Performance Evolution of Mitigating Transient Execution Attacks

Jonathan Behrens
behrensj@mit.edu
MIT CSAIL
Cambridge, Massachusetts, USA

Adam Belay
abelay@mit.edu
MIT CSAIL
Cambridge, Massachusetts, USA

M. Frans Kaashoek
kaashoek@mit.edu
MIT CSAIL
Cambridge, Massachusetts, USA

Abstract

Today's applications pay a performance penalty for mitigations to protect against transient execution attacks such as Meltdown [32] and Spectre [25]. Such a reduction in performance directly translates to higher operating costs and degraded user experience. This paper measures the performance impact of these mitigations across a range of processors from multiple vendors and across several security boundaries to identify trends over successive generations of processors and to attribute how much of the overall slowdown is caused by each individual mitigation.

We find that overheads for operating system intensive workloads have declined by as much as 10×, down to about 3% on modern CPUs, due to hardware changes that eliminate the need for the most expensive mitigations. Meanwhile, a JavaScript benchmark reveals approximately 20% overhead persists today because mitigations for Spectre V1 and Speculative Store Bypass have not become more efficient. Other workloads like virtual machines and single-process, compute-intensive applications did not show significant slowdowns on any of the processors we measured.

CCS Concepts: • **Security and privacy** → **Side-channel analysis and countermeasures**; **Systems security**.

Keywords: Transient execution attack, speculative execution, Spectre, Meltdown

ACM Reference Format:
Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. 2022. Performance Evolution of Mitigating Transient Execution Attacks . In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519559

1 Introduction

Transient execution attacks are a class of side channel attacks that leak information across privilege domains using a processor's microarchitectural details. Spectre and Meltdown were the first—but far from only—of these attacks to be discovered.

From the end-user perspective, a significant concern from transient execution attacks is the performance degradation they cause. This is because operating systems and applications have deployed mitigations to restore their previous security guarantees, but those same mitigations make systems slower. This highly-visible impact on user experience has been measured by Phoronix and others [31, 38, 44]. This paper goes further and attributes overheads to individual mitigations to understand which ones matter to overall performance. The paper also studies internal JavaScript runtime mitigations to understand whether the performance impact on browsers is different from operating systems.

We seek to answer the following questions: Which attacks are primarily responsible for the performance impact, and does that vary across processors or workloads? (§4) What drives the cost of mitigations for those attacks? (§5) What mitigations would benefit from hardware support to lower their cost and what predictions can we make about mitigation overheads going forward? (§7)

To answer these questions we first consider the end-to-end impact of transient execution attacks. Across a range of benchmarks, we characterize the overhead caused by mitigations on each CPU and further attribute how much of the overall slowdown is caused by each individual mitigation. This end-to-end evaluation guides our microbenchmarking of individual mitigations. For mitigations that incurs meaningful overhead, we investigate their performance in detail.

Our measurements focus on security boundaries because mitigations for transient execution attacks usually involve doing extra work for each boundary crossing, often in the form of flushing of microarchitectural state or waiting for in-flight operations to complete. Each of our workloads are chosen to stress a different security boundary. We measure the boundary between user mode and kernel mode, the boundaries between JavaScript sandboxes, and the boundary between a guest OS and a hypervisor. Furthermore, we confirm that mitigation overheads are low in the absence of security boundaries.

Our experiments cover a range of processors including both some that predate the discovery of Spectre and Meltdown—which are still in active use—and more recent ones which incorporate a range of new hardware-based mitigations. We evaluate five processors from Intel and three AMD processors, allowing us to compare across vendors as well.

The primary contribution of this paper is to draw attention to the performance critical areas for improving transient execution mitigations, driven by (1) an end-to-end survey of how mitigation costs have evolved over processor generations, and (2) detailed microbenchmarking of individual mitigations. To analyze hardware mitigations for Spectre V2, we also contribute a new technique to measure speculation using ideas from Bölük [7]. Our benchmarks and analysis code are available online at `github.com/mit-pdos/spectrebench`.

The primary conclusions of the study are as follows. Overhead in the OS boundary has mostly been eliminated, but the browser boundary is still expensive. A simple way to reduce overheads significantly without compromising security is to replace older CPUs with newer models. Most overheads that still exist are caused by a small number of software mitigations, all addressing attacks that were discovered in 2018 or earlier, while all attacks published since have mitigations with only minor performance impact. Hardware designers should focus on migrating these last few expensive software mitigations to silicon.

There are some limitations. Our benchmark workloads may not be representative of all applications. We consider several security boundaries but not all (e.g., we don't study the eBPF/-kernel boundary). We are limited in the number of processor generations we can evaluate and at the same time the processors we do consider are diverse in terms of clock speed, core count, power draw, and many other dimensions. The conclusions we draw are constrained by the lack of public details on how hardware mitigations are implemented. Finally, new attacks may require new mitigations with additional overheads, and existing mitigations may contain flaws, requiring fixes that could change their performance characteristics.

The rest of the paper is organized as follows. §2 relates this paper to previous work. §3 describes transient-execution attacks and their mitigations from a performance perspective. §4 measures the penalty of mitigations on several end-to-end workloads. §5 analyzes individual mitigations that have significant impact on end-to-end performance. §6 zooms in on Spectre V2 mitigations. §7 presents ideas for how to reduce the lingering performance impact of mitigations. §9 summarizes our conclusions.

2 Related Work

The mitigations this paper explores became necessary after the discovery of Meltdown [32] and the original Spectre [25] variants. These were rapidly followed by the discovery of more attacks targeting transient execution, including MDS [11, 43, 47], Speculative Store Bypass [18], and many others [6, 8, 9, 13, 26, 39, 45, 48, 51]. Several survey papers categorize the known attacks [10, 16, 52].

Many of these attacks have now been fixed in production hardware [20]. Other attacks require more substantial changes [1, 3, 50, 53, 54], which generally involve somehow delaying the use of speculative data until it is safe. Although such defenses are more comprehensive, they have higher overheads that impact performance whenever speculation occurs. And since an enormous number of older processors are still in active use, software defenses are also essential. For operating systems, these include KAISER [15] and retpolines [21].

These defenses have not always worked as well as hoped. Concurrently with this work, researchers discovered issues with several of the Spectre V2 mitigations [4, 34].

User-space sandboxing requires its own set of techniques. Swivel [36] is a compiler framework which hardens WASM bytecode against attack, while Firefox's and Chrome's WASM engines rely on Site Isolation [40]. Production JavaScript engines deploy more targeted mitigations like Pointer Poisoning and Index Masking [37], and also reduce the overall timer precision [37, 49]. Compiler techniques like Speculative Load Hardening [12] ensure binaries are completely immune to Spectre, albeit at considerable overhead.

