Transient State Signaling for Spectre/Meltdown Transient Cache Side-channel Prevention

Zelong Li[a] and Akhilesh Tyagi[b]
Iowa State University, Ames, Iowa, U.S.A.

Keywords: Branch Predictor, Side-channel, Speculative Execution, Timing Attacks, Cache Hierarchy.

Abstract: The discovery of Meltdown and Spectre attacks and their variants showed that speculative execution offers a major attack surface for micro-architectural side channel attacks. The secret data-dependent traces in the CPU's micro-architectural state are not cleansed which can be exploited by an adversary to reveal victim's secrets. In this paper, we propose a cache control scheme that cooperates with a novel load store queue(LSQ) unit to nullify the cache side-channel exploited by Meltdown and Spectre attacks and their variants. In our proposed cache scheme, a new saturating reference counter is added to each cache line to hold the number of accesses since its arrival from the higher level of the memory hierarchy. For every squashed (uncommitted) speculative transient load, a corresponding flush request packet is sent to the downstream memory hierarchy. This ensures that any cache line brought into the cache by a transient load is always evicted soon after the corresponding mis-speculation commit. A cache side-channel adversary can no longer detect the existence of a transiently loaded cache block. Our experiment on gem5 shows that by integrating the proposed design, Meltdown and Spectre variants that uses Flush+Reload attack to create the cache covert channel are completely closed.

1 INTRODUCTION

By taking advantage of virtual to physical address translation, operating systems provide each running process its own isolated virtual address space leading to memory isolation for user processes. Operating systems also need to prevent a process which has a lower privilege level from reading or writing a memory address that requires a higher privileged level. For example, the memory address space belonging to the kernel can only be accessed by a user process in a secure way, such as system call. Such mechanisms have proven their effectiveness on architecture level security and hence are widely adopted on billions of devices. However, recently, some micro-architectural level speculation attacks such as Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2019) have been shown to break down all security foundations provided by memory isolation.

In this paper, we propose and evaluate a defense against Meltdown and Spectre attacks by introducing a Load/Store Queue Controller (LSQC) cooperating with a novel cache line design. We focus on the micro-architectural cache state of the cache system which has been the most widely exploited side-channel component in a processor. Our proposed design can identify the cache lines that were brought into the cache by transient instructions. These transiently loaded cache lines are flushed when the corresponding transient load instructions are flushed in the pipeline. This maintains the performance benefit of transient execution with minimal modifications to the existing underlying cache memory hierarchy. We add a new meta data element to each cache line. This enables cache controller to decide whether a cache line should be evicted based on the reference count of each cache line. We present our approach in detail in Section 3.

In Section 4, we analyze the security feature together with the performance & complexity impact of our design. We demonstrate that the proposed solution can successfully defend against Meltdown, Spectre and many of their variants with low performance loss. The performance difference between the baseline processor design and our proposed design shows that the overall performance loss is negligible.

In summary, the main contributions of this work are:

[a] https://orcid.org/0000-0003-0006-0145
[b] https://orcid.org/0000-0001-9096-7116

Li, Z. and Tyagi, A.
Transient State Signaling for Spectre/Meltdown Transient Cache Side-channel Prevention.
DOI: 10.5220/0011307500003283
In *Proceedings of the 19th International Conference on Security and Cryptography (SECRYPT 2022)*, pages 655-660
ISBN: 978-989-758-590-6; ISSN: 2184-7711

- We present a novel cache design that can identify cache lines which are brought in by transient load instructions.
- A LSQ unit controller to complement the proposed cache design to keep track of cache-line non-transient reference count.
- We identify the limitations of our work and propose solutions to mitigate these limitations.
- We evaluate performance penalty caused by implementing our design in modern CPUs on Gem5 simulator. We show that the performance penalty is acceptable.

2 BACKGROUND

In this section, we introduce the mechanism and capabilities of cache timing side-channel attacks. We discuss the exploitation of cache as a side channel and then overview how the Meltdown and Spectre attacks are mounted.

2.1 Timing Side Channel

The significant timing difference between a cache hit and a cache miss has led to a practical measurement to exploit cache as a side-channel. Cache based side-channel attacks have been successfully demonstrated in a wide range of scenarios. It can be mounted on both inclusive and non-inclusive cache hierarchies. Several cache attack techniques have been proposed and implemented based on the timing difference, including Prime+Probe (Osvik et al., 2006) and Flush+Reload.

