Isolating Speculative Data to Prevent Transient Execution Attacks

Kristin Barber, *Member, IEEE,* Anys Bacha, *Member, IEEE,* Li Zhou, *Member, IEEE,*
Yinqian Zhang, *Member, IEEE,* and Radu Teodorescu, *Member, IEEE*

Abstract—Hardware security has recently re-surfaced as a first-order concern to the confidentiality protections of computing systems. Meltdown and Spectre introduced a new class of exploits which leverage transient state as an attack surface and have revealed fundamental security vulnerabilities of speculative execution in high-performance processors. These attacks derive benefit from the fact that, during speculative execution, programs may execute instructions outside their legal control flows. This insight is then utilized for gaining access to restricted data and exfiltrating it by means of a covert channel.

This study presents a microarchitectural mitigation technique for shielding transient state from covert channels during speculative execution. Unlike prior work that has focused on closing individual covert channels used to leak sensitive information, this approach prevents the use of speculative data by downstream instructions until doing so is determined to be safe. This eliminates transient execution attacks at a cost of 18% average performance degradation.

Index Terms—hardware security, transient execution attacks, covert timing channels

✦

1 INTRODUCTION

Speculative execution has been used for decades to expose instruction level parallelism and increase performance. Unfortunately, the recent Meltdown and Spectre [1], [2] attacks have uncovered fundamental security vulnerabilities in how modern processors implement speculative execution. These attacks can generally be broken down into two phases.

First, speculation allows applications to execute instructions that are not part of their legal control flow. When a misprediction is detected the processor discards any erroneous instructions and continues execution. However, instructions outside the legal control flow can retrieve secret data that is otherwise inaccessible to the application (e.g. data belonging to a privileged process or data normally protected by the application's control flow, such as array boundary checks).

The second phase of these attacks involves using a covert channel to leak the secret obtained during the first phase. Covert timing channels consist of a trojan process (transmitter) which intentionally modulates the timing of a shared system resource to illegitimately reveal sensitive information to a spy process (receiver). The trojan and spy do not communicate explicitly, but covertly by observing the timing of certain events with respect to the shared resource. Although cache-based covert channels were used in the Spectre and Meltdown attacks, several other channels are available to an attacker.

Prior hardware-based mitigation solutions have focused on closing or removing the viability of specific covert channels, in most cases the cache [3], [4], [5], [6]. However, as some channels are closed, new ones are discovered [7], making channel-specific solutions less effective. Our approach takes the first steps towards a more general solution.

This study proposes a mechanism for shielding speculative data that may be obtained during the first phase of the attack from any possible covert channels. This is accomplished by blocking the use of speculative data by all dependent instructions until the source instruction is determined to be safe. This design is more general, less complex and with a similar performance impact as other existing hardware solutions.

To the best of our knowledge, this is the first work that has addressed the root cause of transient execution attacks: namely, the propagation of speculative data to downstream instructions that form the transmitter-side of a covert channel.

2 BACKGROUND AND RELATED WORK

A comprehensive summary of known transient execution attacks and defenses can be found in [8]. We briefly present Spectre-v1 (bounds-check-bypass) as an illustrating example. Listing 1 shows the code of the transmitter gadget from that attack. An attacker first trains the branch predictor to ensure a misprediction when $x > lenb$. Execution continues down the mispredicted path and a restricted value is accessed by the memory reference `b[x]`. Even though the secret data has been accessed, it will be cleared out of the architectural state of the system when the misprediction is identified. Before that happens, however, the secret can be leaked through a cache side channel.

This is accomplished by using a technique, such as Flush+Reload [9] or Prime+Probe [10], to set the cached contents of array `a` into a known state, where `a` is a shared resource acting as the covert channel. The secret accessed by the memory reference `b[x]` is then used to access a location in array `a` that is dependent on the restricted value, leaving a secret-dependent footprint in the cache. The attacker probes each cache line containing `a` and infers the secret value from the index exhibiting an anomaly in access time relative to the other indices. In Flush+Reload, the secret data would correspond to the array index with the lowest access latency.

```
1    if (x < lenb)
2        y = a[b[x] * 512];
```

Listing 1: C++ Spectre-v1 transmitter gadget.