Simakov [44] and Prout [38] conducted early performance studies on the impact of transient execution attacks, but the most comprehensive results come from the many articles published by Phoronix [28, 29, 31]. This prior work provides top-line numbers on the total overhead, but does not attribute costs to individual mitigations nor measure the impact of JavaScript level mitigations.

The Linux community has paid close attention to the cost of mitigations throughout, including for IBRS [46], KPTI [14], and MDS [28]. Their efforts has played a role in both understanding and driving down the performance overheads. Where this work stands out is by looking at many different generations of CPUs and characterizing the performance impacts of the set of mitigations deployed in practice.

3 Background: Attacks and Mitigations

We consider attacks from the the perspective of how they affect end-user performance. This outlook differs from prior surveys like Canella, et al. [10] which focus on enumerating and classifying the space of possible attacks.

3.1 Meltdown-Type Attacks

Meltdown-type attacks exploit the processor's fault-handling logic to speculatively access privileged state.

Meltdown [32]. The original Meltdown attack is caused by speculatively translating addresses for kernel pages even while running in user mode, which enables a user process to read any kernel memory mapped into its address space before the processor aborts speculation and raises a fault. At the time

Attack	Mitigation	Broadwell	Skylake Client	Cascade Lake	Ice Lake Client	Ice Lake Server	Zen	Zen 2	Zen 3
Meltdown	Page Table Isolation	✓	✓						
L1TF	PTE Inversion	✓	✓						
	Flush L1 Cache	✓	✓						
LazyFP	Always save FPU	✓	✓	✓	✓	✓	✓	✓	✓
Spectre V1	Index Masking	✓	✓	✓	✓	✓	✓	✓	✓
	lfence after swapgs	✓	✓	✓	✓	✓	✓	✓	✓
Spectre V2	Generic Retpoline	✓	✓						
	AMD Retpoline						✓	✓	✓
	IBRS								
	Enhanced IBRS			✓	✓	✓			
	RSB Stuffing	✓	✓	✓	✓	✓	✓	✓	✓
	IBPB	✓	✓	✓	✓	✓	✓	✓	✓
Spec. Store Bypass	SSBD	!	!	!	!	!	!	!	!
MDS	Flush CPU Buffers	✓	✓	✓					
	Disable SMT	!	!	!					

Table 1. Default mitigations used by Linux on each processor for our experiments. A ✓in a given cell means the mitigation used, while an empty space means it isn't required. In some cases, preventing an attack requires some mitigation that isn't enabled by default, which is indicated by a ! symbol.

of discovery, existing processors from Intel as well as some from IBM and ARM were vulnerable [2, 19, 23].

Software mitigations for Meltdown are expensive, requiring a page table switch on every user-kernel boundary crossing. Processors made by other vendors—and those designed after the attack was discovered—do not engage in this kind of speculation, so they can avoid these software overheads.

L1 Terminal Fault [51]. On certain Intel processors, the present bit in a page table entry (PTE) is ignored during speculative execution, which can allow an attacker to leak L1 cache contents. Operating system software can easily be adjusted to make sure no vulnerable page table entries are included in the page tables, which mitigates the attack at virtually zero cost.

However, when running a hypervisor, the same speculative mechanisms also bypass the nested page table. Taken together, if the hypervisor doesn't flush the L1 cache before every VM entry, it risks leaking recently accessed data from other privilege domains. Both the flush operation itself and subsequent cache misses make this mitigation more costly.

LazyFP [45]. Traditionally, when exposing floating point hardware to user processes, operating systems would optimize context switch time by lazily saving and restoring FPU state. In particular, the assumption was that many processes would not access floating point state, so on a context switch the FPU would be marked disabled, but retain the floating point registers from the previously running process. Any attempt to execute floating point instructions would trigger a trap, during which the OS could save the old process's floating point registers and load in the registers for the current process.

During transient execution, some processors will ignore the enable bit on the FPU and allow computation on the floating point registers even if they actually belong to a different process, potentially leaking sensitive register contents to an attacker.

Linux's mitigates LazyFP by always saving and restoring FPU state during context switches. Amusingly, this mitigation speeds up certain workloads, because modern processors provide fast instructions for saving and restoring this state (e.g., `xsaveopt`) [33]. As a result, the trap handling overhead is often higher than the cost of unconditionally saving and restoring the registers.

3.2 Spectre-Type Attacks

Spectre-style attacks exploit speculative execution following a misprediction. They typically target a gadget that performs two memory loads; see Figure 1 for an example. The first load brings sensitive data into a register, and the second uses that value to index into a large array. An attacker is later able to determine the result of the first load by measuring which array entry was pulled into the cache. This forms the backbone of various Spectre attacks.

Vendor	Model	Microarchitecture	Power (W)	Clock Speed (GHz)	Cores
Intel	E5-2640v4	Broadwell (2014)	90	2.4	10
	i7-6600U	Skylake Client (2015)	15	2.6	2
	Xeon Silver 4210R	Cascade Lake (2019)	100	2.4	10
	i5-10351G1	Ice Lake Client (2019)	15	1.0	4
	Xeon Gold 6354	Ice Lake Server (2021)	205	3.0	18
AMD	Ryzen 3 1200	Zen (2017)	65	3.1	4
	EPYC 7452	Zen 2 (2019)	155	2.35	32
	Ryzen 5 5600X	Zen 3 (2020)	65	3.7	6

Table 2. Information about each of the CPUs we evaluate. All except the Ryzen 3 1200 have 2-way SMT ("hyperthreads" in Intel terminology).

```
int x = array[index];
int y = array2[x * 256];
```

Figure 1. A Spectre gadget. If this code sequence is executed (even speculatively) it will alter the contents of the CPU cache, making it possible for an attacker to learn the value of `x`, or if `index` is attacker controlled, all of virtual memory.

Spectre V1 [25]. The bounds-check-bypass variant of Spectre works by tricking the processor into doing an out-of-bounds array access by speculatively executing the body of an if statement. Software mitigations usually entail manually annotating kernel branches with `lfence` instructions or array accesses with special macros that never read out of bounds.

Spectre V2 [25]. Modern CPUs use a Branch Target Buffer (BTB) to predict the targets of indirect branches. At a high level, a BTB is table mapping from instruction address to the last jump target for the branch instruction located at that address. Processors use BTBs so they can speculatively start executing code following an indirect branch before resolving the true target of that branch. Poisoning the BTB enables attacker code to make the CPU mispredict the targets of indirect branches and route transient execution to specially chosen Spectre gadgets.