2.2 Meltdown

Meltdown (Lipp et al., 2018) attack exploits the out-of-order execution performance enhancing speculation that all accesses satisfy the needed access privilege level. When a user application that is running at a lower privilege level and tries to access a memory address that belongs to the kernel, it will not succeed at the architecture level due to the privilege level violation. However, the speculative memory access request to the target address will bring this data into the cache, even though this data will not be brought into the processor registers. Since cache is not an architecturally visible object, this does not violate the architecturally visible state. However, these changes in the cache micro-architecture state can be probed by an attacker with a side channel to dump the whole kernel address space.

2.3 Spectre

Spectre attack relies on a covert channel between two different processes. It breaks the isolation between the attacker and the victim process. The attacker first mistrains the branch predictor so that the branch prediction will predict the path containing the secret revealing load. Explicit instructions to delay the branch resolution are also inserted. The victim load instructions along the mispredicted path will load the victim's secret into cache lines. On branch resolution, the load or the entire mispredicted path will be squashed. The transiently prefetched data in the cache lines can be read by the attacker process to extract the confidential information through the cache side-channel.

3 THREAT MODEL

We assume that the attacker process is running at user privilege level. It is trying to leak secret information in the victim process or kernel through transient execution. We assume that the attacker process and the victim process are running on the same logical core on the same machine in a time-sharing manner and the system has no software-based defense for transient execution. It is possible for the attacker to arbitrarily control the history of the branch prediction unit and know the source code and the address layout of the victim process.

This paper aims at mitigating the Flush+Reload attack as it is the most widely used method in the existing attack variants to decode the secret in the cache state. We believe the threat model we discussed above represents a large class of Spectre and Meltdown attacks and similar variants such as Foreshadow (Van Bulck et al., 2018). Yet we acknowledge that other types of cache side channel attacks such as Prime+Probe and Evict+Reload need also be mitigated. We will discuss the possible expansion of our proposed work in Section 6 as future work.

4 PROTECTION AGAINST MELTDOWN AND SPECTRE

In this section, we discuss our proposed design in detail. The basic principle is to identify and evict the cache lines loaded into the cache by transient instructions. Thus, during the reload phase, the attacker will not be able to detect a cache hit and infer the secret information.

Figure 1: Cache line organization with reference count.

4.1 Cache Line Reference Count

Along with the payload data, extra bits are stored in each cache line as meta data to describe some specific properties of the corresponding cache line. In this paper, we propose to add one more dimension to each cache line's meta data vector. As shown in Figure 1, the reference count of each cache block is stored in the meta data together with other bits. It is used to keep track of the reference count since the last time this cache line was brought into this level of the cache. Upon a Last-Level-Cache (LLC) miss, the requested cache line will be fetched from the main memory to the CPU and populate all levels of the cache. Initially, when a cache line is brought into a particular level of cache, the pending requests that are waiting for this cache line will be serviced. For each read and write request that has been serviced by a cache line, the reference count will be increased by one. In our proposed design, when the data of a cache line arrives at a particular level of cache, it will first be copied into a temporary cache line buffer. All of the pending requests stored in the miss status holding register (MSHR) are also serviced. Upon completion of servicing the requests to this cache line, it will be stored into the correct cache set.

To complement the proposed cache line design, we also propose a novel LSQ unit. Besides the conventional type of memory requests that an LSQ unit can send to the upstream memory hierarchy, a new type of request, *SpecFlush* is also available. It is designed to help the cache decrement the reference count of a target cache line on the corresponding load instruction squashing. In the proposed LSQ unit design, when a load instruction is later found out to be transient, and needs to be squashed, the LSQ unit will send out *SpecFlush* requests to reduce the reference count by 1. This negates the +1 reference count brought about by the speculative, transient load into the cache line. In summary, for each squashed load instruction that has sent a load request to the cache, a corresponding *SpecFlush* request are sent during the squashing process. In this paper, we present our proposed design on a two level cache system where L2 is the LLC to simplify the discussion. Following the same principle, the proposed work can be easily extended to work on a multi level cache system. Upon receiving a read request from L1 cache, the L2 will perform the same operations with respect to the reference count as L1, where its down stream memory component is the main memory. L2 cache lines also maintain a reference count in a manner similar to L1.