- *K. Barber, L. Zhou, Y. Zhang and R. Teodorescu are with the Department of Computer Science and Engineering, The Ohio State University, Columbus, OH.*
  E-mail: see http://arch.cse.ohio-state.edu
- *A. Bacha is with University of Michigan, Dearborn, MI*

Multiple hardware defenses against transient execution attacks have recently been proposed, but each has limitations. Invisispec [3] proposes introducing shadow structures for caches to remove the side-effects of speculative execution from observation. DAWG [5] proposes adding protection domains to caches. Conditional Speculation [6] protects the data cache from leakage by blocking memory requests in the issue queue until they are known to not have been misspeculated. These approaches have only considered attack variants utilizing the cache as a covert channel, and have non-trivial complexity because they affect memory consistency and cache coherence. Context-sensitive fencing [11] provides a mechanism for automatically inserting fences into the instruction stream when malicious gadgets are dynamically detected with taint-tracking, requiring non-trivial tracking and recovery overheads.

Currently, there has been no solution that is covert-channel agnostic.

3 THREAT MODEL

In this work we address both Spectre and Meltdown-class transient execution attacks. We consider attackers targeting secret data residing in the memory hierarchy, and which may use any micro-architectural structure as a transmission medium–including caches, TLBs, FP units, execution ports, etc. Figure 1 conceptualizes the access path to secret data (shown in red), as well as the covert channels on a modern core microarchitecture (shown in blue).

Out of the scope of this work are attack scenarios in which secret data resides in a register that was preloaded through legal control flow (and non-excepting loads) and subsequently leaked through a side or covert channel. We also do not consider attacks targeting privileged registers such as Spectre Variant 3a (Rogue System Register Read) or registers with stale data that might not have been sanitized following a context switch such as the LazyFP attack. These attacks have been addressed through other mechanisms and represent a much smaller attack surface compared to attacks that can access arbitrary memory locations.

4 MITIGATION DESIGN

Key to our approach is the observation that, by definition, the leakage source (covert channel) has a data-dependence on the secret value, as can be seen in the example in Listing 1. The access into array a is dependent on the secret value referenced by b[x]. These two memory references form the transmitter-side of the covert channel and must both be executed speculatively, before the misspeculated instructions are squashed. Therefore, if we can delay the forwarding of data to dependent instructions until we can confirm the producing instruction is no longer speculative, we can inhibit leakage and restrict the formation of the transmitter. This effectively removes the ability to construct any covert channel that can leak this data.

To accomplish this goal we change the timing of when speculative data loaded into the processor's pipeline can be used by dependent instructions. Figure 1 shows a high-level view of how this integrates into a processor's pipeline design.

4.1 A Conservative Approach

The most conservative design delays the forwarding of data returned by a speculative Load instruction until it reaches the head of the reorder buffer (ROB) and is ready to commit. Figure 2 illustrates the ROB for an example with a LD instruction followed by two dependent instructions (ADD and SUB). Figure
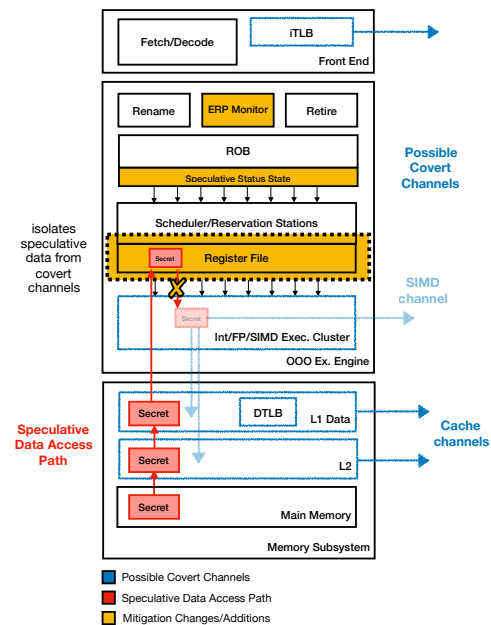

Fig. 1: Isolating speculative data from covert channels used to leak secret information.
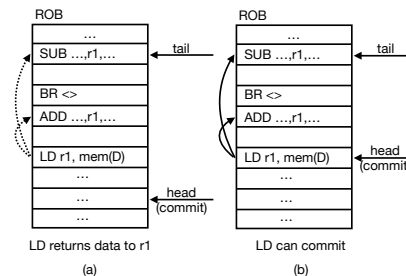

Fig. 2: Delays the forwarding of data loaded from memory location mem(D) into register r1 (a) until the LD instruction is ready to commit (b).

