Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization

Mohammadkazem Taram
University of California San Diego
mtaram@cs.ucsd.edu

Ashish Venkat
University of Virginia
venkat@virginia.edu

Dean Tullsen
University of California San Diego
tullsen@cs.ucsd.edu

Abstract

This paper describes context-sensitive fencing (CSF), a microcode-level defense against multiple variants of Spectre. CSF leverages the ability to dynamically alter the decoding of the instruction stream, to seamlessly inject new micro-ops, including fences, only when dynamic conditions indicate they are needed. This enables the processor to protect against the attack, but with minimal impact on the efficacy of key performance features such as speculative execution.

This research also examines several alternative fence implementations, and introduces three new types of fences which allow most dynamic reorderings of loads and stores, but in a way that prevents speculative accesses from changing visible cache state. These optimizations reduce the performance overhead of the defense mechanism, compared to state-of-the-art software-based fencing mechanisms by a factor of six.

CCS Concepts • **Security and privacy → Side-channel analysis and countermeasures**; *Systems security*; *Information flow control*; • **Computer systems organization → Architectures**.

Keywords secure architectures; side-channel attacks; Spectre; speculative execution; microcode; taint tracking

ACM Reference Format:
Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3297858.3304060

1 Introduction

Maximizing performance has been a major driving force in the economics of the microprocessor industry. Modern processor architectures feature highly complex and sophisticated performance optimizations. However, scaling performance without considering security implications could have serious negative consequences, as evidenced by the recent pile of lawsuits [96] concerning the Meltdown [61] and Spectre [57] class of microarchitectural attacks. These events have highlighted the need to architect systems that can not only run at high speed, but can also exhibit high resilience against security attacks, not just one or the other. The goal of this work is to secure a particular performance optimization integral to modern processor architectures – speculative execution – against the Spectre class of microarchitectural attacks, while maintaining acceptably high levels of performance.

Modern processors employ branch speculation and out-of-order execution to take advantage of the available instruction-level parallelism beyond control-flow boundaries, thereby improving the overall CPU resource utilization and sustaining high throughput. Spectre attacks exploit speculative execution by leaking secret information along misspeculated paths via cache-based and other timing side channels. Owing to their ability to mistrain the branch predictor to deliberately steer execution to an attacker-intended control-flow path [49, 58], these attacks have notably demonstrated the potential to break all confidentiality and completely bypass important hardware/software security mechanisms such as ASLR, even via remotely accessed covert channels [76].

Mitigating Spectre is a particularly hard problem since it could potentially cause highly intrusive changes to the existing out-of-order processor design, severely limiting performance. Although Intel has announced microcode update patches to mitigate certain variants of the attack, a majority of the high impact vulnerabilities still largely rely on software patching [45, 72]. State-of-the-art software countermeasures 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 bounds check) can mitigate the attacks, but doing so severely hurts performance; however, spraying fences more strategically at appropriate locations in the code requires extensive patching of software via recompilation or binary translation – resulting in significant engineering effort and long delays to deployment. We need

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS'19, April 13–17, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN ISBN 978-1-4503-6240-5/19/04...$15.00
https://doi.org/10.1145/3297858.3304060

hardware architectures that can more seamlessly 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: (a) a *microcode customization* mechanism that allows processors to surgically insert fences into the dynamic instruction stream to mitigate undesirable side-effects of speculative execution, (b) a *decoder-level information flow tracking* (DLIFT) framework that identifies potentially unsafe execution patterns to trigger microcode customization, and (c) *mistraining mitigations* that secure the branch predictor and the return address stack.

To perform secure microcode customization with minimal impact on performance, this work leverages context-sensitive decoding (CSD) [80], a recently proposed extension to Intel's (and others') micro-op translation mechanism that enables on-demand and context-sensitive customization of the dynamic micro-op instruction stream. In addition, this work also takes advantage of the reconfiguration framework offered by CSD, allowing the operating system and other trusted entities to dynamically control the frequency, type, and behavior of fences that are surgically inserted into the micro-op stream to secure speculative execution.

This work analyzes a significantly expanded suite of fences, considering different possible enforcement stages and enforcement strategies. In particular, we introduce a new fence that prevents speculative updates to cache state with minimal interference in the dynamic scheduling of instructions.

Context-sensitive fencing has the ability to automatically identify fence insertion points via a novel decoder-level information flow tracking-based detection mechanism. Since the processor front-end typically churns instructions at a much higher rate than the rest of the pipeline, information-flow tracking at the decoder-level is prone to frequent overtainting and undertainting scenarios. While overtainting could hurt performance due to the increased frequency of fence insertion, undertainting could undermine the security of the system for a brief window. By solving these challenges through an early and low-overhead mistaint detection and recovery mechanism, this paper further establishes the viability of decoder-level information flow tracking as an effective attack detection mechanism.

This work further proposes novel micro-op flows that protect the branch predictor, the branch target buffer, and the return address stack against mistraining across different protection domains. While similar in spirit to the proposed Indirect Branch Predictor Barrier (IBPB) instruction by Intel, these micro-op flows apply more generally to a broader class of branch predictors and return address stacks, and offer more fine-grained control.

This research makes the following major contributions:

- It introduces context-sensitive fencing, a mechanism that leverages a dynamic decoding architecture to inject fences between control flow and loads, without recompilation or binary translation.
- It examines several variants of existing fences, and introduces three new fences, which allow aggressive reordering of loads and stores without exposing any microarchitectural evidence, in the cache, of speculative accesses that cross the fence.
- It introduces a decoder-level dynamic information flow tracker, DLIFT, which allows accurate tracking of taints early in the pipeline, giving the pipeline the ability to identify tainted accesses *before* the stages that enable speculative execution.
- It introduces a simple dynamic mechanism that eliminates redundant (per basic block) fences.
- The combination of optimizations introduced in this paper reduce the cost of a fence-based Spectre mitigation technique from 48% overhead (for a conservative scheme) to less than 8%.
- It also introduces a decode-level branch predictor isolation technique that mitigates branch mis-training variants of Spectre.

2 Background and Related Work

Speculative Execution. Dynamic control speculation is a well-known instruction throughput optimization technique used, especially in out-of-order processors, to predict the branch outcome and execute instructions along the predicted path, while waiting for the actual branch outcome to be evaluated at a later pipeline stage. In the event of a misprediction, the processor rolls back execution along the misspeculated path and redirects control to the right branch target. Although speculative execution is largely programmer-invisible in terms of committed architectural register and memory state, in all known instances it leaves some microarchitectural side effects that can be observed through well-established side channels.

Microarchitectural Attacks. Microarchitectural attacks leak secret information of a victim process by observing microarchitectural effects of certain performance/power optimizations such as caches, branch speculation, memory disambiguation, and even dynamic voltage and frequency scaling (DVFS), through side-channels such as memory bus activity [3], power consumption characteristics [7], branch access patterns [1, 30], faults [42, 51], acoustics [33, 34], electromagnetic effects [31], functional unit timing characteristics [94], and most notably cache access patterns [2, 32, 52, 62, 68, 97, 100, 101].

Cache-based side-channel attacks have been shown to reveal secret information such as cryptographic keys [27, 38, 100], keystrokes [37], and browsing activity [70] by co-locating a spy process alongside a victim in such a way that

Table 1. Speculative Attacks Variants

Variant	Vulnerability Name
Spectre v1 [49, 57]	Bounds Check Bypass (BCB)
Spectre v2 [49, 57]	Branch Target Injection (BTI)
Spectre v3 [49, 61]	Rogue Data Cache Load (RDCL)
Spectre v3a [6, 67]	Rogue System Register Read (RSRD)
Spectre v4 [67]	Speculative Store Bypass (SSB)
Spectre-NG [78]	Lazy FP State Restore
Spectre v1.1 [54]	Bounds Check Bypass Store (BCBS)
Spectre v1.2 [54]	Read-only Protection Bypass
Spectre v5 [59, 66]	Ret2Spec and SpecRSB
NetSpectre [76]	Remote Bounds Check Bypass
Foreshadow [87, 95]	L1 Terminal Fault

```
1 if(x < array1_size)
2         y = array2[array1[x] * 256];
```

Figure 1. Example Spectre Variant-1 Gadget

they share cache memory. The attack unfolds through a pre-attack step in which a spy process fills/flushes specific cache sets, so that the victim leaves observable side effects in terms of its cache access patterns, that can be later inferred by the spy process by timing the access to particular cache blocks.