For each instruction that reads a memory address, a load queue entry is placed in the load queue to keep track of the status of a memory read request. The LSQ unit will also send out memory read request packets to the down stream memory hierarchy to request the data. The packet will first arrive at the L1 cache. Two different scenarios could take place upon receiving the packet. The L1 cache may have a copy of the requested data which, in turn, results in a cache hit. In this case, L1 cache can simply satisfy this request by sending a response to the CPU. It can simultaneously increase the reference count of the cache line that was just used to service the request. On the other hand, a L1 cache miss could happen. In this scenario, two sub-cases need to be treated differently. In the first case, a prior instruction that requests the same cache line data may have just sent a request packet to the L1 cache, yet the data has not came back from the down stream memory hierarchy. In this case, the L1 cache must have allocated a MSHR entry to hold the information about the ongoing memory read request to the same cache line data. The MSHR can simply append the current request packet at the back of the packet list while waiting for the data. In the second subcase, upon encountering a cache miss when receiving a read request packet, there may not be an MSHR entry for the requested address which means this packet is the first packet that is requesting this address from L1 cache. An MSHR entry needs to be allocated for a corresponding available cache line. This packet will be the first element in the packet list of this MSHR entry. In order to retrieve the data from the memory, a down stream packet also needs to be constructed based on the received packet to be sent out from L1 to L2 cache. In both of the sub-cases when cache miss happens, upon receiving the requested data from the down stream memory hierarchy, a temporary cache block is used to store the data. All of the pending requests including read, write, SpecFlush, etc. to the requested cache line are serviced. By executing each request in the pending request list, the reference count of temporary cache block will be increased by one when servicing a read/write request or decreased by one when encountering a SpecFlush request. If, after servicing all of the pending requests, the reference count is larger than 0 or the block is dirty, the cache line stays valid. This indicates that at least one load access instruction for this cache line is not squashed or a write request has accessed this cache line. This means that this cache line exists due to a non-transient program behavior and hence should continue to be part of the cache micro-architecture state.

However, if the reference count of the temporary cache block is 0 and it is not in dirty state, implying that there are a equal number of loads and *SpecFlush* requests, this cache line should be invalidated. A reference count of 0 indicates that the cache line existence reasons are exclusively transient. It should also send a SpecFlush request to the cache down stream to indicate that these caches may also need to evict the cache block. In summary, after servicing all of the pending requests at the L1 cache, if the reference count of the requested cache line becomes 0, the temporary cache line should be evicted. Additionally, a SpecFlush request needs to be sent to the L2 cache. Upon receiving the SpecFlush in L2 cache, it will also decrease the reference count of the transiently loaded cache line to 0. This forces the L2 cache to also evict this cache line even if it is not a temporary cache line. This in turn will send a SpecFlush request to the next level of the cache if it exists. The SpecFlush request will percolate through all the levels of the cache and flush the cache line in every level of the cache. In case, the transient load instruction is requesting data that hits in L2 but misses in L1 cache, it increases the reference count of the requested cache line in L2 from n to $n+1$ where $n > 0$. A copy of the cache line will be sent to L1. Once L1 services all of the pending requests using the requested data and invalidates this cache line based on reference count 0, it sends a SpecFlush request to L2. This decreases the reference count of the target cache line in L2 back to n.

For cache lines that are brought into the cache by retired instructions, the reference count must be larger than 0. There is a chance for these cache lines may be accessed by a transient load instruction as well and temporarily increases their reference count. The reference count will be restored to the original value when transient load instructions are squashed resulting in SpecFlush request being sent to the cache. Preserving such cache lines in the cache results in a performance benefit due to locality.

While each non-transient memory access will always increase the reference count by one, it is not necessary for each cache line to maintain an exact, large reference count. A saturating 2-bit counter associated with each cache line may suffice for reference count. It is also much more resource efficient to maintain in hardware. For a cache line *A* with a reference count of three, which is the maximum saturating value, a sequence of five read instructions will keep the reference count at three, instead of increasing it to eight. If these five instructions are finally determined to be transient and are squashed in the pipeline, five SpecFlush request packets will be sent to the cache, which effectively reduces the reference count of cache line *A* from three to zero. This will evict the cache line *A* leading to a performance loss. The program behavior however is not altered. In order to reduce the probability of a false eviction, more reference count bits are preferred. In our testing and evaluation setup on the Gem5 simulator, we observe that a maximum of seven speculative load instruction requests for the same address can be issued to the cache. This ensures that a 4 bits wide reference count field will suffice to prevent false evictions to maximize the program performance.