There is no single mitigation for Spectre V2. It is commonly mitigated by replacing every indirect branch with a retpoline sequence [21] that prevents speculative execution, plus additional kernel logic to flush the BTB on context switches to protect user processes from one another.

Speculative Store Bypass [18]. This attack exploits store-to-load forwarding in modern processors to learn the contents of recently written memory locations. The only available mitigation is a processor mode called Speculative Store Bypass Disable (SSBD), but enabling it has severe negative performance impacts. Given the difficulty of exploiting Speculative Store Bypass and the considerable cost of mitigating it, by default SSBD is only used by Linux for processes that specifically opt in to it via `prctl` or `seccomp`.

3.3 Microarchitectural Data Sampling (MDS)

Microarchitectural Data Sampling describes a class of attacks involving leaks from various microarchitectural buffers within the CPU [11, 43, 47]. Unlike other Spectre and Meltdown variants, MDS attacks cannot be targeted to specific victim addresses, which makes them more challenging to exploit.

From an attacker perspective there are many different variations of MDS with their own specific mechanisms and capabilities. However, mitigations all fall into two categories: specific microarchitectural buffers need to be cleared on every privilege domain crossing or hyperthreading must be disabled to prevent an attacker and victim from simultaneously running on the same physical core. Clearing these CPU buffers is costly because of how frequently it must be done. Not using hyperthreading would have an even larger cost, but by default hyperthreading is enabled even for vulnerable CPUs because the risk was viewed acceptable given the performance difference.

4 End-to-End Benchmarks

We start by evaluating the total cost attributable to all mitigations for transient execution attacks. This value is different for each individual CPU, so we compare both across generations of processors and between vendors. Our goal is to gain a high level understanding of which mitigations are relevant from a performance perspective.

The primary impact of transient execution attacks is to leak information across protection boundaries. Accordingly, mitigations to prevent such leakage often involve extra operations when the CPU transitions from one protection domain to another. Alternatively, some mitigations must be enabled continuously while untrusted code is being executed. Based on this, we focus on several particularly relevant protection boundaries: the user-kernel interface for the operating system, and the sandboxing that web browsers' JavaScript engines provide between execution contexts for different sites. The boundary between a hypervisor and its guest operating system is also notable, but we did not find significant performance differences between running virtualization workloads with and without mitigations enabled.

In addition, we consider the case of a compute-intensive workload running within a single operating system process. This involves no protection boundary crossings, and thus measures only the impact of mitigations the operating system keeps enabled all the time.

4.1 Methodology

In the experiments we use eight different CPU microarchitectures from two vendors. Considering different microarchitectures enables us to observe design improvements between successive releases. The processors we evaluate span from before the discovery of Spectre and Meltdown (Broadwell, Skylake Client, and Zen) to the most recently available Intel mobile and server microarchitectures (Ice Lake Client and Server respectively) and AMD microarchitecture (Zen 3). Despite sharing the same name, Ice Lake Client and Ice Lake Server are different microarchitectures and were designed separately. Table 2 lists out detailed information on each CPU and Table 1 indicates which mitigations are used on each.

This diversity of systems gives a broad view of the ecosystem, but all the different dimensions they vary on complicates our work. The processors range from 1.0 GHz to 3.7 GHz and from 2 cores to 32 cores. Newer ones incorporate not just design improvements, but also tend to have smaller transistors, faster RAM, and so forth. For these reasons our experiments focus primarily on relative differences between configurations of the same machine. All machines have an up-to-date kernel as of October 2021: either version 5.11, the 5.14 release, or the 5.4 long-term maintenance release.

To measure the impact of individual mitigations, we run Linux with the default set of mitigations enabled, and then use kernel boot parameters to successively disable them to determine the overhead that each one causes. Some mitigations are applied separately by Firefox, which we control via its `about:config` interface.

When we started running experiments, variability observed on a single configuration was frequently on the same scale as the overheads we were trying to measure. Additional techniques were required to account for this. We adopted a methodology of running each benchmark configuration many times while tracking the average and 95%-confidence interval, stopping once the error was small enough. Benchmark scores for individual runs of the same configuration would vary by a couple percent each time, but the many iterations give us an accurate estimate of the true average.

4.2 LEBench

LEBench [41] is a collection of microbenchmarks for measuring specific operating system operations.[1] In this experiment, we track the geometric mean of benchmarks from the suite. As seen in Figure 2 the overhead has decreased sharply for newer processors: CPUs that incorporate hardware mitigations (for Intel) or from a vendor whose CPUs were never vulnerable to some of the attacks (AMD) exhibit substantially smaller overheads.

Also notable is that only a small number of mitigations are responsible for nearly all of the overheads. Collectively, all unlisted mitigations caused a fraction of a percent slowdown on Zen 2, but on the other processors had no statistically-significant impact at all.

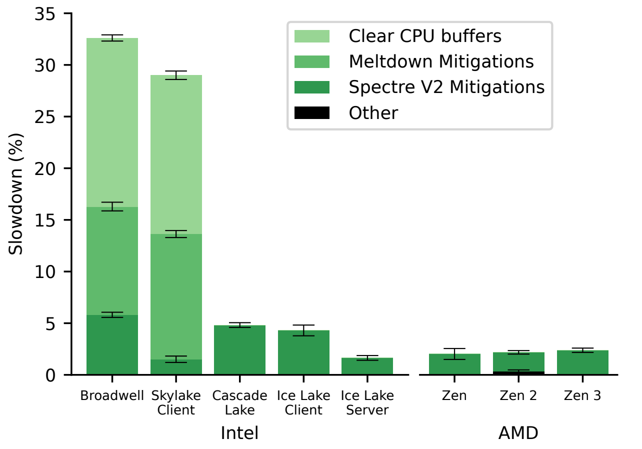

Figure 2. The overhead of mitigations on the LEBench benchmark suite which stresses the operating system interface. Error bars show 95% confidence intervals.

4.3 Octane 2

Octane 2 is a benchmark for JavaScript performance, which we run from within Firefox. Figure 3 plots the percent decrease in scores caused by enabling each mitigation in turn. JavaScript mitigations (Index masking, object mitigations, and "other JavaScript") are shown in blue, while operating system controlled mitigations including Speculative Store Bypass Disable (SSBD) and "other OS" are shown above them in green.

