SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation

Khaled N. Khasawneh
kkhas001@ucr.edu
University of California, Riverside

Esmaeil Mohammadian Koruyeh
emoha004@ucr.edu
University of California, Riverside

Chengyu Song
csong@cs.ucr.edu
University of California, Riverside

Dmitry Evtyushkin
devtyushkin@wm.edu
College of William and Mary

Dmitry Ponomarev
dima@cs.binghamton.edu
Binghamton University

Nael Abu-Ghazaleh
naelag@ucr.edu
University of California, Riverside

ABSTRACT

Speculative attacks, such as Spectre and Meltdown, target speculative execution to access privileged data and leak it through a side-channel. In this paper, we introduce (*SafeSpec*), a new model for supporting speculation in a way that is immune to the side-channel leakage by storing side effects of speculative instructions in separate structures until they commit. Additionally, we address the possibility of a covert channel from speculative instructions to committed instructions before these instructions are committed. We develop a cycle accurate model of modified design of an x86-64 processor and show that the performance impact is negligible.

ACM Reference Format:
Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. *SafeSpec*: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317903

1 INTRODUCTION

Speculative execution is a standard microarchitectural technique used in virtually all modern CPUs to improve performance. The recent Meltdown and Spectre attacks [8, 15–19, 28, 29] (we call this class of attacks *speculation attacks*) have shown that speculation can be exploited to expose information that is otherwise inaccessible. Several attack variations have been demonstrated, including arbitrary exposure of the full memory of other processes, OS kernel, hypervisor, and even SGX enclaves [3, 28] to an unprivileged attacker, making this a dangerous open attack vector on modern systems. We describe these attacks and present our threat model in Section 2.

Although a number of defenses and software patches have been proposed to mitigate Spectre and Meltdown [7, 27], they often address only one aspect of the attack, leaving attackers with other possible variations that are still available. In addition, these patches often lead to high overheads: 10-30% reported on average, but often much higher. For example, Netflix reported 800% slowdown with the Meltdown patches on their systems [6, 26]. Most of the solutions target a subset of the threat models and make assumptions that can be broken by future architectures.

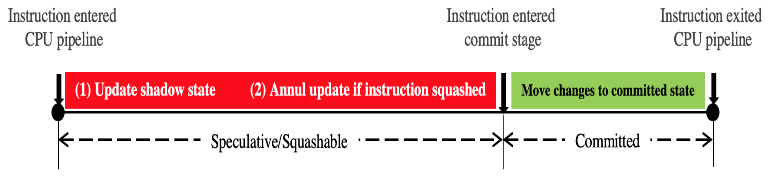
In this paper, we explore whether speculation can be made leakage free in a principled way, enabling CPUs to retain the performance advantages of speculation while removing the security vulnerabilities that speculation exposes. To this end, we introduce *SafeSpec*, a design principle where speculative state is stored in temporary structures that are not accessible by committed instructions. As instructions transition from being speculative to commitable, any speculative state is moved to the permanent structures. On the other hand, if a speculative instruction is squashed, the speculative side effects are canceled in place leaving no measurable side effects in the permanent structures and closing the vulnerability exploited by speculation attacks. We consider two variants that differ in when an instruction is considered safe to commit. *SafeSpec* makes no assumptions on the branch predictor behavior or on speculative execution behavior; for example, it does not prevent the attackers from mis-training or even polluting the branch predictor, nor does it prevent them from speculatively reading privileged data. Rather, SafeSpec interferes with the attacker's ability to create a covert channel using speculative data accesses to communicate illegally-accessed data out. We describe *SafeSpec* in Section 3.

We demonstrate the *SafeSpec* principle by building a memory hierarchy (caches and TLBs) that are free from speculation-induced leakage. In particular, we expand the load-store queues to store a pointer to a temporary associative structure that holds speculatively loaded cache lines. We also introduce a similar structure to hold speculatively loaded translation lookaside buffer (TLB) entries. We describe the design and some of the complexity-performance tradeoffs in Section 4.

Additionally, we identify a transient type of leakage that occurs in the introduced speculative state (byproduct of *SafeSpec*) that we call *transient speculation attacks* (TSAs). We explore how to construct the shadow structures to mitigate TSAs in Section 5. Furthermore, Section 6 presents a performance, complexity and security analysis of *SafeSpec*. We also analyze the complexity of *SafeSpec* including the impact of all new structures, and demonstrate a reasonable increase in the area and power consumption. Finally, we show that *SafeSpec* stops proof-of-concept implementations of all variants of Meltdown and Spectre, as well as the new variants that we introduced.

In summary, the paper makes the following contributions:

- We introduce the *SafeSpec* model to protect speculation by isolating speculative state from committed state.
- We identify a new class of speculative attacks (Transient Speculation Attacks) that arises in *SafeSpec*. We mitigate such attacks by sizing the shadow structures to prevent contention.



Instruction entered CPU pipeline
Instruction entered commit stage
Instruction exited CPU pipeline
(1) Update shadow state
(2) Annul update if instruction squashed
Move changes to committed state
Speculative/Squashable
Committed

Figure 1: SafeSpec overview

- We evaluate *SafeSpec* for caches and TLBs from a security, performance and complexity perspective.

2 BACKGROUND AND THREAT MODEL

Speculation attacks such as Spectre, Meltdown, and their subsequent variants, exploit the fact that permissions are not checked while instructions (or subsets of instructions in the case of Meltdown) are being executed speculatively. Conventional wisdom was that this microarchitecture assumption, which allows aggressive and performance-beneficial speculation, was not dangerous since the effects are simply undone once misspeculation is discovered (or once an exception is raised in the case of Meltdown). The attacks showed that, the secret values that are speculatively read can be communicated through a side channel opening this dangerous and previously unknown class of vulnerabilities.

Spectre and Meltdown attacks differ only in how they trigger speculation. Meltdown attacks exploit speculation within a single instruction that will eventually fail due to permission checks, or processor faults. Before they fail, illegal accesses are executed speculatively and communicated through the side channel. In contrast, Spectre attacks manipulate the branch prediction structures to cause the speculative execution of code that will read the secret data and communicate it.

```
if (offset < array1_size)
    y = array2[array1[offset] * 64];
```

The code snippet above demonstrates Spectre variant 1 of the attack. In this code, the attacker mistrains the branch in the if statement to be always taken. To launch the attack, the code is executed with a large offset, that makes the access to array1 read into the kernel address space. This access is performed speculatively since the branch has been trained to be predicted taken. Then, resulting value is used to perform an access into array2. As we discussed above, accesses into the array2 leaving a footprint in the cache for array2 that can be detected by the attacker (using a standard side channel attack such as Flush and Reload [31].

Given the large number of variants that have been discovered, it is unlikely that simple defenses that target each variant individually would provide principled protection from this class of attacks. SafeSpec is general and applicable to different micro-architectural structures. However, as a demonstration, our prototype implementation only protects caches and TLBs to explore concretely the implications and complications that result from *SafeSpec*. Therefore, we further assume that other covert channels, including the ones through the branch predictor, memory bus and DRAM buffers are out-of-scope for the current paper, but will be addressed using similar principles by future work. Similarly, we only consider a system with a single core. Thus, speculation attacks against the cache coherence and memory consistency model states [25] are also left for future work.

3 *SafeSpec*: LEAKAGE-FREE SPECULATION

SafeSpec is a principled approach to secure processors against speculation attacks while retaining the ability to carry out speculative execution to benefit from its performance. The general principle (shown in Figure 1) addresses the problem at the root by introducing shadow state to separate state that is produced speculatively without affecting the primary structures of the processor (which we call committed state). For example, if a speculative load instruction causes a load of a cache line, instead of loading that cache line into the processor caches, we hold the line in a temporary structure. If the load instruction is later squashed, these effects are removed in place, leaving no changes to the cache from the misspeculated instructions, and closing the vulnerability. Alternatively, if the instruction commits, the cache line is moved from the temporary structure to the L1 cache. While *SafeSpec* is simple in principle, a number of questions relating to its security, complexity and performance have to be resolved.

When to move state from speculative to committed. There are two options available to decide when to move state from the shadow to the committed state. In the first variation, which we call *wait-for-branch* (WFB), we can assume an instruction to be no longer speculative when all the branches (more generally, all predictions) it is dependent on have been resolved. WFB stops all variants of spectre which depend on mistraining the branch predictor/return stack buffer; none of the mis-speculated instructions moves to the committed state. However, it does not prevent Meltdown which relies on speculation within a single instruction.The second variation *wait-for-commit* (WFC) waits until the instruction commits before moving its effects to the committed state, and therefore also prevents Meltdown.

Shadow state organization and size: If the shadow state structures are too small, then either speculative state is replaced (causing a loss of an update to the committed state if this data were to be committed later), or the instruction has to stall until there is room in the speculative structure before it issues. From a performance perspective, the organization and size of the shadow structure should be designed such that the structures can hold the speculative state generated by speculation as measured across typical workloads. However, we will show that security considerations introduce more stringent requirements on the speculative state.

Mitigating Transient Speculation Attacks: *SafeSpec* by construction prevents speculative values from affecting the state of committed structures, which is the pathway used to communicate data covertly in the published speculation attacks. However, it does not create isolation between instructions that are in the speculative state. This creates a possibility for a new variant of attacks which we call *transient speculation attacks (TSAs)*. In particular, since most instructions that commit start in the speculative state, there is a window of time where they can share the speculative state with misspeculated instructions before they are squashed. If we are not careful, it is possible to create a covert channel in this period to communicate the sensitive data from the mis-speculated branch to the branch that will be committed, allowing the data to be exfiltrated. The attack is illustrated in Figure 2 and we discuss how to mitigate TSA attacks in Section 5.

Filtering Delayed Side Effects: One of the issues with *SafeSpec* occurs when an instruction is squashed in the middle of its execution. If the instruction has already initiated a high latency operation such as a read from memory, we have to ensure that the response from memory can be discarded after it is received. We handle this situation by discarding values received if there is no matching transaction. However, it may also be desirable to filter these transactions

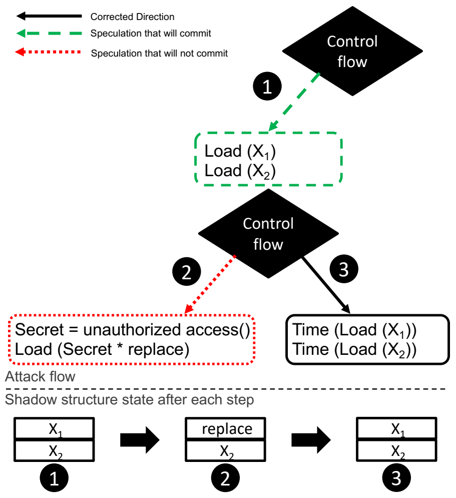


Figure 2: Transient speculation attack (TSA)

lower in the system, such that the committed transactions commit directly, and the squashed ones are cancelled in place. To control the size of this filter, we include a branch id with the transactions and track operations at the branch granularity. The filter can also be used to mark committed branches so that memory responses corresponding to them are committed directly.

4 *SafeSpec* FOR CACHES AND TLBS

To demonstrate the *SafeSpec* principle, we implemented it to protect CPU caches and TLBs from leakage during speculative execution. To provide full protection, all speculatively updated structures should follow the SafeSpec principle. We chose the CPU caches because they are easily exploitable targets for covert communication and the ones used in the Spectre/Meltdown attacks.

To protect from speculative covert channels that occur during memory accesses, and following the *SafeSpec* principles, we need to add shadow state to protect the following structures.

Data caches: this is the covert channel used in all three Meltdown/Spectre variants. We add a shadow structure to hold the cache lines that have been fetched speculatively. The structure is associatively-filled lookup table (filled associatively, but accessed as a lookup-table). In the Load/Store queue, we point speculative loads that have received their data to a corresponding entry in this table. Speculative instructions in the *same execution branch as the load that fetched a shadow cache line* that accesses this cache line can use the value from the shadow structure. If an instruction commits (depending on WFB or WFC), the cache line is moved from the shadow structure to the caches. If the instruction is squashed, the shadow structure entry is marked as available. Thus, not even the cache replacement algorithm state is affected by the speculative data that does not commit.

Instruction caches: we built variants of Meltdown/spectre using the instruction cache that replaces data dependent array access with dependent branches to a location in an array to disclose the data through the i-cache, illustrating that it must be protected as well [12]. To develop this attack variant, we had to overcome branch predictor behavior: data dependent branches were using the branch predictor, rather than the secret data. Thus, we had to initialize the branch target buffer (BTB) to a third location, and then introduce

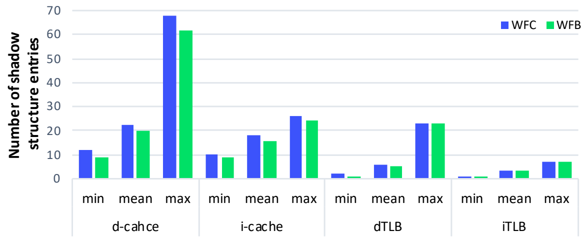


Figure 3: Shadow structure size that fits 99.99% of accesses

sufficient delay in the pipeline for the data dependent branch such that it has time to register the data dependent location in the i-cache. **TLBs**: we also conjectured that the TLBs may be used as a covert channel vector. Given recent attacks such as Foreshadow [28] and TLBleed [5] which directly target the page translation behavior for speculation attacks, its critical to protect these structures.

To implement *SafeSpec* for the data cache, we add an associatively-filled lookup table to hold speculatively read cache lines. It is important to note that memory consistency models, such as Total Store Order (TSO) semantics of the x86-64, often ensure that store side-effects appear in order; in other words, the cache is not updated until the store commits, making stores robust to speculation attacks. We augment the load store queue with a pointer to the shadow cache line for load operations that are speculative. Any instruction dependent on the speculative load reads the cache line from the shadow structure. Once the load instruction commits, the shadow cache line is written to the caches according to the inclusion policy of the caches (in our case, since the caches are inclusive, it is written to all levels of the cache) and freed in the shadow structure. If the load is squashed, the value is freed in the shadow structure. For the i-cache and the TLBs, we create similar shadow structures, and augment the reorder buffer (ROB) with pointers to the shadow state entries if the instruction is speculative and the cache line (or TLB entry) were fetched speculatively.

From a performance perspective, the structures should be sized such that they accommodate the speculative state needed by representative workloads. If the shadow structures are full, we could either drop some of the shadow state (leading to loss of updates to the committed state with performance, rather than correctness implications), or block until there is space in the shadow state before issuing an instruction (also with performance implications). We will see later that the constraints introduced by security requirements to eliminate TSAs are more stringent than those required by performance. Figures 3 show the distribution of the size of the speculative state sampled over time for the SPEC 2017 benchmarks. The shadow d-cache for 3 of our benchmarks grows occasionally to almost the maximum possible size (bound by the size of the load-store queue). A shadow i-cache with about 25 cache lines is sufficient for all of the benchmarks. In addition, less than 10 entries are sufficient for speculative iTLB misses, but some benchmarks require more dTLB entries (up to 25). Given that the overhead of supporting WFC is small, we elect to support WFC to get the increased protection to cover Meltdown.

5 TRANSIENT SPECULATION ATTACKS

The *SafeSpec* principle prevents direct side-channel leakage from the speculative state to the committed state, closing all known speculation attacks. However, although the committed instructions

Table 1: Configuration of the Simulated architecture

Parameter	Configuration
CPU	6-way issue, 96 Issue Queue entries, out-of-order, no SMT, 72 Load Queue entries, 56 Store Queue entries, 224 ROB entries, 64 iTLB entries, 64 dTLB entries, commit up to 6 Micro-Ops/cycle
Private L1 i-/d-Cache	32 KB, 8-way, 64B line, 4 cycle hit
Shared L2 Cache	256 KB, 4-way, 64B line, 12 cycle hit
Shared L3 Cache	2 MB, 16-way, 64B line, 44 cycle hit

and the speculative instructions eventually reside in separate structures, creating the separation and closing the channel, eventually committed instructions can start out as speculative. During this window, the eventually committed instructions share the shadow state with any speculative instructions that will be squashed. If the shadow structures are not designed carefully, covert channels can be created during this transient window to communicate sensitive data (which can only be read by a mis-speculated path) to an instruction pathway that will be committed such that the leakage results are visible to the program. It is important to emphasize that these attacks (which we call Transient Speculation Attacks, or TSAs) are substantially more difficult than Spectre/Meltdown because there is only a limited window of speculation in which the malicious Trojan code must not only read sensitive data, but also create measurable contention to the spy before either of their predicate branches commits.

TSAs are possible only if the shadow structures are shared and sized such that they enable contention. Consider an example where we size the TLB shadow structures based on typical program behavior. Since programs do not have many pending TLB misses within a speculation window, it stands to reason to size these structures to be small. In the rare case when the shadow structures are full, we may handle this by either discarding updates or by blocking the issue of requests when there is no room in the shadow structure. Either of these behaviors provides potential for a covert channel. Consider that the Trojan fills the structures with TLB misses if it wants to communicate a 1. If updates are discarded, a spy can detect a communication if its TLB accesses are not committed (they were discarded). Alternatively, if we block TLB accesses when the structures are full, the spy can detect a communication of 1 if its TLB accesses are delayed causing a longer TLB miss time. The attack is illustrated in Figure 2.

To prevent TSAs through the shadow structures, we elect to provision them for the worst case scenario to make sure that transient contention cannot be created within a speculation window. This approach guarantees that no contention on the shadow structures is possible, at the cost of provisioning fairly large associative structures. We believe that with some more analysis, or with some detection defense that detects an attack when the shadow structures grow abnormally large, this worst case provisioning can be substantially relaxed without introducing leakage.

6 EVALUATION

We conduct experiments with MARSSx86 [21], which is a cycle-accurate full-system simulator of out-of-order x86 cores. We configured the CPU and cache models of MARSSx86 to simulate the Intel Skylake processor as shown in Table 1.

6.1 Performance Analysis

The first experiment measures the performance of *SafeSpec* compared to the baseline processor under conservative condition. In particular, we consider the shadow state access time to be equivalent to the access time of the L1 cache (4 cycles), when it is substantially smaller, and accessed as a lookup table. Figure 4, shows the IPC values for all SPEC2017 benchmarks. We see a small improvement in performance with a geometric mean of about 3%. We believe that this advantage results from a combination of effects including the larger effective cache size and avoiding polluting the cache with wrong path speculative state.

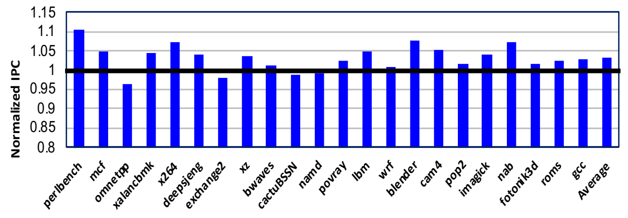


Figure 4: Relative performance to non-secure OoO execution

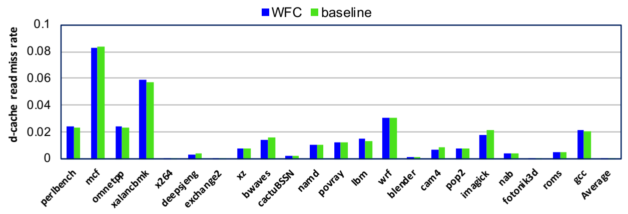


Figure 5: d-cache read miss rates including shadow d-cache

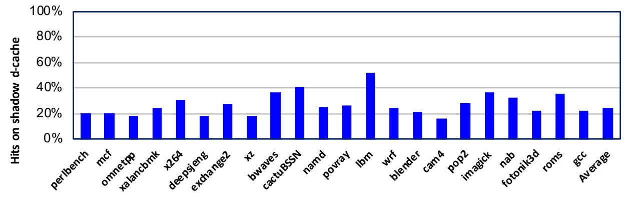


Figure 6: Percentage of hits on shadow d-cache

To gain more insight into the observed performance, Figure 5 shows the miss rate on read operations in the d-cache. There is little difference in behavior between SafeSpec and the baseline with respect to the data accesses. Figure 6 shows the percentage of the reads that hit the shadow structures.

The i-cache behavior is significantly different than the d-cache. Figure 7 shows the miss rate on the i-cache. For the i-cache, there are more substantial differences between WFC and the baseline. Some outlier behavior such as Pop2 and imagick where the percentage of i-cache misses drops significantly could be due to the larger size of the shadow structures expanding the effective size of the cache reducing conflict and capacity misses. Moreover, we see in Figure 8 that most of the hits occur in the shadow i-cache structure reflecting the high spatial locality of the access patterns in the i-cache; in other words, while a cache line is still speculative, several instructions execute from the same cache line. In contrast, the d-cache has less spatial locality, resulting in fewer accesses hitting the shadow state. We note that the cache miss rates are combined for all instructions

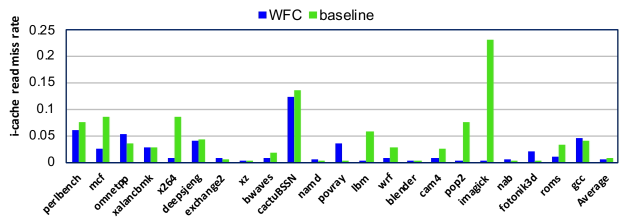


Figure 7: i-cache miss rate including the shadow i-cache

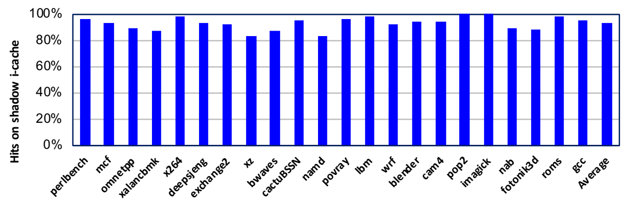


Figure 8: Percentage of hits on shadow i-cache

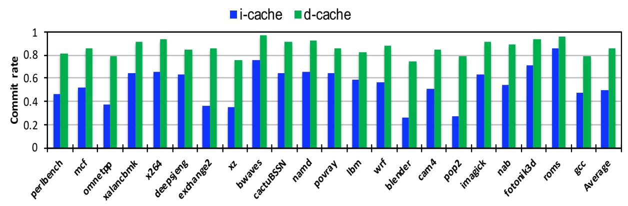


Figure 9: Commit rate of shadow state

Table 2: Security Analysis of Meltdown/Spectre

Spectre	WFC	WFB	Meltdown	WFC	WFB
Spectre-PHT [14, 15]	✓	✓	Meltdown [17]	✓	✗
Spectre-BTB [15]	✓	✓	Foreshadow [28, 29]	✓	✗
Spectre-STL [9]	✓	✗	Variant 3a [1]	✓	✗
Spectre-RSB [16, 18]	✓	✓	Lazy FP [24]	✓	✗
			Variant 1.2 [14]	✓	✗

(i.e., we do not exclude instructions that are squashed); therefore, many of these hits in the shadow structures may not end up being productive.

To understand the benefits of the shadow structure in filtering misspeculated accesses, Figure 9 shows the percentage of the shadow state that ends up being committed for the i-cache and the d-cache. We observe that a substantially higher percentage of the d-cache state ends up being committed, perhaps due to the fact that speculative loads are issued later in the pipeline making them more likely to commit. For both the d-cache and especially the i-cache, the shadow structure filters a large number of misspeculated accesses that are squashed without cluttering the caches.

Table 3: SafeSpec hardware overhead at 40nm.

	Power (mW)	Power (%)	Area (mm^2)	Area (%)
Secure	290.27	26.4	9.79	17
WFC	35.14	3	1.17	2

6.2 Security Analysis

Table 2 shows that both WFC and WFB close Spectre attacks, but only WFC is guaranteed to also stop Meltdown attacks. We evaluated our proof of concept code implementing Spectre in the simulator and found indeed that the attack fails under both WFC and WFB models. We evaluated the protection coverage for Spectre-style attacks targeting structures other than the d-cache (i-cache, iTLB, and dTLB). All three side channels were closed. We tested proof of concept code for the i-cache and a transient attack through the d-cache and observed that the attack fails on the *SafeSpec* protected CPU. We could not get TLB-based attacks working in the simulator, perhaps because of the large delays of page walks, or due to the limitations of the MarSSx86 models of the TLBs.

6.3 Hardware overhead

SafeSpec introduces hardware overheads to the CPU pipeline due to the addition of the shadow structures. We compared the hardware overhead for two different sizes for the shadow structures; 1) Secure: shadow structure size equal to the maximum speculative state during speculation; and 2) SafeSpec with WFC: shadow structure sizes were optimized based on 99.99% speculative state size for SPEC2017 benchmarks using the WFC implementation. We report the area, power, and access time values, as well as a percentage compared to the Skylake CPU L1 cache configuration (shown in Table 1), using CACTI v5.3 [23] in Table 3. The results show that the area overhead is tolerable for the secure design, making the design highly practical.

7 RELATED WORK

Multiple variants of Spectre have been proposed that exploit different control flow speculation triggers; Spectre versions that exploit the Pattern History Table which predicts a branch direction [4, 14, 15], the Branch Target Buffer [15], speculative the store-to-load forwarding [9], and the Return Stack Buffer [16, 18] have been demonstrated. On the other hand, Meltdown-type attack exploits speculative out-of-order instructions that lead to an exception. Multiple variants have been proposed: Meltdown-PF (Meltdown [17] and Foreshadow [28, 29]) exploit the *page fault*, Meltdown-BR (Variant 1.2 [14]) exploit the *bound range exceeded exception*, Meltdown-NM (Lazy FP [24]) exploit the *device-not-available exception*, and Meltdown-GP (Variant 3a [1]) exploit the *general protection fault.* Canella et al. summarize these and additional variants [2].

A range of defenses are starting to be proposed to prevent one or more components of speculative attacks including speculation prevention, providing secret data isolation, or that interfere with side channel communication. In general, these solutions are ad hoc, often focused on a specific attack. Moreover, most result in substantial performance impact. We overview some of the most promising defenses in the remainder of this section.

Speculation prevention: These defenses focus on preventing speculation by preventing misprediction [10, 11, 14, 20, 27] or faults [28]. They focus on protecting against specific attack or attack type; Spectre-PHT [10, 14, 20], Spectre-BTB [10, 27], Spectre-RSB [11], or Spectre-STL [14]. Furthermore, they hurt performance, potentially significantly, since they limit the speculation.

Secret data protection: These defenses focus on making sure that secret data can not be reached [7, 17]. However, they have limitations: Kernel Page-Table Isolation (KPTI) [7, 17] have performance overhead and some privileged memory locations must always remain mapped in user space due to x86 design [2], and Site Isolation [22] limits the amount of data that is exposed to side-channel attacks but attacks are still possible.

Dynamically Allocated Way Guard (DAWG): DAWG [13] is a method to securely partition the cache at the cache way granularity to provide isolation between protection domains. Therefore, it requires changes to the cache and coherence protocol. In addition, it requires domains enforcement management in software. While this solution, similar to our defense, prevents leaking the data through a side-channel, it only protects across isolation domains and not those performed within the same address space or isolation domain.

InvisiSpec: most relevant to our work, and developed concurrently with it (SafeSpec technical report was disclosed in June, 2018 [12]), is an architectural solution called InvisiSpec [30]. Like SafeSpec, InvisiSpec is designed to make transient loads invisible in the cache hierarchy. InvisiSpec focus is on cache coherence and memory consistency rather than understanding the implications on a single core. They did not consider transient side channels on the shadow structures, sizing issues, or carry out overhead characterization. Moreover, InvisiSpec was focusing on protecting the d-cache while we developed attacks and defenses on i-cache and the TLB, applying the principle more widely.

8 CONCLUDING REMARKS

We presented a general principle for supporting speculative execution in a way that makes out-of-order processors immune to speculation-based attacks. The principle relies on leaving speculative state in shadow structures, and only committing this state once the instructions that generate them are guaranteed to commit. Thus, side-effects of misspeculation are hidden from the primary structures of the CPU, closing the vulnerability. We demonstrated the principle to protecting caches and TLBs of the CPU. Our design completely closes all published attacks, as well as new variants that we developed to leak through the i-cache or the TLBs. We showed that careful design is needed to prevent a form of leakage that can arise while instructions share the speculative state. We mitigate this leakage by sizing the speculative state conservatively. Constructed this way, transient attacks also become impractical. The performance of the *SafeSpec* CPU was actually slightly higher than an unmodified CPU, despite conservative estimates on the shadow state. We believe that the presented design represents a first step of many towards a principled protection of speculative execution. To provide complete protection, other microarchitectural states that can be updated speculatively should use the same principle.

9 ACKNOWLEDGEMENT

This material is based on research sponsored by the National Science Foundation grants CNS-1619322, CNS-1422401, CNS-1619450, and CNS-1718997, Office of Naval Research grant N00014-17-1-2893, and Intel Side Channel Academic Program.

REFERENCES

[1] ARM. Vulnerability of speculative processors to cache timing side-channel mechanism. https://developer.arm.com/support/security-update, 2018.

[2] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441* (2018).

[3] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[4] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N., and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. In *Proc. of ASPLOS* (2018).

[5] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proc. of USENIX Security* (2018).

[6] Gregg, B. KPTI meltdown initial performance regressions, 2018. http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[7] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is dead: long live KASLR. In *Proc. of ESSoS* (2017).

[8] Horn, J. Reading privileged memory with a side-channel, 2018.

[9] Horn, J. speculative execution, variant 4: speculative store by-pass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[10] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[11] Intel. Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, 2018.

[12] Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *arXiv preprint arXiv:1806.05179* (2018).

[13] Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., and Emer, J. Dawg: A defense against cache timing attacks in speculative execution processors.

[14] Kiriansky, V., and Waldspurger, C. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[15] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *Proc. of S&P* (2019).

[16] Koruyeh, E., Khasawneh, K., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *Proc. of WOOT* (2018).

[17] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *Proc. of USENIX Security* (2018).

[18] Maisuradze, G., and Rossow, C. ret2spec: Speculative execution using return stack buffers. In *Proc. of CCS* (2018).

[19] Maisuradze, G., and Rossow, C. Speculose: Analyzing the security implications of speculative execution in CPUs. *arXiv preprint arXiv:1801.04084* (2018).

[20] Oleksenko, O., Trach, B., Reiher, T., Silberstein, M., and Fetzer, C. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506* (2018).

[21] Patel, A., Afram, F., and Ghose, K. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *Proc. of QUF* (2011).

[22] Projects, T. C. Site isolation. http://www.chromium.org/Home/chromium-security/site-isolation.

[23] Shivakumar, P., and Jouppi, N. P. Cacti 3.0: An integrated cache timing, power, and area model, 2001. Technical Report 2001/2, Compaq Computer Corporation.

[24] Stecklina, J., and Prescher, T. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).

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

[26] Tung, L. Linux meltdown patch: 'up to 800 percent cpu overhead', netflix tests show, Feb. 2018. ZDNet article: https://www.zdnet.com/article/linux-meltdown-patch-up-to-800-percent-cpu-overhead-netflix-tests-show/.

[27] Turner, P. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[28] Van B., J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proc. of USENIX* (2018).

[29] Weisse, O., Van, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T., and Yarom, Y. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Tech. rep., 2018.

[30] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., and Torrellas, J. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proc. of MICRO* (2018).

[31] Yarom, Y., and Falkner, K. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *Proc. of USENIX Security* (2014).