2(a) illustrates the cycle in which the LD receives the requested data from memory. Normally the value of mem(D) would be forwarded to all dependent instructions as well as stored in physical register r1.

When the LD data returns, the state of its ROB entry is checked. If it is not at the head of the ROB, the result of the LD will silently update the physical register corresponding to r1, but it will not broadcast the data on the result bus to dependent instructions. Since the physical register is assigned to r1, it will not be recycled until the LD instruction retires. Also, the LD will not be marked as complete, forcing any r1-dependent instructions added to the ROB after the LD returns to wait.

When the LD is ready to commit, its data is read from the physical register and broadcast to dependent instructions (Figure 2 (b)). At that point the LD is guaranteed to no longer be speculative. This ensures that no speculative data will be manipulated by any other instruction that could potentially leak information.

The wakeup/select logic has to be modified to consider all LD instructions as high (and variable) latency instructions. As a result, no LD-dependent instructions will be woken up when the LD is dispatched. Figure 3 shows the timing of the typical wakeup/select stages in the pipeline and the changes made to delay the wakeup of dependent instructions until the result of
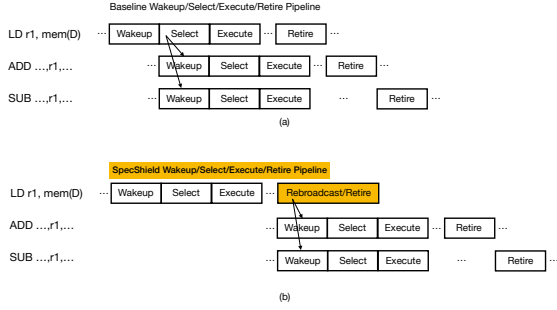
Fig. 3: Baseline design (a) and changes (b) to the timing of the wakeup/select pipeline to delay the broadcast of speculative `LD` results.

Fig. 4: The performance enhanced approach only delays the forwarding of data loaded from memory location `mem(D)` into register `r1` (a) until the `LD` instruction is ready to commit (b).

the `LD` is rebroadcast before retirement.

This approach is straightforward to implement in hardware requiring minimal changes to existing designs. However, delaying all instructions dependent on speculative loads by dozens or potentially hundreds of cycles leads to a significant performance impact, as will be shown in Section 5.

4.2 Early Resolution of Speculative Instructions

In order to address the performance impact of the previous approach, we develop a mechanism for early detection of non-misspeculated loads that allows their results to be forwarded earlier to dependent instructions. We define an Early Resolution Point (ERP) in the ROB as the eldest in-flight instruction in program order for which the following conditions are satisfied:

1) All older branch instructions (in program order) have been resolved and their actual direction is known.
2) All older loads and stores have had their address computed, and a TLB translation has been performed.
3) No branch misprediction or memory access exception has been raised by either of the above instruction types.

As a result of these conditions, all instructions between the head of the ROB and the ERP can be considered safe or *resolved* with respect to their ability to speculatively access data outside their legal control flow.

Figure 4 shows two snapshots of the ROB content and the position of the ERP. In Figure 4(a), the ERP is below the `BR<c1>` instruction. This means that all Branch and Load instructions between the ERP and the ROB head have been resolved and are neither misspeculated nor have they raised an exception. The `BR<c1>` instruction, on the other hand, has not been resolved, thus preventing the ERP from moving upwards.

This enhanced scheme allows the immediate forwarding of results for all the Load instructions that are older than the ERP and can therefore be considered safe. The result of the `LD r1, mem(A)` instruction can immediately be forwarded to its dependent instructions, which in this example is the `ADD` instruction. This allows Load results to be forwarded much earlier than in the previous approach, which restricts all Loads to wait until commit. All other Loads that are above the ERP (e.g `LD r2, mem(B)` in Figure 4) will continue to delay the forwarding of their results until they reach the ERP.

A new ER status bit associated with each ROB entry indicates whether the instruction has been *early resolved* or not. When a Load returns the ER bit is checked. If it is "1" the result is immediately broadcast on the result bus to all dependents. If it is not set, the destination register of the `LD` is silently updated. A *forwa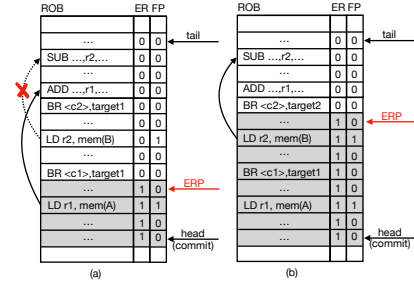rd pending* (FP) bit is set in that `LD`'s ROB entry. When the Load is early resolved the FP bit is checked. If it is set, the result of the Load will be rebroadcast to dependent instructions.

Even though this scheme is significantly less conservative compared to the previous one, it does not meaningfully relax its security properties. Only allowing data loaded by instructions within the Early Resolution window (highlighted in gray in Figure 4) to be forwarded to dependents prevents any speculative data from being accessed by any other instruction in the pipeline. While it is possible for the instructions now considered safe (below the ERP) to experience other types of exceptions or interrupts, prior work has shown these to be ineffective in facilitating transient execution attacks [8].

5 EVALUATION

5.1 Methodology

We used the gem5 cycle-level simulator in full-system mode, running a Linux operating system and modified out-of-order CPU model to implement the designs. Our simulations were run with an Ubuntu 14.04 disk image, Linux v4.18.7 kernel and x86 ISA. Table 1 shows the CPU and cache parameters used. Results presented are from simulation runs consisting of a 1B instruction warm-up period, followed by 500M instructions.

Workloads used in the evaluation were selected from the SPEC2006 benchmark suite to represent a diverse mix of integer and floating-point applications with both memory and compute-bound characteristics, using the *reference* input set.

CPU Architecture			
CPU Clock	2GHz	LSQ Entries	32
L1 ICache	32KB (4-way)	IQ Entries	64
L1 DCache	32KB (8-way)	BTB Entries	4096
L2 Cache	2MB (16-way)	dTLB Entries	64
Issue Width	8	iTLB Entries	64
ROB Entries	192	FP Registers	256
Branch Predictor	LTAGE	Int Registers	256

TABLE 1: Architectural configuration parameters.

5.2 Security Analysis

To analyze the effectiveness of both schemes in preventing leakage through covert channels we simulated proof-of-concept Spectre-v1 code in gem5, similar to the code shown in Listing 1. Flush+Reload was used to recover the secret. Figure 5 shows the empirical results averaged over 100 trials. In theory, the secret value will have the lowest access latency of any index in the array, enabling us to decode its value. The secret value is one byte wide, therefore the array must have 256 indices to represent every possible value. The top of Figure 5 shows the unsecure baseline. We can see that there is only one index
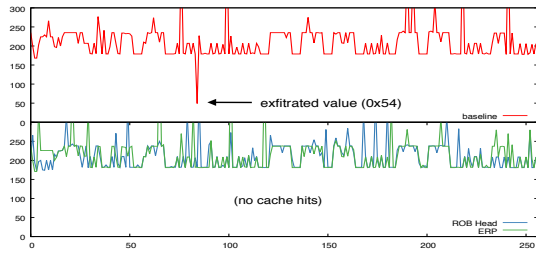
Fig. 5: Access latency for each index of array `a` in Listing 1, showing baseline and both schemes.
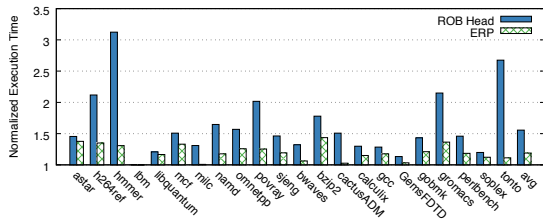

Fig. 6: Performance impact of both schemes.