All JavaScript mitigations are implemented by the JIT engine inserting extra instructions into the generated instruction stream, and are used to prevent different variations of the Spectre V1 attack. For instance, index masking ensures that speculative accesses to an array do not index past the end of an array. It does so by placing a conditional move instruction before every array access which checks whether the access is in bounds and overwrites the index with zero otherwise. This check overall takes very little time but it prevents the CPU from starting to pull the array contents into cache until the array length is known. Across many millions of array accesses in the Octane 2 benchmark, this ends up causing a non-trivial cost.

[1]We use the version of the LEBench benchmarks distributed with the WARD system [5], which addresses a few issues with the original.

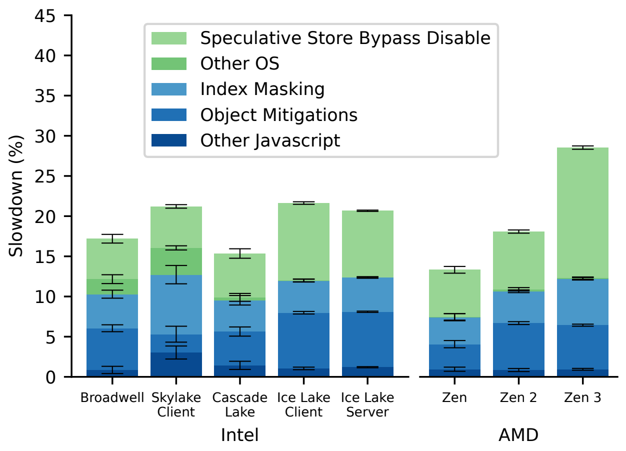

Figure 3. Slowdown on the Octane 2 browser benchmark caused by JavaScript and operating system level mitigations.

Speculative Store Bypass Disable is an OS level mitigation that is disabled by default for most processes, but on the kernel versions we're using is enabled for Firefox because it uses seccomp. Starting with Linux 5.16 released in January 2022, the kernel by default no-longer enables the mitigation for seccomp processes [30]. Applications can still enable the mitigation manually, but Firefox releases so far don't override the kernel setting.

This mitigation may stop being relevant. Intel has reserved a bit in in the `ARCH_CAPABILITIES` model-specific register to indicate that a given processor isn't vulnerable to Speculative Store Bypass and therefore that the associated mitigation is neither needed nor implemented. However, we do not know of any CPUs for either vendor that set that bit, not even models that came out years after the attack was discovered.

4.4 Virtual Machine Workloads

We measure two different virtual machine workloads relevant to how VMs are used in production. The performance of running LEBench inside of a virtual machine with and without host mitigations enabled mirrors running a customer application on a cloud provider. Execution primarily (but not exclusively) stays within the VM so we would expect host mitigations to have limited impact on the performance observed by the guest. This matches our observations: measured overhead was ±3% on all systems, signaling that the mitigations applied by a hypervisor do not have significant impact. Some runs suggested a slowdown in the range of 1-3%, but our methodology resulted in too much variability between runs to be confident whether or not that was caused by noise. In any case, we were unable to attribute the slowdown to specific mitigations because the slowdowns are so small.

Secondly, we measure the overhead of virtual machine exits by running the *smallfile* and *largefile* microbenchmarks from LFS [42] against an emulated disk. The median overhead was under 2%, but once again, we observed high variability between runs. This workload performs many security boundary crossings because every access to the emulated disk requires running code within the hypervisor. However, in contrast to LEBench that reached millions of system calls per second, the higher cost of VM exits meant that this experiment only reached several tens of thousands of VM exits per second. We believe that this explains the lack of a clear slowdown; the comparatively small number of protection domain crossings means that even though the time spent on mitigations during a single VM exit is likely higher than for a system call, in relative terms it is not enough to meaningfully impact the end-to-end performance.

4.5 PARSEC

As a final experiment we measure the overhead of running the *swaptions*, *facesim*, and *bodytrack* benchmarks from the PARSEC suite. These were chosen to get good coverage of compute-intensive benchmarks with different working set sizes. None involve significant numbers of calls into the operating system nor user-level sandboxing, as explored by the previous experiments, which makes them ideal to measure the impact solely of "always on" mitigations that the OS applies to running processes.

We were unable to observe any meaningful difference between running with and without the default set of mitigations: total runtime was usually within ±0.5% for the two configurations, and never differed by more than 2%. This serves as a reminder that slowdowns from transient execution attack mitigations aren't relevant to all workloads.

The one exception is that we observed significant overheads by force-enabling mitigations for Speculative Store Bypass. §5.5 explores this in more detail.

4.6 Summary

Each of these benchmarks plots a different trajectory of mitigation costs. Workloads that stress the operating system interface have received the most attention, and overheads on LEBench have gone from over 30% on older Intel CPUs to under 3% on the latest models, thanks to fixes for several of the attacks. By contrast, none of the attacks impacting JavaScript performance have been addressed in hardware and overhead on Octane 2 has remained in the range of 15% to 25%. Our compute-intensive benchmark has negligible overhead regardless of the processor, and we did not observe significant overheads on either of the two VM workloads measured. These trends are consistent with prior work from Phoronix [31], which found big improvements on OS workloads (perf-bench, ctx_clock, etc.), moderate but consistent overheads for web browsers (Selenium), and minimal overheads for the more compute-intensive workloads.

There has been a significant effort from computer architecture researchers towards addressing Spectre V1 [1, 3, 50, 53, 54], but interestingly software mitigations for the attack had no measurable impact on LEBench performance. By contrast, they account for around half the overhead on the browser workload.

It is also worth pointing out that all three attacks with significant overhead on new processor are actually quite "old". Spectre V1 and Spectre V2 were the first transient execution attacks discovered (along with Meltdown, which was discovered at the same time), while Speculative Store Bypass followed only a matter of months later. Over the subsequent three years of transient execution attack discoveries, they've all either been quickly resolved in hardware or had a negligible cost to mitigate in software. This paints an optimistic outlook for the future (assuming this trend remains true).

5 Performance of Individual Mitigations

This section explores the individual mitigations that had significant performance impact to the previously shown end-to-end overheads. Our aim is to understand why some mitigation costs have declined while others have not. Furthermore, we also want to understand whether moving mitigations from software to hardware truly makes them faster.

For each mitigation, we attempt to isolate the relevant instruction sequence and examine what the cost is on each of our processors. To achieve precise timings, we rely on the timestamp counter functionality available on x86 and average over one million runs to eliminate noise.