Unlike InvisiSpec (Yan et al., 2018) and Cleanupspec (Saileshwar and Qureshi, 2019) that only support software prefetching due to the lack of knowledge about the prefetched cache lines, both software and hardware prefetching are supported by our design. At the moment when a perfected cache line arrives the cache, if the prefetched cache line is not accessed by any pending load or store request, it will have a reference count of 0. Therefore, prefetched cache line will be evicted after receiving the response from the downstream memory hierarchy.

5 EVALUATION

5.1 Evaluation Setup

To evaluate the proposed design, we use Gem5 to model an x86 system that supports out-of-order and speculative execution. We integrate our proposed LSQ unit design and cache scheme in the Gem5 simulator and simulate a subset of SPEC CPU2006 benchmarks in syscall emulation mode on a single core system. To facilitate the simulation process yet capturing the performance impact of our proposed design with high accuracy, we used Pinplay(Patil et al., 2010) with SimPoint(Hamerly et al., 2005) to discover the representative simulation locations for SPEC benchmarks. For each benchmark program we use the *reference* input size and generate up to ten simulation locations. For each simulation location, we use 10 million instructions to warm up the cache and then execute 1 million instructions to observe the performance. The CPU and cache configurations of our evaluation setup are shown in Table 1.

5.2 Performance Overhead

Figure 2 breaks down the performance overhead of implementing our proposed design compared to the baseline CPU. It illustrates normalized IPC values for the tested SPEC benchmarks. Here, we see that mcf is hurt the most among the tested benchmarks

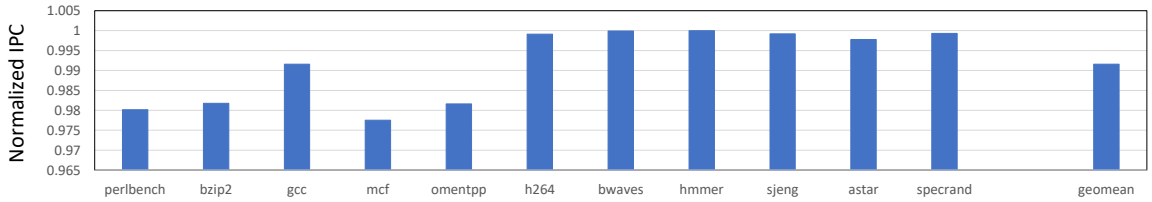

Figure 2: Relative Performance Compared to Baseline CPU.

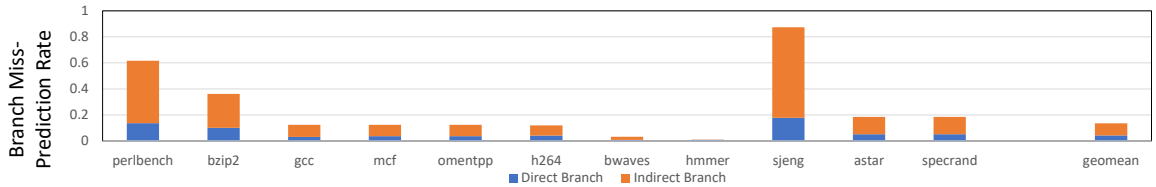

Figure 3: Direct + Indirect Branch Miss-Prediction Rate.

Table 1: Configuration of the Simulated CPU.

Parameter	Configuration
Core	1-Core, 8-issue, Out-of-order, 2.0GHz
Pipleline	224 ROB entries, 32 Load Queue entries, 32 Store Queue entries, 256 Int and 256 FP physical registers
Branch Predictor	LTAGE, 4096 BTB, 16 RAS
L1-I Cache	32KB, 64B line, 2-way, 5 cycle hit, 4 MSHRs
L1-D Cache	32KB, 64B line, 2-way, 5 cycle hit, 4 MSHRs
L2 Cache	8MB, 64B line, 8-way, 20 cycle hit, 20 MSHRs
Memory	4GB, DDR4-2400

as its IPC is decreased by 2.7%. While some other benchmarks such as bwaves and hmmer slow down by less than 0.1%. The geometric mean of the performance overhead over all the benchmarks in Figure 2 is 0.8%. We believe the performance overhead is significantly affected by the number of branch mis-predictions and the number of transiently loaded cache lines. For benchmarks that have lower mispredictions or cache misses counts during the transient window, intuitively, the performance degradation should be minimal.