Fig. 7: Average number of ROB entries from ERP to commit.

with an access latency less than 175 cycles; indeed, this index corresponds to the correct secret value extracted from the victim process (`0x54`).

However, both the conservative and enhanced schemes do not exhibit such an outlier, as shown in the bottom of Figure 5. This shows that our mitigation techniques are effective in shielding speculative data from the covert channel.

5.3 Performance Impact

The performance impact of both schemes, relative to the baseline system is shown in Figure 6. As expected, the more conservative mechanism results in a relatively large runtime increase, which averages at 55%. This is due to the fact that each Load instruction must wait until it reaches commit before it is permitted to forward its data, which could leave dependent instructions stalled for dozens or even hundreds of cycles.

The performance hit is worse for benchmarks that have relatively low miss rates such as *hmmer*, *tonto* or *gromacs*. For instance, *hmmer* has less than 10 misses per 1K instructions. Since for these applications most Loads are hits, delaying their resolution to commit has the highest performance impact. Applications with high miss rates (especially LLC misses) such as *lbm* or *mcf* exhibit a lower performance impact because Load misses already stall the pipeline considerably.

5.3.1 Impact of Early Resolution

In order to evaluate the opportunity created by the *early resolution* of instructions, we measure the distance in number of instructions from the ERP to the head of the ROB (commit point). This data is shown in Figure 7 as a per benchmark average. This metric allows us to quantify the opportunity for earlier forwarding of Load results.

The *hmmer* benchmark illustrates well how early resolution can translate into performance improvement. *hmmer* has the worst performance hit from the conservative design, at a 3.1× slowdown. However, *hmmer* also has a relatively large ERP to commit distance of 36 instructions, which can be interpreted to mean that, on average, 36 of the instructions in the ROB have reached the ERP and would be allowed to forward their results– potentially a significant performance improvement. This is
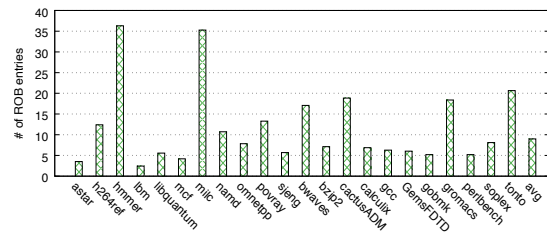
corroborated by the results in Figure 6, where we see that the runtime for *hmmer* with SpecShieldERP is only 30% longer than the baseline.

On the other hand, some benchmarks such as *mcf* and *astar* exhibit a short ERP-to-commit distance of only 4 and 3 instructions, respectively. As a result, they benefit less from ERP monitoring compared to other benchmarks, but still exhibit a 17% and 8% performance improvement, respectively.

Overall, the performance benefits of the ERP-based design are substantial, reducing the performance penalty to an average of 18% across all applications we examine, compared to 55% for the conservative design.

6 CONCLUSION

Transient execution attacks have revealed fundamental weaknesses in how modern processors handle speculative data. This study presents a first step towards isolating speculative data from all covert channels that could be used to leak secret information, at a cost of 18% average performance degradation.

REFERENCES

[1] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[2] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2019.

[3] M. Yan *et al.*, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[4] K. N. Khasawneh *et al.*, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation." *arXiv preprint arXiv:1806.05179*, 2018.

[5] V. Kiriansky *et al.*, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 2018, 2018, pp. 974–987.

[6] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 264–276.

[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[8] C. Canella *et al.*, "A systematic evaluation of transient execution attacks and defenses." *arXiv preprint arXiv:1811.05441*, 2018.

[9] Y. Yarom and K. E. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," *usenix security symposium*, vol. 2013, pp. 719–732, 2014.

[10] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," *the cryptographers track at the rsa conference*, vol. 2005, pp. 1–20, 2006.

[11] T. Mohammadkazem *et al.*, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.