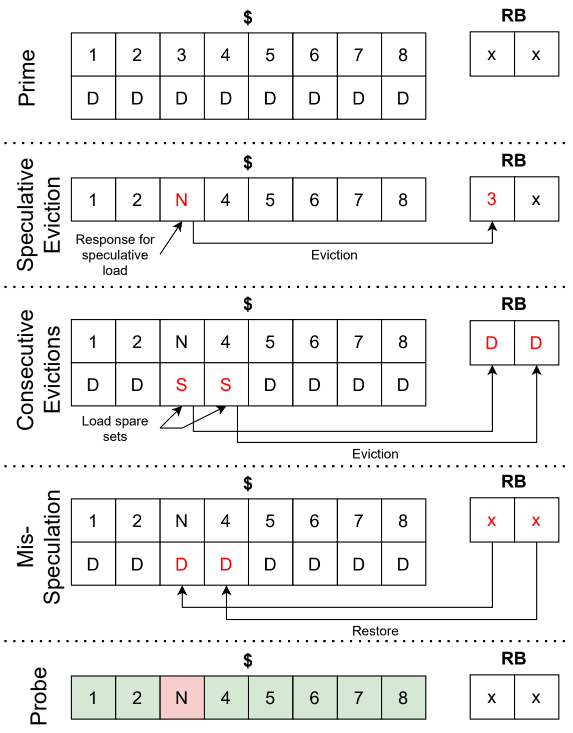

Fig. 2: Attack mechanism of the modified Prime+Probe attack

III. Restore Buffer Overflow Attack

As mentioned in the previous section, by using the restore buffer the undo-based mechanisms are effective to defend against the *eviction*-based attacks that probe the victim data blocks in the eviction set. However we *must* note that the restore buffer has limited resources (i.e. entries for the victim blocks) thus the restore buffer buffer cannot hold the victims evicted by the secret information if the restore buffer is full. It is because the victims in the restore buffer cannot be restored to the data cache if the victims are also evicted from the restore buffer. In this section we will explain why the restore buffer that has limited entries is also vulnerable to the eviction-based attacks. We will also describe the algorithmic flow of the modified Prime+Probe attack that can break the undo-based defense mechanisms.

Figure 2 depicts how attackers can break the undo-based mechanism we describe in the previous section. First, the attacker primes the data cache with *two* eviction sets: one is for probing the secret data (an original eviction set) and the other set is for making the victims evicted from the restore buffer (a dummy set). The attacker accesses the array of data indexed by the secret information during the speculative execution, thus one of the cache blocks (block #3 in the figure) is evicted from the data cache. This victim is allocated in the restore buffer by the undo-based protection. Within the attack window, the attacker continues allocating new cache blocks to evict the cache blocks in the dummy set from the data cache, thus the restore buffer is filled with these victim blocks. Note that the victim by the secret data (i.e. block #3) is evicted from the restore buffer due to the limited number of entries in the restore buffer. When the speculation fails the undo-based protection mechanism restores the data cache with the victims stored in the restore buffer, however, the victim block evicted by the secret data cannot be restored since the victim was evicted even in the restore buffer. Finally, the attacker probes the original eviction set in the data cache to identify the victim block indexed by the secret data. The attacker can observe a long cache response time for the victim since this victim block is not restored, thus the attacker can detect the secret data using the cache timing side-channel.

Algorithm 2 Modified Prime+Probe Spectre Attack

```
1: prime L1 cache with spare set data
2: prime L1 cache with dummy set data // fills L2 with spare
   set data
3: prime L1 cache with eviction set data
4: if x is in bounds then
5:     secret = A[x];
6:     temp = array[secret]; // evicts an entry
7:     consecutive accesses to spare set data // evicts dummy
       set data
8: end if
9: probe eviction sets
```

In Algorithm 2 we exhibit the algorithmic flow of the *restore buffer overflow* attack that can break the undo-based protection mechanism. We modify the conventional Prime+Probe Spectre attack to implement the restore buffer overflow attack. As described in Figure 2 the attacker requires to prime to eviction data sets: an original eviction set (line 3) and a dummy set (line 2). Since the attacker needs to complete the attack within the attack window while the speculative execution is performed, the attacker also primes a spare set in L2 to reduce cycles for fetching data blocks to evict dummy set blocks from the L1 data cache. For this purpose, the modified Prime+Probe pseudo code also includes spare set data that share the L1 data cache blocks with the dummy set data. To locate the spare set data in L2, the attacker first primes spare set data in the L1 data cache (line 1) and then allocates the dummy set data in L1 (line 2). Note that the spare data set exists in L2 since the data blocks of the spare set are evicted from L1. Like the conventional Prime+Probe attack, the restore buffer overflow attack also performs the victim codes speculatively during the attack window. After one of the data blocks in the original eviction set is evicted from L1, the attack also accesses data blocks in the spare data set thus the victim blocks in the dummy set fill the restore buffer. To make the victim block by the secret data evicted from the restore buffer, the attacker needs to fetch as many spare blocks as possible within the attack window. Finally, the attack process proves the eviction set data to identify the secret data similar to the conventional Prime+Probe attack.

IV. Evaluation

We evaluate the restore buffer overflow attack using the Gem5 architecture simulator with system-call emulation (SE) mode [7]. We configure the typical out-of-order processor as

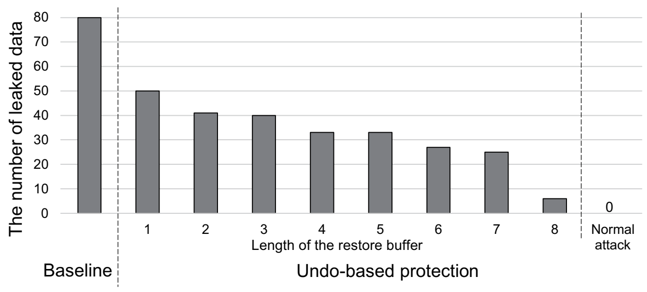

Fig. 3: The number of leaked data out of 80 secret characters w.r.t the number of the restore buffer entries

listed in Table I. We modify the OoO processor model of Gem5 to support the undo-based protection mechanism with a restore buffer nearby the L1 data cache.

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
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
Data prefetcher	Disabled
Protection	Undo-based protection with a restore buffer nearby L1 cache

We implement the attack code based on the Prime+Probe Spectre attack. We allocate 16 sets in the L1 data cache to prime the cache with the eviction set data. For the restore buffer overflow attack, we also allocate spare set data and then dummy set data in the L1 data cache as described in Algorithm 2. The attack code is compiled using gcc compiler for x86 ISA.

Figure 3 exhibits the attack results using our attack codes. The attack process tries to leak 80 secret characters. We study the success rate of the restore buffer overflow attack with respect to the number of entries in the restore buffer. On the x-axis of the graph *baseline* represents the typical OoO processor architecture without the undo-based protection approach. The numbers on the x-axis represent the number of entries in the restore buffer, which means the restore buffer overflow attack is tested on the OoO processor architecture protected by the undo-based defense scheme. *Normal attack* means the normal Prime+Probe Spectre attack described in Algorithm 1 is tested on the processor with the undo-based defense.

Our evaluation exhibits that the restore buffer overflow attack successfully leaks all secret characters on the unprotected baseline machine. Even if the processor is protected by the undo-based defense approach, the restore buffer overflow attack can leak part of secret data as shown in Figure 3. However, the attack cannot steal all secret characters if the processor is protected using the restore buffer. It is because the attacker requires sufficient time for allocating spare set blocks within the attack window in order to make the victim by the secret data evicted from the restore buffer. Sometimes the attack fails since the speculation is resolved before the attacker fetches enough blocks to the L1 cache. Our evaluation reveals that the success rate of the attack decreases as the number of entries in the restore buffer increases. It is because the attacker requires to allocate more blocks to fill the larger restore buffer. On the other hands the normal Prime+Probe Spectre attack (*normal attack* in the figure) fails to leak any secret data from the protected processor, thus we can say the undo-based protection mechanism effectively defends against the normal Prime+Probe type transient execution attacks.

V. Conclusion

Researchers have proposed several architectural defense approaches against transient execution attacks. It is known that the undo-based safe speculation scheme is effective to protect out-of-order processors from Prime+Probe-type attacks. In this paper, we prove that such protection schemes are still vulnerable to the elaborated Prime+Probe type attack, called a restore buffer overflow attack, since the restore buffer has a limited number of entries for the victim blocks. We evaluate the effectiveness of the restore buffer overflow attack using the architectural simulator that configures the out-of-order processor protected by the undo-based defense mechanism. We reveal that attackers can still leak part of secret information by exploiting the restore buffer overflow attack although the attack success rate decreases when the undo-based solution is applied.

Acknowledgement

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. 2019-0-00533, Research on CPU Vulnerability Detection and Validation / IITP-2021-2020-0-01819, ICT Creative Consilience Program) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1C1C1012172).

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] 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.

[4] 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.

[5] 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.

[6] C. Trippel, D. Lustig, and M. Martonosi, “Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols,” *arXiv preprint arXiv:1802.03802*, 2018.

[7] 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.