5.1 Meltdown

Meltdown mitigations account for one of the most substantial performance impacts on LEBench, single-handedly causing an around 10% overhead. On processors vulnerable to Meltdown, production operating systems use page table isolation (PTI) to mitigate it. This approach adds significant overhead to every user-kernel boundary crossing, because it requires switching the page tables every time via a `mov %cr3` instruction. Among the systems we evaluated, only Broadwell and Skylake are vulnerable to Meltdown.

As seen in Table 3, on these processors the cycles required to swap page tables when entering and again on leaving the kernel far exceeds the time for the actual `syscall` or `sysret` instruction that triggers the entry/exit. For syscalls, the Ice Lake Client CPU takes fewer cycles (which will prove a pattern—likely due to its lower base clock speed) and the Cascade Lake model stands out by taking longer than both earlier and later Intel models.

One other impact of page table isolation is that on old processors it can cause increased TLB pressure due to much more frequent TLB flushes. Both Broadwell and Skylake Client, however, support PCIDs which tag page table entries with a process identifier. This allow many TLB flushes to be avoided, and makes TLB impacts marginal compared to the direct cost of switching the root page table pointer.

CPU	`syscall`	`sysret`	swap `cr3`
Broadwell	49	40	206
Skylake Client	42	42	191
Cascade Lake	70	43	N/A
Ice Lake Client	21	29	N/A
Ice Lake Server	45	32	N/A
Zen	63	53	N/A
Zen 2	53	46	N/A
Zen 3	83	55	N/A

Table 3. Average cycles to execute a `syscall` or `sysret` instruction, and for vulnerable processors, to swap page tables.

5.2 Microarchitectural Data Sampling

The other substantial mitigation on LEBench is clearing CPU buffers, which is required to mitigate Microarchitectural Data Sampling (MDS). On processors that are vulnerable to MDS, a microcode patch extends the `verw` instruction to also implement this clearing functionality. Without the patch the `verw` only has its old behavior related to segmentation.

Table 4 shows that the cost of performing this flush is approximately 500 cycles. This cost is substantial because microarchitectural buffers must be flushed not just on context switches between processes but also on every kernel-to-user privilege transition. Recent Intel processors and all processors from AMD are not vulnerable to MDS. On these processors the `verw` has only its legacy segmentation-related behavior and takes only tens of cycles.

Vendor	CPU	Clear Cycles
Intel	Broadwell	610
	Skylake Client	518
	Cascade Lake	458
	Ice Lake Client	N/A
	Ice Lake Server	N/A
AMD	Zen	N/A
	Zen 2	N/A
	Zen 3	N/A

Table 4. Cycles required to clear microarchitectural buffers using the `verw` instruction on processors vulnerable to MDS.

5.3 Spectre V2

Spectre V2 involves poisoning the branch target buffer so that an indirect branch in victim code jumps to a Spectre gadget. As we saw, mitigating Spectre V2 is a small but largely consistent drag on LEBench performance across all the processors.

Indirect Branch Restricted Speculation. Indirect Branch Restricted Speculation (IBRS) was the first mitigated proposed for Spectre V2 and is enabled by setting a MSR bit which must be repeated on every entry into the kernel. Newer Intel processors—Cascade Lake and onward—support enhanced IBRS (eIBRS), which allows the operating system to enable IBRS once at boot time, and have it remain in effect without additional system register writes.

Retpoline. The cycle cost of doing this MSR write on every system call was viewed as unacceptably high [46], so production operating systems investigated alternative approaches, ultimately settling on retpolines for any processor not supporting eIBRS. This makes them the primary software mitigation for Spectre V2 today.

Retpolines involve replacing every indirect branch in the kernel with an alternate instruction sequence. A retpoline sequence has identical behavior to an indirect branch instruction, except that the branch destination (and more importantly any Spectre gadgets) are never jumped to speculatively.

There are a couple variations of retpolines, with slightly different characteristics. So called "generic retpolines" use a code sequence involving a `call` instruction, a write instruction to replace the saved return address with the jump target, and a `ret` instruction to cause the processor to speculatively jump back to the call site (due to the return value stack) before correcting to the intended branch target. This version works on both Intel and AMD processors.

An alternative version "AMD retpoline", involves simply doing an `lfence` followed by a normal indirect branch. As might be inferred from the name, this variant is not intended to work on Intel: code using it would still be vulnerable to Spectre V2. Concurrently with this work, however, researchers discovered a race condition that causes them to not protect AMD processors either [34]. Linux subsequently switched to prefer generic retpolines on AMD [27].

```
1  generic_retpoline:
2      call 2f   ; Jump forward to line 6
3  1:  pause     ; [skipped]
4      lfence    ; [skipped]
5      jmp 1b    ; [skipped] Jump back to line 3
6  2:  mov %r11, (%rsp) ; overwrite return address
7      ret       ; Jump to target destination
8
9  amd_retpoline:
10     lfence     ; Wait for loads to complete
11     call *%r11 ; Jump to address in r11
```

Figure 4. Assembly sequences for the two kinds of retpolines

Table 5 shows extra cycles of each of these variations across our machines, relative to a baseline of doing an unsafe indirect branch. One noticeable takeaway is that IBRS adds tens of cycles of overhead to indirect branches except on processors with eIBRS support (Cascade Lake and the two Ice Lake CPUs) where it is inexpensive. Retpolines, however, can be as or even more costly.

The AMD processors have different performance executing AMD retpolines: on the Zen 2 model we measure no overhead compared to a normal indirect branch, while the other AMD processors they are even slower than a generic retpoline.

CPU	Baseline	IBRS	Generic	AMD
Broadwell	16	+32	+28	N/A
Skylake Client	11	+15	+19	N/A
Cascade Lake	3	+0	+49	N/A
Ice Lake Client	5	+0	+21	N/A
Ice Lake Server	1	+1	+50	N/A
Zen	30	N/A	+25	+28
Zen 2	3	+13	+14	+0
Zen 3	23	+19	+13	+18

Table 5. Cycles to perform an indirect branch with either no mitigations, IBRS enabled, using generic retpolines, or using AMD retpolines.

Indirect Branch Prediction Barrier (IBPB). In addition to preventing indirect branches in the kernel from being hijacked, it is also important that one user process cannot launch a Spectre V2 attack against another process. To prevent this attack, on every context switch between processes the operating system triggers an Indirect Branch Prediction Barrier [22] to clear the branch target buffer. To do so, the OS executes a `wrmsr` instruction to set bit zero in the `IA32_PRED_CMD` Model Specific Register.

We verified across all our processors that executing an IBPB between poisoning the branch target buffer and performing an indirect branch prevents execution from being routed to the attacker-controlled target. Interestingly, however, we noticed that the performance counters report that indirect branches executed after an IBPB result in mispredictions. We speculate that this behavior is caused by the IBPB setting all entries in the branch target buffer to point to a specific harmless gadget rather than simply clearing them.

Table 6 shows that the cost of an IBPB has generally declined over time from many thousands of cycles on the Broadwell server to hundreds of cycles on Cascade Lake and Ice Lake Server. This improvement is likely related to the fact that older processors implemented IBPB via a microcode patch, whereas newer ones may have some amount of support in hardware. The Ice Lake Client processor somewhat bucks the trend of improving performance when compared to the earlier Cascade Lake, but still requires many fewer cycles than Broadwell or Skylake. AMD processors we tested show a similar improvement across generations.

Return Stack Buffer Filling. When a user process employs generic retpolines to protect itself from Spectre V2, it is counting on the return stack buffer not being tampered with during the code sequence. Unfortunately, if the operating system triggers a context switch at an inopportune time then

Vendor	CPU	IBPB cycles
Intel	Broadwell	5600
	Skylake Client	4500
	Cascade Lake	340
	Ice Lake Client	2500
	Ice Lake Server	840
AMD	Zen	7400
	Zen 2	1100
	Zen 3	800

Table 6. Average cycles to execute an indirect branch speculation barrier.

this condition might be violated. Linux uses two approaches to guarantee that user-level retpolines still work despite interrupts potentially happening at any time during execution.

The first is a static analysis pass over the Linux kernel at build time to ensure that the operating system itself doesn't have unbalanced `call` and `ret` pairs anywhere, which incurs no runtime cost at all. Since any code compiled with the regular toolchains will already have this property, this check is not expected ever to fail.

Secondly, when context switching between different user threads Linux will fill the the return stack buffer with harmless entries. This is required so that any interrupted retpoline sequence will avoid jumping to any Spectre gadgets—meaning that despite not causing a speculative jump to the intended retpoline landing point, it will still produce safe results.

Table 7 shows the cycles required to fill the return stack buffer on each processor. There is improvement across generations of Intel processors but less of a clear trend across the AMD CPUs. These changes are likely realized more by improving performance overall than trying to optimize for return stack buffer filling specifically, but regardless, the cost of these mitigations is relatively minor compared to the total overhead of doing a context switch between processes (which takes at least several thousand cycles)

Vendor	CPU	RSB Fill Cycles
Intel	Broadwell	130
	Skylake Client	130
	Cascade Lake	120
	Ice Lake Client	40
	Ice Lake Server	69
AMD	Zen	114
	Zen 2	68
	Zen 3	94

Table 7. Cycles to stuff the RSB.

Return stack buffer filling also provides protection against variations of SpectreRSB [26], which exploits the return stack buffer itself. Thus while the overall toggle to enable the functionality is controlled by Linux's `nospectre_v2` option, some amount of the overhead attributed to Spectre V2 should probably be accounted to mitigating SpectreRSB instead.

5.4 Spectre V1

On the Octane 2 benchmarks, the various Spectre V1 mitigations collectively accounted for a large fraction of the total overhead. We discuss each of them in more detail.

lfence. One mitigation for Spectre V1 is to execute an `lfence` instruction immediately following each bounds check and `swapgs` instruction. This instruction waits until all prior loads have resolved, thereby preventing any subsequent Spectre gadget from executing. The cost of an `lfence` varies significantly based on operations in flight. Table 8 shows the results of a simple microbenchmark of running an `lfence` instruction in a loop. An important caveat is that the performance will depend a lot on what other instructions have been executed prior so this is not a fully representative experiment.

We see that all times are roughly of the same scale, with newer processors showing better performance. The `lfence` does more work on AMD than on Intel (as evidenced by the AMD retpoline sequence described earlier) so the numbers are not directly comparable across vendors.

Vendor	CPU	lfence cycles
Intel	Broadwell	28
	Skylake Client	20
	Cascade Lake	15
	Ice Lake Client	8
	Ice Lake Server	13
AMD	Zen	48
	Zen 2	4
	Zen 3	30

Table 8. Cycles to execute a single lfence instruction on each machine. In real applications, the cost will heavily depend on the other loads in flight.

Index Masking. Instead of preventing speculation past bounds checks, an alternative mitigation is to force the array index to zero for any out of bounds access. SpiderMonkey (the JavaScript engine used by Firefox) uses this strategy: before every array indexing operation it inserts a `cmov` instruction that overwrites the array index with zero if it would be past the end of the array. Unlike in many compiled languages, JavaScript always knows the lengths of arrays so this mitigation can be applied automatically to the generated assembly. On the committed execution path the conditional move will always be a no-op (because as a safe language JavaScript always does bounds checks), but in the speculative case it blocks execution until the array length has resolved. Our measurements of the Octane 2 benchmark suite indicate this approach incurs a roughly 4% performance overhead on most of the systems.

Object Mitigations. Since JavaScript is dynamically typed, the compiler must insert many runtime checks on the types of variables. This presents another possible avenue for Spectre V1 attacks, because mis-speculating an object's type can cause its fields to be misinterpreted [24], potentially resulting in out of bounds memory reads. The mitigation is similar to index masking: object guards insert a conditional move that zeros out the object pointer if the check fails. This mitigation incurs an overhead on Octane 2 on the order of 6% on the tested processors.

5.5 Speculative Store Bypass

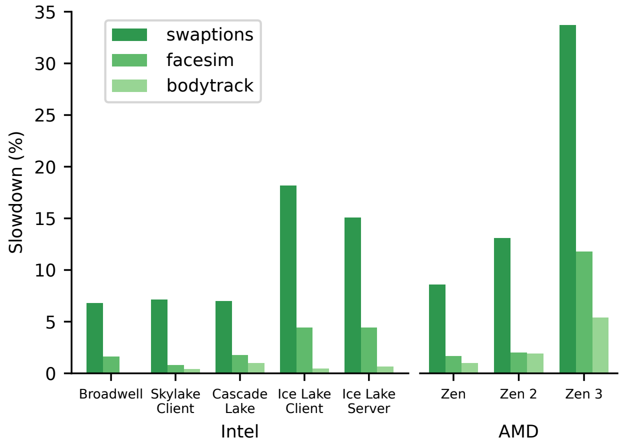

Figure 5. The slowdown caused by Speculative Store Bypass Disable on three benchmarks from the PARSEC suite.

Speculative Store Bypass exploits the processor's store-to-load forwarding to enable an attacker to learn the contents of recently written memory locations. The only available defense against the attack is to enable a processor mode called Speculative Store Bypass Disable (SSBD) that blocks this forwarding. A downside is that this can come at substantial cost, even when normal non-malicious code is being run.

The compromise reached by the Linux developers was to enable SSBD only for processes which opted into it via its `prctl` or `seccomp` interfaces. To see the full impact of this mitigation if enabled all the time, we measured the slowdown it causes to some benchmarks from PARSEC. Figure 5 shows that the slowdown can be as much as 34%, and is trending worse over time. It isn't entirely clear why this would be the case, but it may be related to newer processors have a more complete SSBD implementation compared to what was possible via microcode patches. These overheads are especially considerable given that the combined impact of all default mitigations for these benchmarks is generally well under one percent (§4.5).

5.6 L1 Terminal Fault

One other attack worth mentioning is L1 Terminal Fault, which can leak the entire contents of the L1 cache when page tables contain PTEs with certain bit patterns. Linux avoids ever creating such PTEs, which can be done with essentially no overhead. This is consistent with it not showing up in our end-to-end performance study earlier.

However, the problem is more severe when virtual machines are involved because an untrusted guest operating system could insert such specially crafted PTEs into its own page table. Doing so would enable it to learn L1 cache contents lingering from memory accesses done by the host. The necessary mitigation on vulnerable processors is for the host to flush the L1 cache prior to entering a guest virtual machine.

Our benchmarking of virtual machine workloads did not show any measurable impact from enabling this mitigation and it has also been patched on newer processors, so the relevance should be minimal going forward.

5.7 Other attacks

The attacks discussed so far are hardly the only transient execution attacks discovered. Many others like System Register Read [35], and so forth have commanded significant time and attention for computer architects, operating system developers, and security researchers. However, the cost they incur on workloads today seems to be minimal, so we skip evaluating them individually.

6 Analysis of Hardware Spectre V2 Mitigations

For nearly all the attacks we've looked at so far, either the mitigation approach has remained the same across all the processor generations we've studied, or it has gone from an expensive software mitigation to a hardware fix with no measurable cost at all. Spectre V2 notably does not follow this trajectory. It has an array of hardware and software mitigations, yet remains a non-trivial expense on every CPU we've tested. In this section we attempt to understand under which conditions the Branch Target Buffer is used to speculatively execute instructions and when not.

6.1 Measuring Speculation

To understand when CPUs speculatively execute instructions, we need a method to determine what instructions are being speculatively executed by the CPU. Bölük [7] describes a technique using performance counters to determine whether a processor starts speculatively executing at a given address, which we adopt to probe the behavior of the Branch Target Buffer, as we describe next.

The performance counters are specific to an individual generation of CPU and provide detailed information about microarchitectural events. All our processors have a performance counter to measure the number of cycles that the divider (the

		With intervening system call			No system call	
Vendor	**CPU**	user→kernel	user→user	kernel→kernel	user→user	kernel→kernel
Intel	Broadwell	✓	✓	✓	✓	✓
	Skylake Client	✓	✓	✓	✓	✓
	Cascade Lake		✓	✓	✓	✓
	Ice Lake Client		✓	✓	✓	✓
	Ice Lake Server		✓	✓	✓	✓
AMD	Zen	✓	✓	✓	✓	✓
	Zen 2	✓	✓	✓	✓	✓
	Zen 3					

Table 9. Whether the processor will speculatively execute an indirect branch in the given configuration when IBRS is disabled. A check mark in column X→Y indicates that training the branch target buffer in mode X is able to control the target of a subsequent victim indirect branch in mode Y, either with or without an intervening `syscall` and/or `sysret` instruction between them.

		With intervening system call			No system call	
Vendor	**CPU**	user→kernel	user→user	kernel→kernel	user→user	kernel→kernel
Intel	Broadwell					
	Skylake Client					
	Cascade Lake		✓	✓	✓	✓
	Ice Lake Client		✓		✓	
	Ice Lake Server		✓	✓	✓	✓
AMD	Zen	N/A	N/A	N/A	N/A	N/A
	Zen 2					
	Zen 3					

Table 10. Same as Table 9 but with IBRS *enabled*. IBRS always prevents problematic cases like user→kernel, but on many processors blocks all speculation including predicting the target of userspace indirect branches based on prior branches done by the same process (user→user). The entries for Zen are marked N/A because that processor does not support IBRS.

component within the CPU that executes divide instructions) is active. Some also have a dedicated performance counter to indicate the number of mispredicted indirect branches. By reading one of these counters before and after a block of instructions, we can tell whether executing that code triggered any of the relevant operations.

Figure 6 sketches out how we use this method to know whether code at a specific target location was executed speculatively. We execute indirect branches that may potentially be mispredicted as targeting a specially constructed landing pad, and see whether we measure any use of the divider corresponding to executing instructions at the landing pad. Care has to be taken to ensure no divide instructions are executed by the committed execution trace.

Interestingly, we sometimes observed mispredicted indirect branches without any divide instructions being performed, which we interpret as the processor speculatively executing instructions at a different location than the one we attempted to poison the branch target buffer with. For this reason, we focus on the performance counter for cycles with the divider active even when the other performance counter is also available.

Prior work discovered that for a Spectre V2 attack, only some bits of the virtual and physical addresses have to match between the victim and attacker [17]. However, to maximize the chance of success, we ensure all 64 bits match by sharing the same page of memory between the victim and attacker.

6.2 Results

Tables 9 and 10 show the results produced using this methodology. The columns indicate the mode that the attacker and victim run in respectively (e.g., user→kernel is the classic configuration of a user-space attacker trying to misdirect a victim running in kernel space). We also indicate the presence of an intervening `syscall` instruction.

Not shown in that figure, we also attempted to run the attacker in kernel mode and the victim in user mode. This is not reflective of a real world attack scenario, but it revealed that the same attacks processors vulnerable to the user→kernel version were vulnerable to a kernel→user attack.