Exploiting Speculative Execution. This work tackles a highly evasive class of microarchitectural attacks called Spectre [57] that leaks information by exploiting side effects of speculative execution through cache-based side channels. These attacks not only exploit unintended side effects due to speculation, but have the ability to deliberately mislead execution into attacker-intended paths by mistraining the branch predictor and other associated structures. Several variants of the attack have been described (shown in Table 1) that can potentially bypass software security/integrity protection mechanisms such as bounds checking and ASLR [77].

Figure 1 shows a vulnerable code target for the variant-1 attack. The code fragment is composed of a conditional branch that performs a bounds check, and a *Spectre gadget* that results in an observable microarchitectural side effect upon execution. Upon misspeculation, the bounds check is bypassed and the Spectre gadget executes, leaving an observable cache footprint to the attacker, that remains even after the processor detects the misprediction and rolls back execution. In the simplest variant of the attack where the attacker controls both *x* and *array2*, the attacker can potentially leak the entire address space of the victim.

The variant-2 Spectre attack further allows the attacker to hijack speculative execution by mistraining the branch predictor and associated structures, enabling a ROP-style attack [75] that stitches together Spectre gadgets. To mount such an attack, a co-located adversary process that shares the branch predictor (e.g., a browser process with several user threads) with the victim, first forces an artificial BTB (Branch Target Buffer) entry collision using a carefully chosen indirect branch address. The attacker next poisons the value of the colliding BTB entry by repeatedly executing its

```
1  mov eax, arr1_size
2  cmp edi, eax
3  jge END_LBL
4  mov eax, edi

6  mov eax, [eax+arr1]
7  shl eax, 0x8
8  mov eax, [eax+arr2]
9  mov [y], eax
10 END_LBL:
```

(a) Vulnerable gadget (x86)

```
mov eax, arr1_size
cmp edi, eax
jge END_LBL
mov eax, edi
____FENCE____
mov eax, [eax+arr1]
shl eax, 0x8
mov eax, [eax+arr2]
mov [y], eax
END_LBL:
```

(b) Fenced gadget (x86)

Figure 2. Mitigating Spectre-v1 using a Fence Instruction

own branch whose target is the address of a Spectre gadget. After successful mistraining of the branch predictor, the processor now predicts the victim's indirect branch that collides in the BTB to be the attacker-chosen Spectre gadget and starts speculatively executing it.

The variant-3 attack (a.k.a. Meltdown) exploits the fact that most out-of-order processors that employ dynamic speculation supress loads with protection violation at instruction retirement (i.e., commit stage), rather than during instruction execution. Therefore, a non-privileged memory access can find its way to the cache and leave a footprint, allowing an attacker to now effectively read arbitrary memory contents of another process or even the kernel or the hypervisor, through careful cache-based side channel analysis.

The literature describes multiple attacks that conform to the above variants, but exploit different attack targets and/or different side channels. These include SgxPectre [13] that bypasses Intel's SGX [44] security mechanisms to steal secrets from SGX enclaves, the MeltdownPrime/SpectrePrime [82] that leverage a PRIME+PROBE [62] cache attack instead of FLUSH+RELOAD [100] by exploiting the side effects of cache line invalidation mechanisms in modern cache coherence protocols, and the NetSpectre attack that leaks information across independent virtual machines on Google Cloud via an AVX-based covert channel [76].

The variant-4 Spectre attack exploits microarchitectural side effects of the memory disambiguation feature employed by most out-of-order processors in order to allow loads to be speculatively ordered and executed before any outstanding store whose effective address has not been calculated yet. Upon misspeculation, i.e, if the load address has a conflict with the outstanding store, the processor flushes the pipeline and triggers the re-execution of the load and all subsequent instructions [48], while the microarchitectural side effects of the misspeculation linger, resulting in memory disclosures similar to variant 1 and 3. Similarly, the NG variant-3 attack [78] leverages the lazy x87 floating-point restore functionality of Intel CPUs to read floating-point registers of a victim process.

The Spectre variants 1.1 and 1.2, dubbed Speculative Buffer Overflows [55], exploit the store-to-load forwarding optimization to stitch together Spectre gadgets. More specifically, these attacks bypass a stack buffer overflow check similar to variant-1 and execute a Spectre gadget that speculatively stores malicious content (the address of the next Spectre gadget) into the return address on the stack. Store-to-load forwarding combined with the fact that most modern processors break down the *return* instruction into micro-ops that *load* the return address from the stack before transferring control, then result in a ROP-style execution of Spectre gadgets that leave a trail of microarchitectural state behind. The variant-5 attack [58] achieves similar effect, by instead mistraining the return address stack employed by most processors to speculatively predict the target of a procedure return.

Spectre Mitigations. The current set of mitigations for Spectre range from simple coding guidelines[10] to proposals that advocate exposing microarchitectural details in the ISA [64]. To mitigate Meltdown, Kernel Page Table Isolation (KPTI) [17, 36] has been proposed and recently patched to the Linux kernel, incurring about 6% in performance. To mitigate Spectre v1, multiple chip manufacturers including Intel [45], ARM [6], and AMD [4], have suggested instrumenting code with serializing instructions or fences to inhibit speculation at specific points in execution.

For example, consider Figure 2a that shows the assembly code for the Spectre variant-1 gadget in Figure 1. Figure 2b shows a software-patched version that employs a serializing instruction or a fence to prevent the speculative execution of the Spectre gadget (i.e., lines 6-9). In most implementations, upon decoding the fence, the processor stops fetching new instructions until the fence gets committed or squashed, thereby serializing execution. In our example, if the attacker calls the vulnerable gadget with out-of-bounds values in the *edi* register, the processor front-end stalls until the fence is committed, thereby disallowing the speculative execution of the Spectre gadget and completely mitigating the attack.

The associated performance overhead with liberal fence insertion, however, could be as high as 10x [69]. While it is possible to perform compiler-directed code instrumentation with fences at a lower performance overhead [72], locating the potential Spectre targets using static analysis is non-trivial and could therefore result in less than full coverage [56]. Speculative Load Hardening [11, 12] and YSNB (You Shall Not Bypass) [69] propose to use predicated execution [50, 65, 73] to alleviate the high overheads for fences by injecting an artificial data dependency between the conditional branch and the Spectre gadget. These proposals incur 36-60% overhead in performance.

To cope with variant-2, both Intel and AMD have announced microcode update patches that introduce new fence instructions [4, 45] that prevent instructions preceding the fence from controlling the indirect branch prediction of branches that follow the fence. The fences also prevent software with lower privileges from influencing the indirect branch prediction of software with higher privileges. Furthermore, these patches restrict indirect branch prediction from being controlled by co-located threads via simultaneous multithreading [83, 84]. Furthermore, the use of *retpolines* [45, 46, 85] have been advocated to replace indirect jumps and calls with an equivalent *push+ret* instruction sequence, in order to bypass the indirect branch predictor. However, they could further expand the attack surface in the wake of v5 [58, 66], v1.1, and v1.2 attacks [54].

On the hardware front, SafeSpec [53] and InvisiSpec [99] propose mitigating side effects of speculative execution by adding new shadow user-invisible structures for caches and TLBs that store transient results from speculative instructions, and committing them to main cache/TLB only if the speculation was deemed correct and the corresponding instructions gracefully retire. Although these techniques make disruptive changes to the processor/memory architecture and consistency models, they make significant strides in secure hardware design with minimal performance impact. Finally, Dong, et al. [28, 29] propose leveraging Virtual Ghost [22] to protect applications running on a compromised OS kernel from Spectre.