For the baseline processor, when a load instruction cannot commit in the ROB, the load request may have already been sent to the memory system and the cache line will still be brought into the cache. This is despite the fact that the instruction is squashed in the pipeline. Due to data locality, this speculatively loaded cache line is very likely to be accessed in the near future. A branch is likely to exhibit both its taken and not-taken paths over a short time window. Hence a current speculative path is likely to be a committed path in the near future. This results in a cache hit on these speculatively loaded data to improve the performance. However, in the proposed design, such cache lines must be flushed to eliminate the attack surface. Intuitively, this likely increases the cache miss count compared to the baseline processor which in turn causes performance degradation.

To dive deeper in the causes of performance degradation, Figure 3 shows the branch misprediction rate in the tested SPEC 2006 benchmarks on the baseline CPU. Due to the extremely low misprediction rate, the IPC performance of benchmarks such as h264, bwaves and hmmer are barely affected. Though sjeng has a relatively high branch miss-prediction rate, its performance is barely affected due to its low cache miss rate. For a lower overhead, it is beneficial to have a branch prediction unit with a high accuracy. For instance, a branch predictor with 50% accuracy will request half of the speculative loaded cache lines to be flushed while a branch predictor with 95% accuracy will only flush 5% of the speculative loaded cache lines.

5.3 Hardware Overhead

The extra bits for storing the reference count of each cache line brings hardware overheads to the CPU cache. We use CACTI (Muralimanohar et al., 2009) v6.5 to calculate the area and power overhead that are needed to implement the proposed design in the L2 cache. The number of extra bits added to the cache tag vector when simulating in CACTI is the same as the evaluation setup, which is 4 bits. At 32nm, CACTI estimates that the area overhead for the L2 cache is $0.18mm^2$ with dynamic access energy increased by 8.9% when accessing the tag. Compared with the L2 cache area size, which is $4.75mm^2$, the area overhead is negligible.

6 LIMITATION AND FUTURE DIRECTIONS

We recognize that the major limitation of the current work is that it is only feasible when defending against transient execution attacks that use side-channels such as Flush+Reload and Flush+Flush. Attacks, such as Prime+Probe, can still succeed. The first step we take is to use a temporary cache block to store the latest loaded cache line instead of putting the data into the cache directly before servicing all of the pending requests. Very often the temporary cache line will be evicted after servicing, thus helping to decrease the success rate and bandwidth of the Prime+Probe attack.

Our discussions of the design details are based on a single thread model. The proposed design can be expanded to work with a multi-core system. In a multi-core system, a cache line loaded during the transient execution window by one thread can be accessed by another thread before it is evicted from the cache. This forms a temporary side channel in the shared L3 cache. We can resolve this problem by taking advantage of the saturating counter associated with each cache line. For a system with n cores, if a transient cache line is loaded into the L3 cache and all of the cores are accessing this cache line transiently during the same transient window, the reference count of this transiently loaded cache line in L3 can reach up to n. Thus, a cache line with a reference count greater than n has established its bona fides as a non-transient data. The L3 cache can respond to requests to this line in the usual manner. However, if the reference count of a cache line is less than n, the L3 cache can delay its response to simulate a cache miss.

7 RELATED WORK

Since the discovery of Spectre and Meltdown, a wide variety of defenses were proposed by both CPU vendors and other researchers. Most of these proposed defenses involve either mitigating transient execution or mitigating the side channel. Adding a fence instruction after each branch instruction is the most intuitive approach that can defend against transient execution. However, this approach abandons the benefits of speculative execution causing an average of 88% performance loss (Yan et al., 2018).

To limit the covert channels, hardware mitigation approaches are also proposed by researchers. SafeSpec (Khasawneh et al., 2019) propose to introduce extra shadow structures for caches and the TLBs to store speculative states temporarily. If an instruction is squashed in the pipeline, the corresponding entry in the shadow structures will be discarded to leave no traces in the micro-architecture. However, the extra shadow structure requires a larger area to implement compared to our proposed design.

REFERENCES

Hamerly, G., Perelman, E., Lau, J., and Calder, B. (2005). Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28.

Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2019). Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2019). Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown. *arXiv preprint arXiv:1801.01207*.

Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. (2009). Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28.

Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer.

Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. (2010). Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11.

Saileshwar, G. and Qureshi, M. K. (2019). Cleanupspec: An" undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 73–86.

Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008.

Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., and Torrellas, J. (2018). Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE.