Mitigating Speculative Execution Attacks via Context-Sensitive Fencing

Mohammadkazem Taram
Department of Computer Science and Engineering,
University of California at San Diego,
La Jolla, CA 92093 USA

Dean Tullsen
Department of Computer Science and Engineering,
University of California at San Diego,
La Jolla, CA 92093 USA

Ashish Venkat
Department of Computer Science,
University of Virginia,
Charlottesville, VA 22904 USA

Editor's notes:
This article proposes a microcode-level defense against speculative execution attacks.
—Jeyavijayan "JV" Rajendran, Texas A&M University

The tension between security and performance has always been an important theme. However, Spectre and related attacks [1], [2] have exposed new dimensions of that tension, exploiting some of the most fundamental architectural performance features. Mitigating most variants of Spectre, as a result, potentially requires highly intrusive changes to the existing out-of-order processor design, severely limiting performance. Although Intel has announced microcode updates to mitigate certain variants of the attack, a majority of the high-impact vulnerabilities still largely rely on software patching [3]. Software mitigations take advantage of *fences* that mute specific effects of speculative execution by constraining the order of certain memory operations, or in some cases by completely serializing a portion of the dynamic instruction stream. Liberal fence insertion (e.g., at every load) can mitigate the attacks, but doing so severely hurts the performance; however, deploying fences more strategically at appropriate locations in the code requires extensive patching, resulting in significant engineering effort and long delays to deployment. We need architectures that can more seamlessly and quickly react to such attacks via unobtrusive field updates.

This work proposes context-sensitive fencing (CSF), a novel microcode-level defense against Spectre. The key components of the defense strategy include: 1) a *microcode customization* mechanism that allows processors to surgically insert fences into the dynamic instruction stream to mitigate undesirable side effects of speculative execution and 2) a *decoder-level information flow tracking* framework that identifies potentially unsafe execution patterns to trigger microcode customization.

Digital Object Identifier 10.1109/MDAT.2022.3152633
Date of publication: 17 February 2022; date of current version: 22 June 2022.

This defense can operate on legacy binaries, including completely untrusted code, without the need for recompilation or binary translation. Among the significant advantages of this approach is that a global solution to a new speculation-based attack can be deployed quickly via microcode update, without the need to wait for every software vendor to deploy a solution and update their products.

This work analyzes a significantly expanded suite of fences, considering various possible new enforcement stages and enforcement strategies. We introduce a new fence that prevents speculative updates to cache state with minimal interference in the dynamic scheduling of instructions, considering both a version that minimizes disruption of dynamically scheduled instruction execution in a nonshared-memory scenario, and a slightly more constrained version that respects total store order (TSO) semantics for shared-memory execution. To mitigate a larger suite of Spectre variants, we also introduce new fences that protect against attacks that exploit port contention [4] and memory disambiguation prediction [5].

CSF, a novel decoder-level information flow tracking-based detection mechanism, has the ability to automatically and dynamically identify the points in the program that require a fence. Information-flow tracking at the front-end of the pipeline is a new challenge, as all operations at that point are speculative. We develop a solution that addresses both the resulting overtainting and undertainting issues.

Architectural overview

This section describes the overall architectural approach of our defense mechanism—CSF. Figure 1 shows the expected implementation. Key to our approach is an ×86 microcode engine that enables context-sensitive decoding (CSD) [6], allowing it to optionally translate a native ×86 instruction into a customizable, alternate set of micro-ops. CSF leverages this capability to dynamically perform conservative insertion of fences at potentially vulnerable Spectre code targets. To this end, we introduce new custom micro-op flows and new configuration mechanisms that trigger such micro-op flows.

The CSD-enabled microcode engine is provisioned with fine-grained reconfiguration capabilities via a set of model-specific registers (MSRs) that can control the frequency, type, and enforcement criteria of fences inserted into the dynamic instruction stream. Such a fine-grained runtime reconfiguration capability is especially important to this work because speculation fences have a high-performance cost. This approach allows us to surgically insert fences that impose varying degrees of restrictions on speculative execution, depending upon the runtime conditions, current level of threat, and the nature of the code being executed.

Moreover, CSF benefits from a novel decoder-level information flow tracking engine [7] that has the ability to identify instructions with untrusted inputs that can potentially form a Spectre gadget and can then trigger alternate micro-op flows that insert speculation fences. Being in the decoder, and thus inherently speculative, decoder-level information flow tracker (DLIFT) relies on mistaint detection and recovery hardware implemented in the execute stage to avoid overtainting and undertainting scenarios. Finally, CSF also includes hardware and microcode-level mechanisms to achieve branch predictor state isolation across protection domains to mitigate the variant-2 and variant-4 attacks. Both these enhancements are described in detail in [7].

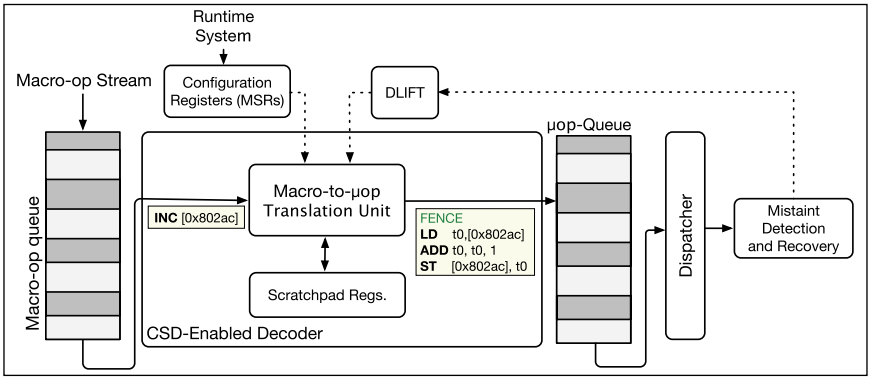

Figure 1. Architectural overview.

Design and implementation

This section describes, in greater detail, the architectural techniques and building blocks that together constitute the proposed defense strategy—CSF.

Microcode customization

Speculation fences are a processor's primary mechanism to override speculative execution. For Spectre variant-1, CSF works with CSD by providing alternate decodings of all load instructions, with the alternate decoding always including a *fence* micro-op that appears before the load micro-op. The alternate decoding will then be triggered or not based on runtime conditions. The first consideration, then, is what fence instruction to incorporate. Most processors already provide a variety of fences and serializing instructions. However, none of these were designed for security. Several were designed for synchronization, and thus with very different priorities. Others are just unintentional side effects of other operations, most of which are expected to be rare, so the serialization aspects are typically implemented with little attention to performance. In this study, we find ample opportunity to significantly shift and reduce the aforementioned tension between performance and security—identifying and eliminating several key dimensions of these fences that restrict performance with no actual benefit for security.

The following sections look at fence options, from existing alternatives to design options for novel, targeted fences.

- *Serializing instructions and memory fences:* Serializing instructions are the strictest among all speculation fences and completely override speculative execution. Upon decoding a serializing instruction, the processor stalls fetching any subsequent instruction until all instructions preceding the serializing instruction retire. Due to the high pipeline depth and issue width of modern out-of-order superscalars, the usage of serializing instructions could result in long delays and considerable throughput loss. Memory fences, on the other hand, enforce a memory serialization point in the program instruction stream. More specifically, these fences ensure that memory operations that appear after the fence are stalled until all prior memory requests (and the fence) complete execution.

  Intel's SFENCE instruction does not allow stores to pass through it, but does not affect loads. The MFENCE instruction restricts all memory operations from passing through it. LFENCE, unlike SFENCE and MFENCE that are memory ordering operations, is actually a serializing operation. In particular, LFENCE does not stop the processor from fetching and decoding instructions that appear after the LFENCE, but it restricts the dispatch of such instructions until the instructions preceding the LFENCE complete execution. Since LFENCE serializes execution and yet makes some progress in the frontend, it has been recommended by Intel as a low-overhead fence that can be inserted at vulnerable points in execution to defend against Spectre [3].
- *Fence enforcement policies:* Since none of the existing instructions that provide fence support were actually created for the purpose we (or the whole industry) need for Spectre mitigation, it is useful to consider the landscape of existing fences and potential fence implementations. We examine several possible properties of fences in this section.

Early versus late enforcement

We first categorize fences into *early-enforced* and *late-enforced* fences, based on the pipeline stage at which they are enforced. In particular, we refer to any serializing instruction, such as an LFENCE, that is enforced at the instruction queue or earlier in the pipeline as *early-enforced*. If the fence is enforced at a later stage such as the reservation station, the load/store queue, or the cache controller, we refer to it as a *late-enforced* fence. Late enforcement has two advantages: 1) it allows the frontend to make more progress in the presence of a fence and 2) it shifts the fence enforcement point toward the leaking structure (e.g., the cache), reducing the impact on instructions that do not access that structure, and allowing for the enforcement of more fine-grained serialization rules (e.g., allow cache hits but not misses). Figure 2 shows potential fence enforcement points at various stages in the processor pipeline.

The later a fence is enforced, the fewer the side channels it protects against. For example, Intel's LFENCE prevents information leakage through all microarchitectural structures that appear after the instruction queue. However, to prevent information leakage through the instruction cache side

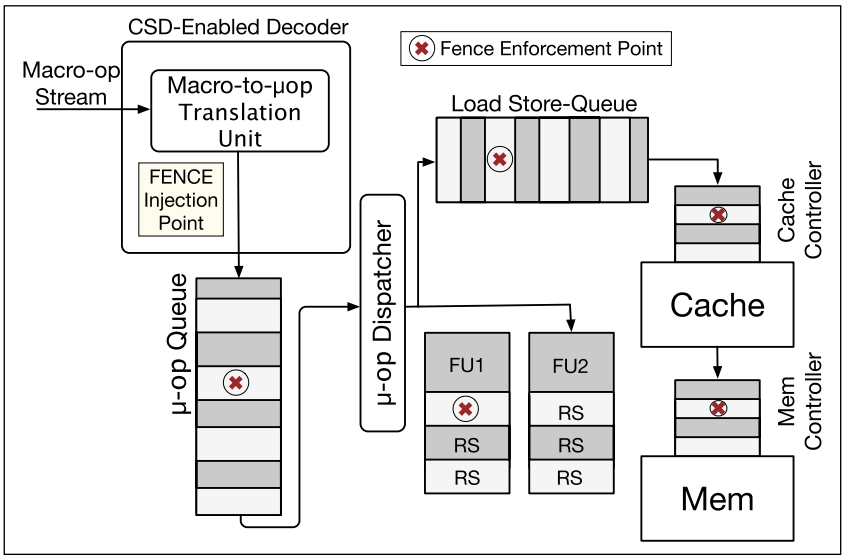

Figure 2. Fence enforcement points.

channel, we would have to resort to a regular serializing instruction, resulting in even higher-performance overheads. Similarly, a fence enforced at the data cache controller level would only mitigate data cache-based side channels and will not protect against an FPU-based side channel (refer to [7] for more detailed description of the attacks). Instead, the FPU-based side channel may be mitigated using a different fence that is appropriately configured and enforced at the reservation station, the FPU, or an execution port.

Strict versus relaxed enforcement

Depending upon how prohibitive they are, we next classify fences into *strict* and *relaxed*. *Strict* fences are highly prohibitive and do not allow any instructions to pass through them until the fence retires, whereas *relaxed* fences allow certain types of instructions to pass through them. For example, an alternate version of SFENCE that is enforced at the load/store queue and allows all instructions to pass except stores is a *late-enforced* and *relaxed* fence. On the other hand, all ×86 serializing instructions including LFENCE are *early-enforced* and *strict*. Customizing the micro-op stream with *early-enforced* and *strict* fences typically results in slower execution when compared to customization with *late-enforced* and *relaxed* fences. However, with carefully enforced constraints, the *late-enforced* and *relaxed* fences could offer similar, if not better, security guarantees.

Early versus late commit

A fence typically remains effective until it gets committed or squashed. According to Intel [8], a serializing instruction is only allowed to be committed if there is no preceding outstanding store that is waiting to be written back. While this behavior might be necessary for device synchronization or memory ordering enforcement, for the purpose of securing speculative execution against Spectre attacks, there is no need to wait for stores to be written back. This is because write buffers are not committed to the cache until the store reaches retirement, and therefore the fact that a store is waiting for an outstanding writeback request to complete is sufficient evidence that it did not occur along a misspeculated path.

Allowing a fence to commit early without waiting for preceding outstanding stores to write back can release the pipeline to again making forward progress, in some cases saving significant delays. Therefore, in this work, we propose and study the effects of an early-commit version of each fence.

Newly prsposed fences

To better understand the potential for fences beyond those that already exist, we propose and evaluate six new types of fences, as summarized in Table 1. We describe each of them in greater detail below.

The LSQ-LFENCE and the LSQ-MFENCE are relaxed fences enforced at the load/store queue. While LSQ-LFENCE is in effect, it does not allow any subsequent

load instruction to be issued out of the load/store queue, thereby preventing the cache state from being changed by load instructions on misspeculated paths. Thus, the LSQ-LFENCE mitigates the Spectre variant-1 attack. On the other hand, the more restrictive LSQ-MFENCE does not allow any subsequent memory instruction (both loads and stores) to be issued out of the load/store queue, until the fence commits.

The CFENCE is a relaxed fence and is enforced at the cache controller level using the following set of rules. First, like any other fences, it also allows all preceding instructions to proceed. Second, since store instructions do not commit the contents of the write buffer until the instruction retires, they are unaffected by the CFENCE. Finally, it labels any subsequent load as a *nonmodifying load* and allows it to pass through the fence, but the load is restricted from modifying the cache state. In particular, a *nonmodifying load* that results in a cache hit is allowed to read the contents of the cache, but is restricted from changing the LRU and other metadata bits. A *nonmodifying load* that results in a cache miss is marked as uncacheable, allowing the memory read request to complete without altering the cache state. In this way, we avoid updating the cache state upon encountering a speculative load and do not leave any observable cache footprint along misspeculated paths, thereby mitigating the Spectre variant-1 attack. For most well-behaved programs, the cache miss rate will be small enough that using CFENCE results in considerably lower performance overhead than other types of fences.

This approach works on an application not utilizing crosscore shared memory or any program running on a processor with a relaxed consistency model. However, it is important to extend our protection to cross-core shared memory applications as they still constitute a significant proportion of today's software systems. Later in this section, we introduce a constrained version of this fence that works on shared-memory applications with a strong consistency model.

PFENCE is a relaxed fence that is enforced at a specific execution port, and while in place, does not allow any instruction to be dispatched through that execution port. For example, PFENCE can sit in the execution ports that are associated with the floating-point unit and then not allow floating-point instructions to be executed speculatively.

Table 1. Characteristics of different fence types.

Fence Name	Enforcement Point	Strict/ Relaxed	Instructions not Allowed	Mitigates Variants	Existing/ New
Intel's SIs (CPUID)	Fetch	Strict	All	All	Existing
LFENCE	IQ	Strict	All	All	Existing
LSQ-LFENCE	LSQ	Relaxed	Ld	v1	New
LSQ-MFENCE	LSQ	Relaxed	Ld&St	v1,v1.1,v1.2	New
CFENCE	CC*	Relaxed	None	v1	New
CFENCE-TSO	CC*	Relaxed	None	v1	New
PFENCE	RS	Relaxed	FP	FP/SmotherSpctre	New
DFENCE	Disambiguation*	Relaxed	Ld	v4	New

* CC: Cache Controller, RS: Reservation Station, Disambiguation: Mem. Disambiguation Predictor Unit

This protects against speculative execution attacks that use the contention on FPU or a particular execution port as a medium to leak information. In addition, since PFENCE is a relaxed fence, it does not affect the instructions that do not access the specific execution port. In the case of FPU-based attacks, it allows integer, load/store, and control instructions to proceed, minimizing the performance overhead of the defense. Since these types of attacks usually target infrequently used instructions (e.g., AVX2 instructions), a targeted fence like PFENCE can defend against them with minimal performance overhead.

DFENCE is a relaxed fence that is enforced at the memory disambiguation predictor unit and does not let load instructions pass if there is an inflight store whose address is not yet calculated. If all preceding stores have known addresses, then loads can be issued if there is no address conflict. By stopping the memory dependency prediction, DFENCE can protect against variants of the attack that exploit this kind of speculation, such as variant-4 or speculative store bypass.

Interactions with memory consistency

Memory consistency models define correct behavior of a shared memory system in terms of both allowed values that dynamic loads may return and the final state of the memory. TSO is a relatively strict memory consistency model implemented by × 86 that permits *store-to-load* reordering but forbids all other observable reordering of loads and stores. Therefore, for example, any observable *load-to-load* reordering is forbidden in TSO. However, to attain memory-level parallelism, modern processors allow speculative reordering of the loads while it is not observable; and upon detecting a violation, just like other speculative features in the processor, they also squash and rollback.

Recall that when an LSQ-LFENCE or an LSQ-MFENCE is in effect, no younger load is allowed to be issued out of the load/store queue. Therefore, the proposed LSQ fences comply with TSO, as they maintain the ability of the processor core to invalidate the speculatively loaded values. The proposed CFENCE, however, allows younger loads to be issued, but restrict them from modifying the cache state, until the CFENCE is in effect. In the case of directory-based cache coherency protocols, this can cause the shared cache to be accessed without updating the directory. This could potentially cause scenarios in which the core does not receive any invalidation for the speculatively read values and could therefore cause a TSO violation and incorrect execution.

On a machine with a weaker consistency model, this behavior is not guaranteed to the programmer and would thus require other synchronization primitives to enforce it, thus removing the need for the CFENCE to provide this guarantee.

CSF, due to its fine-grain reconfigurablity, has the ability to protect different regions of the programs with different fences. For multithreaded shared-memory programs that need the TSO consistency model, we introduce a new variant of CFENCE, called CFENCE-TSO. CFENCE-TSO allows the fence-passing younger loads to access the local caches (e.g., L1 and L2), but the load is restricted from being forwarded to the next-level shared caches. In the case of local cache misses, the request would get delayed until the CFENCE-TSO gets committed, that is, when the load is no longer speculative. Note that CFENCE-TSO can allow fence-passing younger loads to access a lower-level local cache (e.g., L2) only if the cache forward invalidation requests to the core. As with CFENCE, this does not inhibit the performance in the case of cache hits, but incurs a higher cost than CFENCE in the presence of frequent misses.

Fence frequency optimization

The most naïve, yet secure way to restrict speculation attacks is to liberally instrument every instruction of a vulnerable type (e.g., load instructions in the case of cache side channels) in the program with a fence micro-op. In fact, not every instance of a vulnerable instruction type is necessarily vulnerable; for example, all the loads in the program are not vulnerable to speculative attacks via cache side channels. Therefore, we can reduce the number of fences inserted. However, failing to insert even a single necessary fence would enable a Spectre attacker to read the whole victim's memory space, so it is of crucial importance that fence frequency optimizations be conducted meticulously. In the following, we introduce two secure optimizations for reducing the number of fences.

Basic block-level fence insertion

The source of the Spectre attack is dynamic control speculation, which implies that the speculation begins with a branch prediction and the processor starts speculatively executing along the predicted path. To fully mitigate this attack, we want a fence between each branch and subsequent loads; but if one branch is followed by four loads, we only need one fence to protect all four. Thus, in this optimization, we propose to only instrument the first instance of a vulnerable instruction type (e.g., first load) of each basic block. This is simply implemented by setting a flag in hardware whenever a branch is decoded and then insert a micro-op in the alternative load decoding that, along with the fence, also resets the flag. When the flag is not set, the original decoding is used.

Taint-based fence insertion

However, even one fenced load per basic block is likely still too conservative. In all known instances of the attacks, the attacker performs some operations (mostly memory read) based on untrusted data that leak information. For example, in Spectre variant-1, the attacker provides an out-of-bound index to an array. Here, we assume any information that comes from the user address space and input devices (e.g., via the ×86 IN instruction) is untrusted. In particular, we instrument *syscall* and *int* 0 × 80 instructions with a set of micro-ops that mark all the input registers of the syscall (e.g., rax, rdi, rsi, etc., on Linux × 86_64) as tainted. In addition, we also consider DMA'd pages as untrusted, for which we rely on the IOMMU to mark DMA'd pages as tainted in the page table. For the taint-based fence insertion optimization, we propose the insertion of fences for only vulnerable/tainted loads that operate on untrusted data, by leveraging the decoder-level information flow tracking strategy described in [7]. For the scenarios where the kernel is invoked without a syscall (e.g., exceptions and interrupts), we rely on an always-on fence insertion policy instead of the taint-based fence insertion to protect the kernel

code—a policy that is easily enabled by CSF's high level of dynamic configurability.

Methodology

We closely follow the experimental methodology described in [7], but here, to evaluate CFENCE-TSO, which is aimed at protecting shared-memory programs, we also experiment with the multi-threaded PARSEC benchmarks. In that experiment, we simulate a 4-core processor with 2 MB last-level cache and a defense that is always on, with basic block-level fence insertion for all user and kernel memory accesses.

Evaluation

Figure 3 measures the performance impact of three different fences: 1) the standard × 86 LFENCE; 2) the LSQ-MFENCE; and 3) the CFENCE, all enforced with late commit, pessimistically inserted for every kernel load in the program. CFENCE incurs the lowest performance overhead, since it is less restrictive than the other two and is enforced at a much later pipeline stage. Overall, CFENCE reduces the incurred performance overhead due to fencing by 2.3×, bringing down the execution time overhead from 48% to 21%.

Furthermore, our experimentation with the early-commit version of the CFENCE shows that, by allowing the fence to commit earlier, it consistently outperforms the late-commit version, saving about 4% in overall execution time on average.

In addition to CFENCE, we also measure the performance effects of *PFENCE-FPU* that protects against the variants of the attack that exploit FPU, *PFENCE-MULTDIV* which allows all the instructions to pass except multiplication and division instructions to protect against port contention attacks, and *DFENCE* that protects against variant-4, or speculative store bypass. Pessimistically inserting PFENCE-FP, PFENCE-MULTDIV, and DFENCE for all the vulnerable instructions in the kernel code, on average, incur only 3.7%, 3.4%, and 1.8% execution time overhead, respectively. Furthermore, our experiments with PARSEC benchmarks show that CFENCE-TSo, in spite of the liberal fence insertion (for all user and kernel loads, once per basic block) and the always-on nature of the defense, degrades the performance by just 20%.

Figure 4 shows the performance of our proposed fence frequency optimization techniques. In the first optimization, we inject the CFENCE only for tainted loads, as indicated by the DLIFT engine. This reduces the performance overhead of the defense from 21% to 11% on average. We further optimize the number of fences inserted by performing basic

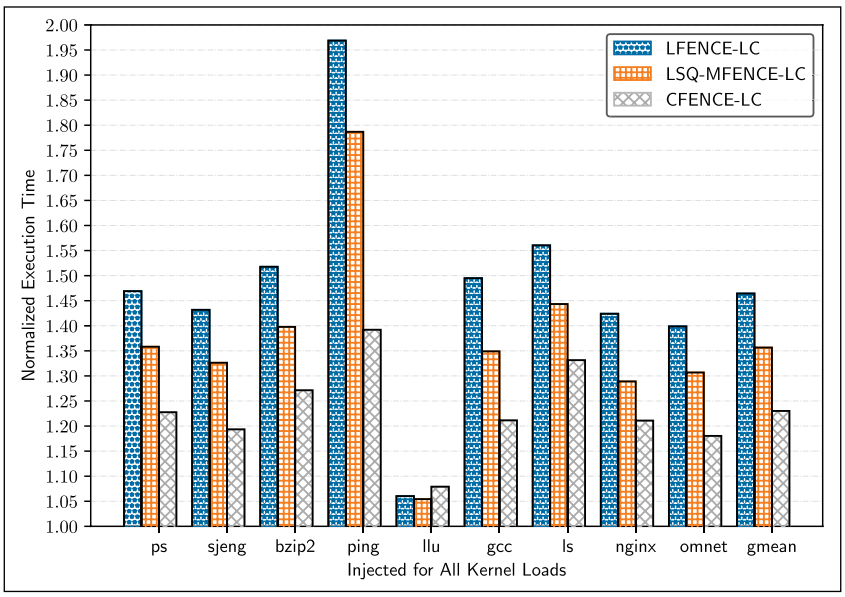

Figure 3. Execution time of different fence enforcement levels (normalized to insecure execution).

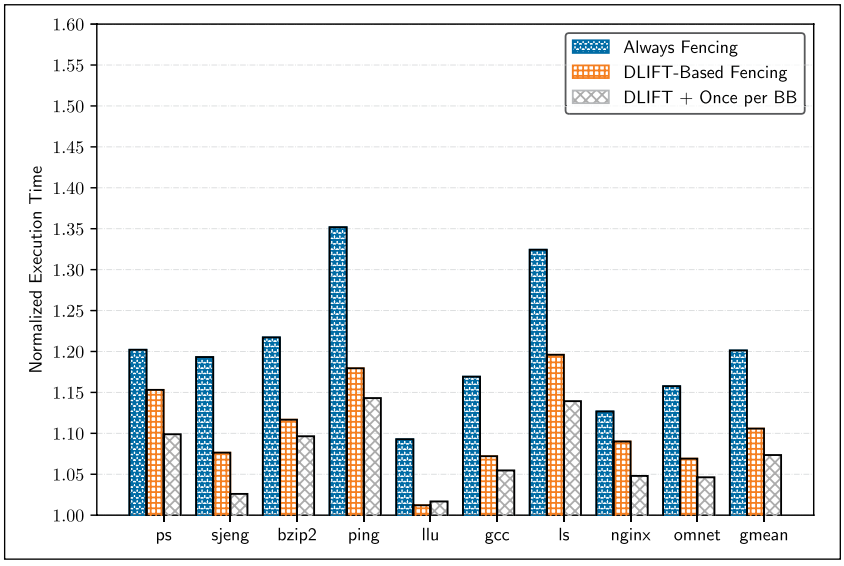

Figure 4. Impact of fence frequency optimizations on execution time.

block-level fence insertion. This results in an additional 4% improvement in performance.

Related work

Several works have focused on mitigating the speculative execution attacks at the hardware level. SafeSpec [9] and InvisiSpec [10] propose mitigating the side effects of speculative execution by adding new user-invisible shadow structures for caches and TLBs that store transient results from speculative instructions and commit them to the main cache/TLB only if the speculation was deemed correct and the corresponding instructions gracefully retire. Even more recently, STT [11] and ConTExT [12] have also used taint tracking techniques that allow tracking of spurious information flow along misspeculated paths, enabling more fine-grained protection against covert communication that hinges on using speculatively accessed data. CSF's approach toward information flow tracking is different than those works. They consider any speculatively loaded value as untrusted, whereas CSF's notion of trust is similar to a classic information flow tracking where any data that comes from untrusted channels is considered as untrusted.

In this work, we propose CSF, a set of architectural techniques that provide high-performance defense against Spectre-style attacks. We show that we can reduce fencing overhead by a factor of 6 compared to a conservative fence insertion method. This is done by injecting fence instructions dynamically with no recompilation and binary translation required. This allows us to employ runtime information to strategically insert fences when needed. This work also introduces new fence primitives which protect sensitive structures from speculation-based microarchitectural effects, with minimal impact on instruction throughput in the pipeline. We introduce protections for a wide variety of attacks via a small set of targeted fence implementations and an adaptive framework for fence insertion. ■

Acknowledgments

We would like to thank the anonymous reviewers for their helpful insights. We would also thank Josep Torrellas for helpful suggestions on this work. This work was supported in part by NSF under Grant CNS-1652925 and Grant CNS-1850436, in part by NS-F/Intel Foundational Microarchitecture Research under Grant CCF-1823444 and Grant CCF-1912608, and in part by DARPA under Agreement HR0011-18-C-0020. This article is an extended version of the paper, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," presented at the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

References

[1] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," Jan. 2018, *arXiv:1801.01203v1*.

[2] M. Lipp et al., "Meltdown: Reading kernel memory from user space," in *Proc. USENIX Secur. Symp.*, 2018, pp. 1–18.

[3] Intel Corporation, "White paper: Intel analysis of speculative execution side channels," Intel, Santa Clara, CA, USA, Tech. Rep. 336983-001, Revision 1.0, Jan. 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel- Analysis- of-Speculative-Execution- Side- Channels.pdf

[4] A. Bhattacharyya et al., "SMoTherSpectre: Exploiting speculative execution through port contention," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2019, pp. 785–800.

[5] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018, *arXiv:1807.03757*.

[6] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2018, pp. 624–637.

[7] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2019, pp. 395–410.

[8] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., Santa Clara, CA, USA, Mar. 2009.

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

[10] M. Yan et al., "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2018, pp. 428–441.

[11] J. Yu et al., "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2019, pp. 954–968.

[12] M. Schwarz et al., "ConTExT: A generic approach for mitigating spectre," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2020, pp. 1–18.

Mohammadkazem Taram is pursuing a PhD with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA, USA. He is interested in computer architecture, security, and privacy. Taram has an MS in computer engineering from Sharif University of Technology, Tehran, Iran.

Ashish Venkat is an Assistant Professor with the Department of Computer Science, University of Virginia, Charlottesville, VA, USA. His research interests are in computer architecture and compilers, especially in instruction set design, processor microarchitecture, binary translation, code generation, and their intersection with computer security and machine learning. Venkat has a PhD from the University of California at San Diego, La Jolla, CA, USA.

Dean Tullsen is a Professor with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA, USA. He works in the areas of computer architecture, compilers, and security. He is a Fellow of IEEE and ACM.

■ Direct questions and comments about this article to Mohammadkazem Taram, Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093 USA; mtaram@cs.ucsd.edu.