Secure Instruction Stream Customization. Prior work has established that instruction customization is an effective means to instrument the dynamic execution stream with security checks, and thereby mitigate several attack vectors at a relatively low performance overhead. Instruction stream customization has been proposed and evaluated in various forms and levels, ranging from secure virtual architectures at the ISA and compiler level [23, 88, 90] to full-blown processor binary translation at the microcode level [9, 25, 71, 89]. Corliss, et al. [18–20] propose dynamic instruction stream editing (DISE), a macro-engine that customizes the dynamic instruction stream at the decoder level, by pattern-matching user-defined production rules pushed into the decoder. Taram, et al. [80] propose *context-sensitive decoding* that leverages the CISC to RISC translation feature of modern instruction set decoders to dynamically alter the behavior of programmer-visible instructions without recompilation or binary translation [26, 91]. While we leverage these approaches in this research, this work targets a different class of emerging attacks, and proposes several additional techniques beyond microcode customization.

Information Flow Tracking. Suh, et al [79] first proposed the idea of dynamic information flow tracking (DIFT) that tags data from untrusted channels as spurious, and further tracks the information flow of spurious data, flagging any violation of an enforced security policy. Multiple hardware information flow tracking techniques have been described at various levels of the hardware [14–16, 21, 24, 81,

86, 92] as a detection mechanism to circumvent code injection [98], cross-site scripting attacks [24, 93], buffer overflows [24, 98], and SQL injection [24, 39, 40, 60, 102]. This work proposes decoder-level information-flow tracking as a detection mechanism for Spectre, and addresses several associated challenges.

3 Assumptions and Threat Model

Among the Spectre variants, the Bounds Check Bypass (variant-1) is the one which has had the highest impact in terms of affected platforms and devices. Therefore, the main focus of this work is to mitigate the variant-1 attack at an acceptable level of performance. However, owing to the flexibility advantages of microcode customization, we also propose new mitigations against other Spectre variants documented in Table 1.

We assume that the goal of the attacker is to read arbitrary memory contents by exploiting a Spectre gadget. Also, without loss of generality, in this work we defend against a Spectre attacker that targets the most obvious and the most important victim, i.e., the OS kernel, but our framework allows us to extend the proposed techniques to mitigate different targets such as virtual machines and browsers. Our attack model assumes an adversary with the following capabilities.

- **Information Leakage.** We assume that the adversary has the ability and privilege to probe, flush, or evict any cache line including that of the kernel at any particular time. Also, they can make precise timing measurements using instructions such as rdtsc and rdtscp.
- **Co-location.** We assume that the adversary is colocated with the victim, and can not only leak information through cache-based side channels, but can also mistrain shared branch predictor structures including branch target buffers and return address stacks, and can further influence the branch outcome of the victim, as described in the variant-2 attack.
- **User-Mode Access.** We assume that the adversary has standard user-mode access, and can invoke any system call with arbitrary and carefully chosen arguments, and can further access devices such as the keyboard or network to feed the kernel/driver code with carefully chosen data.

Moreover, we assume that the only communication channel between the attacker and the kernel code is through micro-architectural side channels and the attacker does not have any other channel, direct or indirect, to leak data. Among all the micro-architectural side-channels, data caches are the ones that are predominantly used and are the easiest to exploit. Thus, the mitigation strategies described in this work primarily focus on those variants of the attack that use the data cache as their covert channel to leak information. Again, due to the flexiblity and programmability of the proposed strategy, it is possible to easily extend the approach to mitigate other side-channels such as functional unit contention and AVX timing channels [76].

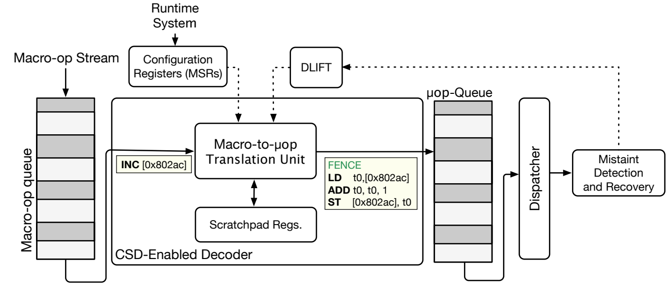

Figure 3. Architectural Overview

4 Architectural Overview

Figure 3 gives the architectural overview of our defense strategy – *context-sensitive fencing*. The central piece of the proposed architecture is an x86 microcode engine that has context-sensitive decoding (CSD) capabilities [80], allowing it to optionally translate a native x86 instruction into a customizable, alternate set of micro-ops. In this work, we leverage this capability to perform the surgical insertion of speculation fences (some existing and some newly proposed) 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 speculation fences inserted into the dynamic instruction stream. Such a fine-grained reconfiguration capability is especially important to this work because speculation fences are particularly expensive, allowing 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, the context-sensitive fencing framework also benefits from a novel decoder-level information flow tracking (DLIFT) engine that has the ability to identify untrusted instructions that are potentially in Spectre gadgets and trigger alternate micro-op flows that insert speculation fences. Owing to its decoder-level and inherently speculative implementation, DLIFT relies on a mistaint detection and recovery hardware implemented in the execute stage, to avoid overtainting and undertainting scenarios. Finally, the proposed defense framework 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.

Table 2. List of Intel's Serializing Instructions

Type	Instr.	Desc.
Privileged Serializing Instructions	INVD	Invalidate Internal Caches
	INVEPT	Invalidate Translations from EPT
	INVLPG	Invalidate TLB Entries
	INVVPID	Invalidate Translations Based on VPID
	LIDT	Load Interrupt Descriptor Table Register
	LGDT	Load Global Descriptor Table Register
	LLDT	Load Local Descriptor Table Register
	LTR	Load Task Register
	MOV	Move to Control Register
	MOV	Move to Debug Register
	WBINVD	Write Back and Invalidate Cache
	WRMSR	Write to Model Specific Register
Non Priv.	CPUID	CPU Identification
	IRET	Interrupt Return
	RSM	Resume from System Management Mode
Mem. Ordering	SFENCE	Store Fence
	LFENCE	Load Fence
	MFENCE	Memory Fence

5 Design and Implementation

In this section, we describe in greater detail the architectural techniques and building blocks that together constitute the proposed defense strategy – *context-sensitive fencing*. First, we describe a microcode customization framework that enables the surgical insertion of fences to secure speculative execution with minimal performance impact. Second, we examine the full design space of fences to provide the isolation properties we need without undue obstruction of existing pipeline performance features. Third, we propose a decoder-level information flow tracking (DLIFT) technique to follow potentially malicious execution patterns and trigger secure microcode customization. Finally, we propose protection mechanisms that circumvent the mistraining of the branch predictor and associated structures.

5.1 Microcode Customization

Speculation fences are a processor's primary mechanism to override speculative execution. For Spectre variant-1, context-sensitive fencing 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. We first study the performance and security impact of the existing suite of fences and serializing instructions, and then explore and propose new speculation fences and context-sensitive fencing techniques that manage the impact on performance. For Spectre variant-2, CSF works

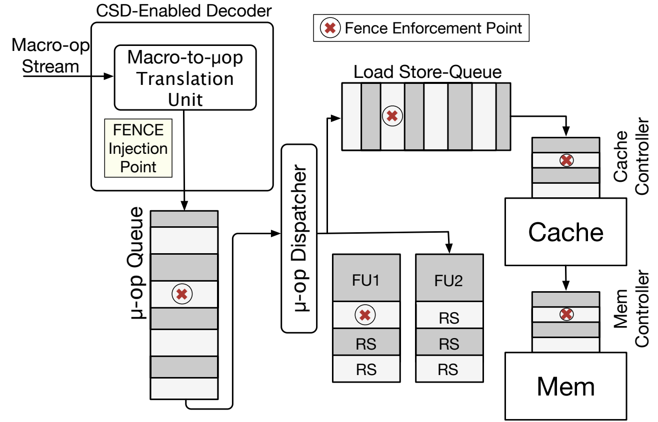

Figure 4. Fence Enforcement Points

with CSD by providing alternate decodings of some control flow, and possibly insert fences and/or branch predictor resetting micro-ops – more detail is in Section 5.3.

5.1.1 Serializing Instructions and Memory Fences.

Serializing instructions are the strictest amongst 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 in execution after the fence are stalled until all the outstanding memory requests (including the fence) complete execution.

Table 2 shows a complete list of Intel's existing serializing instructions and fences [48]. Most of the serializing instructions need to be run in privileged mode which restricts their usage in defenses for victims that lack sufficient privileges (e.g., browsers). Moreover, a majority of them modify the state or contents of architectural registers, the program counter, cache, TLB, etc. and only serialize execution as a side effect, thereby requiring an additional backup/restore step when used for the sole purpose of serializing execution. An exception to this is the *MOV to debug register* instruction that does not corrupt any architectural state, if not actually in debug mode.

The SFENCE instruction does not allow stores to pass through it, but does not affect loads. That is, it enforces that all stores that precede the SFENCE are executed to completion before any store that succeeds the SFENCE is fetched. The MFENCE instruction restricts all memory operations from passing through it. Unlike SFENCE and MFENCE that are memory ordering operations, LFENCE performs 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 front-end, it has been recommended by Intel as a low-overhead fence that can be inserted at vulnerable points in execution to defend against Spectre [45]. In other words, while serializing instructions such as CPUID are enforced at Fetch, LFENCE is enforced at the instruction queue level. Therefore, in this work, we start with the LFENCE, and propose new fences that come with fewer restrictions, different enforcement policies, and/or sport additional optimizations.

5.1.2 Fence Enforcement Policies

It is important to note that none of the existing instructions that provide fence support were actually created for the purpose we (or the whole industry) need for Spectre mitigation. Thus, this work will examine existing fences, variants of existing fences, and also introduce a new fence primitive. To better understand the design landscape, we examine several possible properties of fences in this section.

Early vs. 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, in essence, shifts the fence enforcement point towards 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 4 shows potential fence enforcement points at various stages in the processor pipeline.

It is important to note that 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 channel, we would have to resort to a regular serializing instruction or an MFENCE, resulting in prohibitively high 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. Instead, the FPU-based side-channel may be mitigated using a different fence that is appropriately configured and enforced at the reservation station or the FPU.

Strict vs. 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 SFENCE that is enforced at the load/store queue and allows all instructions to pass except stores that have a greater sequence number than itself, is a *late-enforced* and *relaxed* fence. On the other hand, all x86 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.

Table 3. 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

* CC: Cache Controller, RS: Reservation Station

Early vs. Late Commit. A fence typically remains effective until it gets committed or squashed. Based on Intel's manual [47], 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 aren't 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 enough 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 considerably improve performance because as soon as a fence gets committed, a stream of instructions can advance further in the pipeline. Therefore, in this work, we propose and study the effects of a late-commit version of each fence that does not wait for stores to be written back. However, we note that these versions should only be limited to the security use case we describe and should not be used for synchronization.

Newly Proposed Fences. Table 3 summarizes the characteristics of different existing and newly proposed fences. Although the existing set of fences defend against all variants, they incur prohibitively high costs on performance, due to their strict enforcement constraints. To better understand the potential for fences beyond those that already exist, we propose and evaluate three new types of fences. In an attack scenario, we typically insert one of these fences between a branch instruction and a load instruction that potentially leaks sensitive information via a cache side-channel. The proposed fences more effectively and efficiently mitigate the

high impact variant-1 attacks, by preventing information leaks along misspeculated paths through cache-based side channels. 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 fence 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. In addition to mitigating the variant-1 attack that the LSQ-LFENCE protects against, the LSQ-MFENCE mitigates the variant-1.1 and variant-1.2 attacks that exploit store-to-load forwarding between speculative loads and stores.

The CFENCE is a relaxed fence and is enforced at the cache controller level using the following set of rules. First, like any other fence, it 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 *non-modifying load* and allows it to pass through the fence, but the load is restricted from modifying the cache state. In particular, a *non-modifying 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 *non-modifying 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 don't leave any observable cache footprints along misspeculated paths, thereby mitigating the variant Spectre variant-1 attack. In addition, due to locality of references, the miss rate of a reasonable program is typically very low, and therefore, using CFENCE results in considerably lower performance overhead than other types of fences.

Note that this can be applied recursively at each cache level. For example, an L2 hit (L1 miss) will bypass the L1 cache and not initiate a fill, then read from the L2 cache without altering the LRU bits.

5.1.3 Fence Frequency Optimization

The most naïve yet secure way to insert fences is to liberally instrument every instruction of a vulnerable type (e.g., load instructions in the case of cache side-channels) in the program and add the fence micro-op to all of them. In fact not every instance of a vulnerable instruction type is necessarily vulnerable; for example, all the loads in the program aren't vulnerable to speculative attacks via cache side-channels. Therefore, we can reduce the number of fences inserted. However, since failing to insert fences, even in one scenario, would enable a Spectre attacker to read the whole victim's memory space, 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 (provide the alternate decoding for) the first instance of a vulnerable instruction type (e.g., first load) of each basic block, and it is safe to leave the rest of the instructions uninstrumented. This is simply implemented by setting a flag in hardware whenever a branch is decoded, 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 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 leads to the information leak. For example, in Spectre variant-1 the attacker provides an untrusted out-of-bound index to an array. In this work, we assume any information that comes from the user address space and input devices (e.g., via the x86 IN instruction) as untrusted. In addition, we also consider DMA'd pages as untrusted, for which we rely on the IOMMU [5] 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 a novel decoder-level information flow tracking (DLIFT) strategy described in the following section.

5.2 Decoder-Level Information Flow Tracking

As its name indicates, the distinguishing feature that makes Decoder-Level Information Flow Tracking (DLIFT) unique from typical hardware taint tracking systems is that it can provide the taint information at the decoder stage of the pipeline rather than at commit stage. This is a critical distinction for two reasons. First, of course, our solution is decoder-based. Second, and perhaps more important, execute- or commit-based taint tracking comes too late in the pipeline for any speculation-based attack, and is of little use.

However, commit is still the only stage where taint information is guaranteed to be correct, so the design of a decode-based taint tracker is challenging. In particular, since the front-end of the pipeline typically runs far ahead in execution than the rest of the pipeline, reading actual taints from register files at the decode stage will read inaccurate values. Therefore, in the proposed DLIFT framework, we

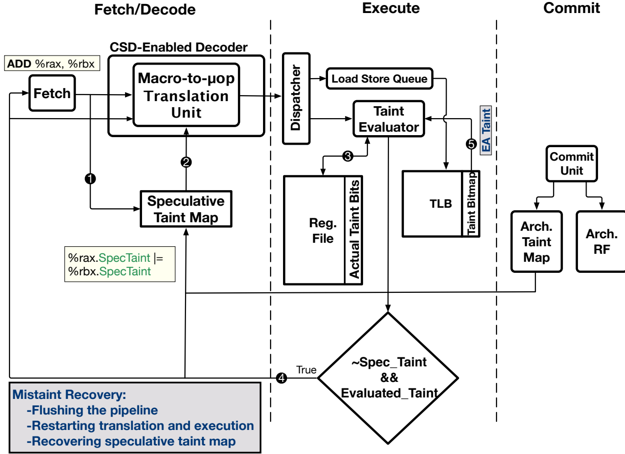

Figure 5. DLIFT integration with a CSD-enabled pipeline

separate the taint information into four taint structures – (a) a decoder-level taint map that tracks and maintains the taint information for architectural registers, (b) the physical register file augmented with taint information that maintains dynamically computed taint information at execute, (c) the TLB and page tables augmented with a taint bitmap to track cache block-level taint information, and (d) a commit-level taint map that maintains verified architectural register taint information. While the latter three structures are typical of any standard DIFT implementation [79], the first structure is a new addition proposed by this work. Instruction translation thus depends on some speculative taint tracking that might not be exact, potentially leaving vulnerable instructions unfenced (i.e., translating to non-secure micro-ops). For this reason, DLIFT relies on a mistaint recovery mechanism that, upon detecting a mistaint at the execution stage, redirects and restarts the execution from the incorrectly tainted instruction.

Figure 5 shows the integration of DLIFT with a CSD-enabled pipeline. The DLIFT engine maintains a taint map that stores a speculative taint bit for each architectural register. For each incoming instruction (❶), the DLIFT engine evaluates the taint of the destination register(s), based on the speculative taint bits of the source registers. Taint evaluation follows the standard DIFT procedure [79]. If a source register is marked as tainted in the speculative taint map, DLIFT propagates this taint to the destination register and further triggers context-sensitive translation (❷) of the instruction (e.g., with speculative fence insertion), depending upon the configured levels of performance and security.

Tracking taints at the decoder level is not always straightforward. First, during speculative execution, the DLIFT engine continues to propagate taints along the misspeculated path, potentially leaving the taint map in an inconsistent state upon recovery from a branch misprediction. To this end, the DLIFT engine rolls back the state of the taint map to its commit-level counterpart, as part of the branch misprediction recovery. Second, the effective addresses of load instructions is usually not known at the decoder stage, due to which the DLIFT engine cannot accurately compute and propagate taint information. A conservative approach that marks all loads with unknown effective addresses as tainted could severely degrade performance since overtainting typically results in a high frequency of fence insertion. Therefore, we take a more optimistic approach by assuming loads with unknown effective addresses are untainted, and further rely on our mistaint detection mechanism at the execute stage to validate the predicted taints against the dynamically evaluated taints (❸ and ❺). In the event a mistaint is detected, we update the speculative taint map, flush the pipeline, and restart the translation and execution of the mistainted instruction. Note that we only perform mistaint recovery for the undertainting scenario, since it could potentially leave vulnerable instructions unfenced.

Furthermore, the DLIFT engine is capable of following instruction sequences and execution patterns such as a tainted load followed by a branch, allowing for the seamless detection of different Spectre gadget variants. In this way, although speculative, the DLIFT engine has the potential for fast recovery from misspeculation, and further allows the microcode customization framework to perform a more targeted and surgical insertion of fences for particular vulnerable targets.

5.3 Mitigations for Control-flow Mistraining

Two Spectre variants (v2 and v5) rely on the mistraining of the branch predictor and the return stack buffer to influence the victim's branch outcome across different protection domains. To mitigate these attacks, we examine several hardening mechanisms. In the simplest case, where we are protecting kernel branches from getting influenced, we can instrument syscall instructions to enforce branch predictor state isolation. If we want to protect a wider range of domain crossings, we could rely on a simple hardware mechanism that identifies control transfer to a domain with a higher privilege and then sets a flag. The first control flow instruction that decodes after that flag is set would then trigger an alternate decoding and reset the flag. The alternate decoding would insert a fence (to protect against this branch being mispredicted), then execute micro-ops that enforce branch predictor state isolation, and subsequently resume the original control flow.

This solution, then, is flexible enough that any region of code could be protected in this way by configuring model specific range registers (MSRRs) to indicate a protected region that would always reset the branch predictor upon entry; this, then, even prevents mistraining of sensitive code of different threads within the same process. To prevent the MSRRs from being tampered by the attacker, we only allow

Table 4. Architecture Detail for the Baseline x86 Core

Baseline Processor			
Frequency	3.3 GHz	I cache	32 KB, 8 way
Fetch width	4 fused uops	D cache	32 KB, 8 way
Issue width	6 unfused uops	ROB size	168 entries
INT/FP Regfile	160/144 regs	IQ	54 entries
RAS size	8,16, 32 entries	BTB size	256, 512, 1024 entries
LQ/SQ size	64/36 entries	Functional	Int ALU(6), Mult(1),
Branch Predictor	LTAGE	Units	FP ALU/Mult(2), SIMD(2)

the operating system to configure MSRRs, and we use speculative fences to guard x86's *WRMSR* instruction. Furthermore, in the case of remapping via mmap(), the operating system is also responsible for reconfiguring MSRRs.

The easiest and the most heavyweight way to isolate the BP state is to clear all the states by a series of return/branch micro-ops. However, such a hard reset of the BP can be more quickly and efficiently performed using a special micro-op that clears some or all state of the branch predictor, thereby enabling custom solutions to reset different structures within the prediction unit, if such a micro-op is available. In this work, we assume a special micro-op to clear the branch target buffer (BTB) and the return stack buffer (RSB), the primary targets of the Spectre attack.

It should be noted that while resetting the BTB and the RSB will induce more mispredictions (something the attacker wants), it does the attacker no good with respect to these Spectre variants because the attacker can no longer control the target of the control flow misspeculation.

6 Methodology

This section details the experimental methodology for the performance and security evaluation of the *context-sensitive fencing* framework.

Our baseline processor is modeled after the Intel Sandybridge microarchitecture [43]. Table 4 shows the architectural configuration of our baseline processor in more detail. Note that we evaluate against three different branch predictor configurations (with different BTB and RAS sizes) to measure the performance impact of our micro-op flows that perform branch predictor state isolation across protection domains to defend against variant-2. We model this architecture using the gem5 [8] architectural simulator which already features x86 micro-op translation. Furthermore, gem5 already features a Spectre test infrastructure and a visualization tool that allow us to validate our claims regarding the security guarantees of context-sensitive fencing [63].

One of the primary goals of this work is to protect kernel memory from being leaked along misspeculated paths. Therefore, we use the full system simulation mode of gem5 which allows us to boot an Ubuntu 18.04 distribution of Linux with a kernel version of 4.8.13. Furthermore, in order to provide realistic estimates regarding attack coverage and performance impact of our proposed techniques, we select a good mix of benchmarks that spend different amounts of execution time in kernel mode, touch different aspects of the operating system, and perform different number of system calls during execution, as illustrated in Table 5. To this end, we include four benchmarks, *bzip2*, *gcc*, *sjeng*, *omnet*, from the SPEC CPU2006 suite [41] that exhibit varying degrees of instruction-level parallelism. Furthermore, we use three commonly used Unix tools that target different functionalities of the kernel and use different device drivers – (*ping*) that sends and receives ICMP packets, (*ps aux*) that queries process information, and (*ls /*/**) that queries the filesystem in order to list multi-level directory information. In addition, we also add a memory-intensive program *llu* that allocates a large amount of memory for a linked list and traverses the list making random memory accesses [103]. Finally, we also evaluate the impact of our fences on the *nginx* web server [74] using the *wrk* [35] framework to generate HTTP requests.

Table 5. Benchmarks Description

Benchmark	Description	Kernel-Time
nginx	HTTP Web Server	66%
ps	Process information query	75%
ping	Sends ICMP ECHO_REQUEST to loopback	95%
ls	Performs two level directory listing	79%
llu	Linked list traversal micro-benchmark	7%
bzip2	Compression	34%
gcc	C Language optimizing compiler	11%
omnetpp	Discrete event simulation	15%
sjeng	Artificial Intelligence (game tree search and pattern recognition)	22%

7 Evaluation

In this section, we evaluate both security and performance impacts of the proposed strategy, starting first with security assessment and following that with performance evaluation.

7.1 Security Discussion

The proposed defense strategy can mitigate five variants of the Spectre attacks. We discuss these in detail below.

Variants 1, 1.1, and 1.2. The goal of these variants is to leak memory contents of a victim along a misspseculated path by bypassing a bounds check and/or further stitch together such Spectre gadgets via store-to-load forwarding. All variants exploit a data cache-based side channel to leak information. Context-Sensitive Fencing mitigates these variants primarily by preventing information from being leaked along misspeculated paths via fences that are surgically inserted into the instruction stream and strategically placed between the conditional branch that performs the bounds check and the first load in the Spectre gadget. This work proposes three new fences – LSQ-LFENCE, LSQ-MFENCE, and CFENCE that each defend against the variant-1 attack by disallowing loads along misspeculated paths to modify the data cache state. LSQ-MFENCE additionally protects against the variants 1.1 and 1.2 by preventing both speculative loads and stores from being issued out of the load/store queue,

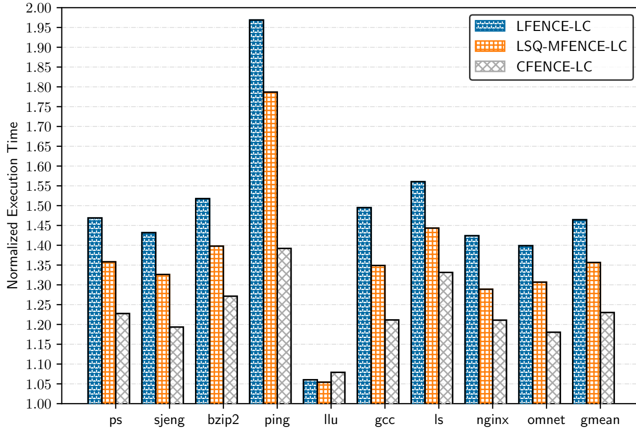

Figure 6. Execution Time of Different Fence Enforcement Levels (normalized to insecure execution)

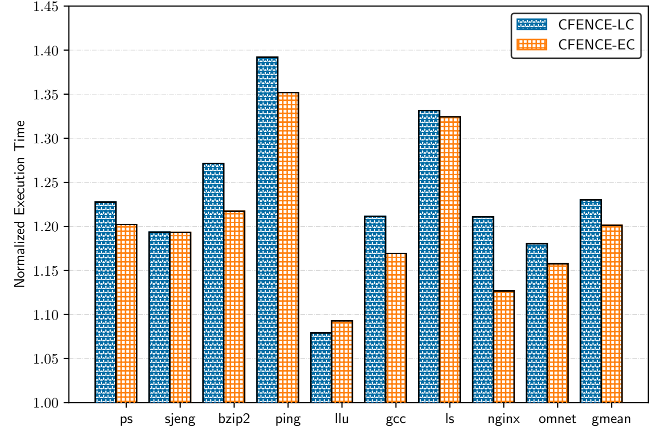

Figure 7. Execution Time of Early and Late Commit of CFENCE (normalized to insecure execution)

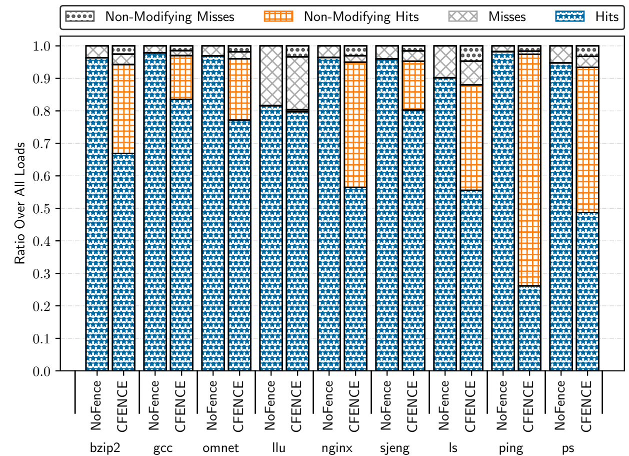

Figure 8. Effects of injecting CFENCE on Cache Miss Rate

effectively avoiding store-to-load forwarding between speculative loads and stores.

The surgical insertion of these fences is facilitated by the decoder-level information flow tracker (DLIFT) that can follow instruction sequences that could serve as Spectre gadgets, as described in Section 5.2. While the DLIFT itself performs speculative tracking of taint information, it has the ability to detect mistaints and recover within three stages of the pipeline. We further experimentally show later in this section that the speculative nature of the DLIFT engine may sometimes result in overtainting (where trusted operands/instructions get marked as tainted), but never results in an undetected and unrecovered undertainting (where untrusted operands/instructions remain untainted).

Variants 2 and 5. These variants reverse-engineer and mistrain the branch predictor and the return address stack to influence the branch outcomes of a victim, which in our case is the Linux kernel. In addition to the surgical fence injection and speculative taint tracking, context-sensitive fencing further employs mistraining mitigations that circumvent such attempts. More specifically, we intercept all protection domain crossings via *syscall* and *int 0x80* instructions and perform a full reset of the branch target buffer (BTB) and the return stack buffer (RSB), clearing all previous state. This prevents user code from influencing branch outcomes of the kernel code. We record no instance of BTB collision between user and kernel branches, and we ensure that we always start with a clean BTB and RSB upon entering kernel mode.

To show that our different fences and frequency optimizations close Spectre attacks, we use a proof of concept Spectre implementation and visualization tool [63]. The attacks fail in every case when we use context-sensitive fencing to insert fences, despite our performance optimizations.

7.2 Performance

Figure 6 measures the performance impact of three different fences – (a) the standard x86 LFENCE, (b) the LSQ-MFENCE, and (c) the CFENCE, all enforced with the standard late commit, pessimistically inserted for every kernel load in the program. Clearly, from the figure, the CFENCE incurs the lowest performance overhead, since it is less restrictive than the other two and is enforced at a much later pipeline stage. We further study the effect of late and early commit policies for the CFENCE, as shown in Figure 7. The early commit version of the CFENCE consistently performs better than the late commit version, saving about 4% in overall execution time on average.

Overall, the CFENCE reduces the incurred performance overhead due to fencing by 2.3X, bringing down the execution time overhead from 48% to 21%. Furthermore, this performance improvement is consistent across all the benchmarks; the only exception being *llu*, which performs random memory accesses due to the linked list traversal and suffers from a very high cache miss rate.

In Figure 8, we study the effect of the CFENCE on cache miss rate. Recall that when a load passes a CFENCE, it gets marked as *non-modifying*, and as a result, if it ends up being a miss in the cache, it is deemed *uncacheable* and therefore

the fetched cache block isn't brought into cache. If the miss rate of a program is already high, as in the case of *llu*, the presence of a CFENCE in the instruction stream could potentially result in under-utilization of the cache since the missed blocks aren't being filled back into the cache while the CFENCE is begin enforced. For programs that have unusually low hit rates, we suggest using the standard LFENCE instead of the CFENCE. However, in those cases it should be noted that the overhead of the mitigation is low regardless of the fence used.

Furthermore, it is important to note that the number of non-modifying loads in the dynamic instruction stream do not necessarily have a negative impact on performance, while a CFENCE is being enforced. For example, while the *ping* program has the most non-modifying loads, it does not suffer much in terms of overall performance, because the working set fits well into the cache and most of the non-modifying accesses end up being hits. In general, we gain more from employing the CFENCE instead of standard serializing fences when the cache hit rate is low.

Figure 9 shows the performance of our proposed fence frequency optimization techniques. In the first optimization, we inject the CFENCE only for tainted loads and branches, 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 block-level fence insertion, where we only instrument the first instance of a vulnerable load in each basic block. This results in an additional 4% improvement in performance. As Figure 9 shows the once per basic block opitimization is successful at improving the performance of all the benchmarks except *llu*. That is because CFENCE changes the cache access pattern, usually at some cost, but occasionally beneficially by bypassing accesses and reducing pressure on the cache. In such a case, like *llu* with its large working set, executing fewer fences would then be slightly less beneficial. Overall, compared to the state-of-the-art fence injection scheme that pessimistically injects an LFENCE for all kernel loads, our DLIFT-Based CFENCE injection reduces the performance overhead from 48% to just 7.7% on average.

We next study the accuracy and coverage of our DLIFT implementation. Figure 10 examines overtainting scenarios, where the DLIFT engine conservatively marks instructions as tainted when they're actually not. The results are normalized to the total number of dynamic load instructions executed. Further, we examine two scenarios – the bars to the left represent DLIFT-based fencing without the basic-block level fence insertion, and the bars to the right includes the basic-block level fence insertion optimization. In most cases, the percentage of overtainted loads remains low in both scenarios.

Figure 11 examines undertainting scenarios, where the DLIFT engine optimistically forgoes instrumenting certain instructions that are actually vulnerable, but recovers from

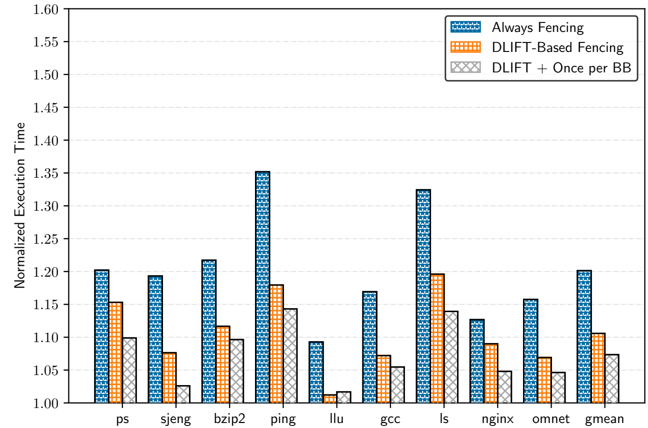

Figure 9. Impact of Fence Frequency Optimizations on Execution Time

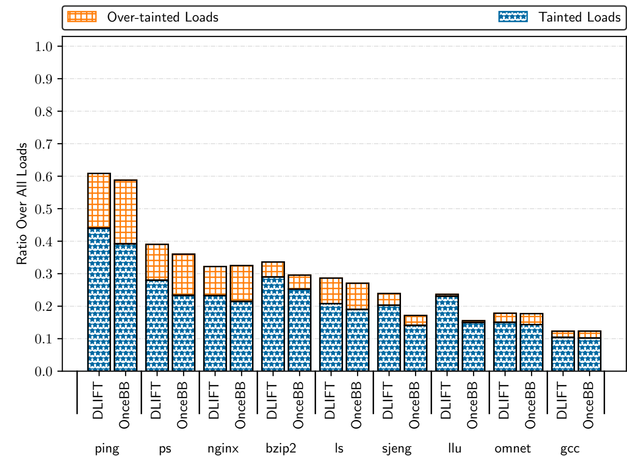

Figure 10. Accuracy of DLIFT: Overtainting Rate of Loads

such scenarios later in the pipeline. We observe that the percentage of undertainted loads is low and consistently below 10%, with the outliers being the kernel-heavy applications, *ping* and *ps*, which have the highest number of tainted instructions (instructions with at least one tainted source register) as shown in Figure 12.

Note that our DLIFT engine performs explicit information flow tracking modeled after DIFT [79] that tracks copy, load-address, store-address, and computational dependencies. However, we also evaluate CSF with a more conservative implicit information flow tracking model in which we taint the program counter when the branch outcome depends upon a tainted value, and further track both sides of the branch. This results in more tainted instructions and incurs 11.8% extra performance overhead on top of our taint-based CFENCE insertion.

To evaluate the effect of our mistraining mitigations, we measure the impact of a full micro-op based BTB and RSB reset at every protection domain crossing. We do this experiment on three separate branch predictor configurations: (a) a small predictor with 256 BTB and 8 RAS entries, (b) a

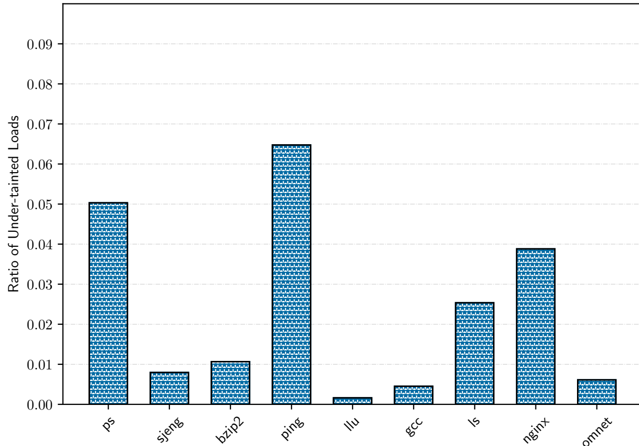

Figure 11. Coverage of DLIFT: Undertainting Rate of Loads

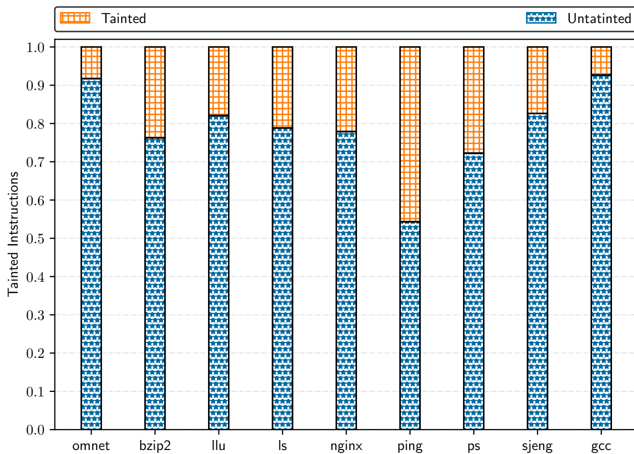

Figure 12. Ratio of instructions marked as tainted by DLIFT

medium predictor with 512 BTB and 16 RAS entries, and (c) a large predictor with 1024 BTB and 32 RAS entries. We measure an average performance degradation of 2.7% on our small predictor, 6.6% on our medium predictor, and 15.2% on our large predictor. Naturally, the overhead is almost completely due to the loss of prediction accuracy, rather than the cost of micro-op expansion.

In summary, the proposed defense strategy introduces a flexible microcode customization framework that perform the surgical insertion of newly introduced speculation fences, that mitigate five variants of the Spectre class of attacks, reducing the fencing overhead of state-of-the-art fence-based Spectre mitigations by a factor of 6.

8 Conclusion

In this work, we propose context-sensitive fencing (CSF), a set of architectural techniques that provide high-performance defense against Spectre-style attacks. In particular, 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, in the decoder, with no recompilation and binary translation required. This allows us to employ runtime information to strategically insert fences when needed, using taint information and avoiding redundant fences after branches. 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.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF Grant CNS-1652925, NSF/Intel Foundational Microarchitecture Research Grant CCF-1823444, and DARPA under the agreement number HR0011-18-C-0020.

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *CryptographersâĂŹ Track at the RSA Conference*.

[2] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. 2007. Cache based remote timing attack on the AES. In *CryptographersâĂŹ Track at the RSA Conference*.

[3] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 94–106. https://doi.org/10.1145/3079856.3080232

[4] AMD. 2018. *White paper: SOFTWARE TECHNIQUES FOR MANAGING SPECULATION ON AMD PROCESSORS*. Technical Report REVISION 1.24.18. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

[5] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU Emulation. In *USENIX Annual Technical Conference*.

[6] ARM. 2018. *Whitepaper: Cache Speculation Side-channels*. Technical Report Version 2.2. https://developer.arm.com/-/media/developer/pdf/Security%20update%2010%20July%2018/Cache$_S$peculation$_S$ide-channels-v2.2.pdf?revision=7de26366-a49f-4c23-85f0-34e8f5e38881.

[7] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*.

[8] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. *Micro, IEEE* (2006).

[9] Darrell Boggs, Gary Brown, Nathan Tuck, and K. S. Venkatraman. 2015. Denver: Nvidia's First 64-bit ARM Processor. *IEEE Micro* (2015).

[10] Chandler Carruth. 2018. Mitigating Speculative Attacks in Crypto. https://github.com/HACS-workshop/spectre-mitigations/blob/master/crypto$_g$uidelines.md. Online; accessed Jul 2018.

[11] Chandler Carruth. 2018. RFC: Speculative Load Hardening (a Spectre variant1 mitigation). https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT$_6$1e$_K$o3TmoCS3uXLcJR0/edit. Online; accessed Jul 2018.

[12] Chandler Carruth. 2018. RFC: Speculative Load Hardening (a Spectre variant1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html. Online; accessed Jul 2018.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave

Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv:1802.09085 http://arxiv.org/abs/1802.09085

[14] H. Chen, X. Wu, L. Yuan, B. Zang, P. c. Yew, and F. T. Chong. 2008. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *2008 International Symposium on Computer Architecture*. 401–412. https://doi.org/10.1109/ISCA.2008.18

[15] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, and Todd C. Mowry. 2011. Log-based Architectures: Using Multicore to Help Software Behave Correctly. *SIGOPS Oper. Syst. Rev.* 45, 1 (Feb. 2011), 84–91. https://doi.org/10.1145/1945023.1945034

[16] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Software-based Gate-level Information Flow Security for IoT Systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 328–340. https://doi.org/10.1145/3123939.3123955

[17] Jonathan Corbet. 2017. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/. Online; accessed Jan 2018.

[18] Marc L Corliss, E Christopher Lewis, and Amir Roth. 2003. DISE: A programmable macro engine for customizing applications. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*.

[19] Marc L Corliss, E Christopher Lewis, and Amir Roth. 2005. Low-overhead interactive debugging via dynamic instrumentation with DISE. In *11th International Symposium on High-Performance Computer Architecture*.

[20] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. 2005. Using DISE to Protect Return Addresses from Attack. *SIGARCH Comput. Archit. News* (March 2005).

[21] Jedidiah R. Crandall and Frederic T. Chong. 2004. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, Washington, DC, USA, 221–232. https://doi.org/10.1109/MICRO.2004.26

[22] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 81–96. https://doi.org/10.1145/2541940.2541986

[23] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 351–366. https://doi.org/10.1145/1294261.1294295

[24] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 482–493. https://doi.org/10.1145/1250662.1250722

[25] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*.

[26] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. 2012. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[27] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 51–67. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[28] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. 2018. Spectres, Virtual Ghosts, and Hardware Support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18)*. ACM, New York, NY, USA, Article 5, 9 pages. https://doi.org/10.1145/3214292.3214297

[29] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. 2018. Shielding Software From Privileged Side-Channel Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1441–1458. https://www.usenix.org/conference/usenixsecurity18/presentation/dong

[30] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204

[31] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic analysis: Concrete results. In *International Workshop on Cryptographic Hardware and Embedded Systems*.

[32] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[33] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*.

[34] Daniel Genkin, Adi Shamir, and Eran Tromer. 2016. Acoustic Cryptanalysis. *Journal of Cryptology* (2016).

[35] Github. [n. d.]. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. Online; accessed Jun 2018.

[36] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. Springer, 161–176.

[37] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[38] Maurice Clémentine Wagner Klaus Gruss, Daniel and Stefan Mangard. 2016. *Flush+Flush: A Fast and Stealthy Cache Attack.*

[39] V. Haldar, D. Chandra, and M. Franz. 2005. Dynamic taint propagation for Java. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. 9 pp.–311. https://doi.org/10.1109/CSAC.2005.21

[40] W. Halfond, A. Orso, and P. Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering* 34, 1 (Jan 2008), 65–81. https://doi.org/10.1109/TSE.2007.70748

[41] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* (Sept. 2006).

[42] Jonathan J Hoch and Adi Shamir. 2004. Fault analysis of stream ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*.

[43] Intel. 2008. *2nd Generation Intel Core vPro Processor Family*. Technical Report. http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf

[44] Intel. 2014. Software guard extensions programming reference. (2014). "https://software.intel.com/sites/default/files/329298-001.pdf"

[45] Intel. 2018. *White paper: Intel Analysis of Speculative Execution Side Channels*. Technical Report 336983-001, Revision 1.0. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[46] Intel. 2018. *White paper: Retpoline: A Branch Target Injection Mitigation*. Technical Report 337131-003, Revision 003. https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.

[47] Intel Corporation. 2009. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[48] Intel Corporation 2011. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

[49] Project Zero Jann Horn. 2018. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. Online; accessed Jan 2018.

[50] Morteza Mohajjel Kafshdooz, Mohammadkazem Taram, Sepehr Assadi, and Alireza Ejlali. 2016. A Compile-Time Optimization Method for WCET Reduction in Real-Time Embedded Systems Through Block Formation. *ACM Trans. Archit. Code Optim.* 12, 4, Article 66 (Jan. 2016), 25 pages. https://doi.org/10.1145/2845083

[51] R. Karri, K. Wu, P. Mishra, and Yongkook Kim. 2002. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21, 12 (Dec 2002), 1509–1517. https://doi.org/10.1109/TCAD.2002.804378

[52] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2017. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3061639.3062313

[53] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2018. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv preprint arXiv:1806.05179* (2018).

[54] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[55] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv preprint arXiv:1807.03757* (2018).

[56] Paul Kocher. 2018. Spectre Mitigations in Microsoft's C/C++ Compiler . https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html. Online; accessed Jul 2018.

[57] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[58] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh

[59] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh

[60] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. 2008. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '08)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1328408.1328410

[61] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[62] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[63] Jason Lowe-Power. 2018. Visualizing Spectre with gem5. http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html. Online; accessed Jun 2018.

[64] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. 2018. Position Paper: A Case for Exposing Extra-architectural State in the ISA. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18)*. ACM, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/3214292.3214300

[65] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *MICRO*.

[66] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

[67] Microsoft Security Response Center (MSRC) Matt Miller. 2018. Analysis and mitigation of speculative store bypass (CVE-2018-3639). https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/. Online; accessed Jul 2018.

[68] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. 2012. Are AES x86 cache timing attacks still feasible?. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*.

[69] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *CoRR* abs/1805.08506 (2018). arXiv:1805.08506 http://arxiv.org/abs/1805.08506

[70] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*.

[71] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. 2011. Harmonia: A Transparent, Efficient, and Harmonious Dynamic Binary Translator Targeting the Intel&Reg; Architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF '11)*. ACM, New York, NY, USA, Article 26, 10 pages. https://doi.org/10.1145/2016604.2016635

[72] Andrew Pardoe. 2018. Spectre mitigations in MSVC . https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/. Online; accessed Jul 2018.

[73] Joseph CH Park and Mike Schlansker. 1991. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California.

[74] Will Reese. 2008. Nginx: The High-performance Web Server and Reverse Proxy. *Linux J.* 2008, 173, Article 2 (Sept. 2008). http://dl.acm.org/citation.cfm?id=1412202.1412204

[75] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* (2012).

[76] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read Arbitrary Memory over Network. *arXiv preprint arXiv:1807.10535* (2018).

[77] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[78] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480* (2018).

[79] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[80] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2018. Mobilizing the Micro-Ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[81] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/1508244.1508258

[82] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *CoRR* abs/1802.03802 (2018). arXiv:1802.03802 http://arxiv.org/abs/1802.03802

[83] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ACM SIGARCH Computer Architecture News*, Vol. 24. ACM, 191–202.

[84] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 392–403. https://doi.org/10.1145/223982.224449

[85] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. Online; accessed Jul 2018.

[86] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on.* 243–254. https://doi.org/10.1109/MICRO.2004.31

[87] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association. See also technical report Foreshadow-NG [95].

[88] Ashish Venkat. 2018. *Breaking the ISA Barrier in Modern Computing*. Ph.D. Dissertation. UC San Diego.

[89] Ashish Venkat, Arvind Krishnaswamy, Koichi Yamada, and Rajan Palanivel. 2015. Binary Translation driven Program State Relocation. In *United States Patent Grant US009135435B2*.

[90] Ashish Venkat, S Shamasunder, Hovav Shacham, and Dean M. Tullsen. 2016. HIPStR: Heterogeneous-ISA Program State Relocation. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*.

[91] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*.

[92] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 173–184. https://doi.org/10.1109/HPCA.2008.4658637

[93] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.. In *NDSS*, Vol. 2007. 12.

[94] Zhenghong Wang and Ruby B Lee. 2006. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 473–482.

[95] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018). See also USENIX Security paper Foreshadow [87].

[96] Troy Wolverton. 2018. Spectre and Meltdown are now a legal pain for Intel, the chip maker faces 35 lawsuits over the attacks. https://www.businessinsider.com/35-lawsuits-have-been-filed-against-intel-over-spectre-and-meltdown-2018-2. Online; accessed Aug 2018.

[97] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 159–173. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu

[98] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks.. In *USENIX Security Symposium*. 121–136.

[99] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 428–441. https://doi.org/10.1109/MICRO.2018.00042

[100] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security*.

[101] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A timing attack on OpenSSL constant time RSA. In *International Conference on Cryptographic Hardware and Embedded Systems*.

[102] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1629575.1629604

[103] Craig Zilles. 2001. Linked List Traversal Micro-Benchmark. http://zilles.cs.illinois.edu/llubenchmark.html. Online; accessed Jun 2018.