One final note is that we did not manage to poison the branch target buffer at all on our Zen 3 processor. We suspect this isn't because it is immune to the attack, but rather due to some change to the Branch History Buffer (used to compute the index for the branch target buffer) or another implementation detail that experiments did not account for.

```c
void victim_target() {
  int c = 12345 / 6789;
}
void nop_target() {
  // do nothing
}

void(*target)();

void test() {
  // configure performance counter to measure
  // whether the divider is active
  configure_pmc(ARITH_DIVIDER_ACTIVE);

  // train the branch target buffer
  target = victim_target;
  for (int i = 0; i < 1024; i++)
    divide_happened();

  // potentially overwrite the entry
  ...

  // measure whether the trained entry is
  // jumped to speculatively
  target = nop_target;
  if (divide_happened())
    printf("victim_target ran speculatively!");
}

bool divide_happened() {
  // fill branch history buffer
  for (int i = 0; i < 128; i++)
    ;

  // flush branch target from cache
  clflush(target);

  // read performance counter
  int start = rdpmc();

  // perform the indirect branch
  (*target)();

  // see whether performance counter changed
  return rdpmc() > start;
}
```

Figure 6. Sketch of our approach. The `test` function prints whether it was able to poison the branch target buffer to route speculative execution to `victim_target`.

6.2.1 Indirect Branch Restricted Speculation. Recall that the original version of Indirect Branch Restricted Speculation (IBRS) was the first mitigation proposed for Spectre V2 but is not used by default on any production operating system because it requires an expensive write to a model-specific register on every entry into the kernel.

According to Intel documentation, this mitigation prevents indirect branches executed from less privileged modes from impacting the predicted destination of indirect branches in more privileged modes. We experimentally validated this claim by poisoning the branch target buffer and then seeing whether the processor would speculatively jump to the programmed branch destination. Our measurements indicated that toggling this mitigation caused the user space code to be unable to redirect kernel execution. However, subsequent experiments (replicated in Table 10) revealed that on pre-Spectre processors, IBRS was disabling all indirect branch prediction both in user space and kernel space. Not having this prediction even for user processes incurs a high performance cost.

6.2.2 Enhanced IBRS (eIBRS). Enhanced IBRS provides the same guarantees as the original IBRS but doesn't require an MSR write on every kernel entry. Given the lackluster performance of IBRS compared to retpolines, that may not seem promising, but the presence of this feature signals more serious mitigations built into the hardware. In particular, eIBRS does not disrupt indirect branch prediction at the same privilege level. When it is available, Linux by default uses eIBRS instead of retpolines.

As seen in Table 10, Cascade Lake and the two Ice Lake processors (the microarchitectures that support eIBRS) both do indirect branch prediction only based on prior indirect branches executed in the same privilege mode. We speculate this is achieved by using a branch target buffer that is either partitioned or tagged using a bit indicating the current privilege mode.

When running with eIBRS enabled, we have observed that kernel entries (caused by page faults, the `syscall` instruction, etc.) have bimodal performance. Most times they take a similar number of cycles (on the order of 70 cycles), but one in every 8 to 20 or so entries they take an additional 210 cycles. On the same processor, when running without eIBRS the time is always 70 cycles.

We have been unable to fully determine what is causing this behavior, but a few patterns have emerged. Under some conditions, the slow system calls will happen exactly every eight times, meanwhile at other times the processor will go long stretches without any slow syscalls. Additionally, we have sometimes observed behavior consistent with the branch target buffer being flushed only during slow kernel entries: poisoning the branch target buffer in the kernel prior to a system call causes misprediction of subsequent indirect branches in kernel mode only if the intervening kernel entry was fast. Monitoring performance counters reveal that slow system calls involve both executing more micro ops and more cycles spent stalling, but do not provide a clear hint of what those additional micro ops are doing.

The documentation for eIBRS doesn't make any promises, but the functionality may be intended as additional protection against attempts to misdirect indirect branches.

6.3 Takeaways

The original IBRS design not only added substantial overhead to every kernel entry, it also blocked indirect branch speculation everywhere. eIBRS improves on this by seemingly partitioning or tagging the branch target buffer based on the CPU privilege mode.

Partitioning / tagging the branch target buffer however is not a complete mitigation for Spectre V2. User processes still need their own defenses and even within the kernel indirect branches executed by the operating system could be used to mistrain the branch target buffer to misdirect subsequent operating system indirect branches. In concurrent work, Barberis et al. demonstrate a practical attack against eIBRS [4].

7 Discussion

Spectre V1. One takeaway from the previous sections is the continued impact of Spectre V1. There are no hardware mitigations available for the attack in high performance commercial CPUs. And yet, despite being among the first transient execution attacks discovered, it still presents a significant—and largely unchanging—overhead when mitigated in software.

Because Spectre V1 mitigations are specifically applied by JIT engines doing code generation, they also may present a unique opportunity for computer architects. The JIT annotates each vulnerable gadget with a leading `cmov` instruction. This pattern of a conditional move followed by a load instruction could be detected by hardware to trigger special handling.

Even if this approach proves unworkable, that doesn't rule out hardware acceleration for Spectre V1 mitigations. JIT engines generate code on the fly based on the processor they are running on, which means that unlike native applications, the author of a given JavaScript application doesn't need to be involved in porting/recompiling it to leverage a new ISA extension. And since current web browsers generally receive new updates on a short release cycle, any new hardware could be leveraged quickly.

Speculative Store Bypass. Speculative Store Bypass Disable was initially implemented in microcode, and while we cannot tell whether more recent CPUs include actual hardware changes as well, the performance overhead hasn't improved. This attack in particular also emphasizes the need to look at the performance impacts of transient execution attacks across representative workloads. Despite being disabled by default, Speculative Store Bypass Disable incurs a substantial overhead on JavaScript execution in web browsers—one of the most common workloads run by end-users.

This may be changing however. Linux 5.16 released in January 2022 has a different default configuration for Speculative Store Bypass. Going forward, processes that use seccomp but do not specifically request SSBD will not have the mitigation enabled. This is particularly notable because Firefox currently falls in that category. It remains to be seen whether Mozilla will issue a patch to restore the old mitigation behavior, but if not, this could be a signal that SSBD was never actually required in the first place.

Additionally, Intel's inclusion of a hardware capability to detect whether a processor is vulnerable to Speculative Store Bypass (without a way to toggle it) strongly suggests that they believe future hardware will be able to prevent the attack with negligible overhead.

8 Summary

Our goal was to answer a number of questions, which we now revisit.

Which attacks have the greatest performance impact? We found that the primary impact on current processors comes from mitigations for Spectre V1 and V2, and Speculative Store Bypass. These are some of the earliest attacks discovered: the first two are described in the first transient execution attack paper, and the third was discovered only a matter of months later. On operating system intensive workloads, older Intel processors also incur significant costs from Meltdown and MDS, but these have been resolved on the newest models.

What drives the cost of mitigations for those attacks? Other than Indirect Branch Prediction Barriers which address one component of Spectre V2, mitigations themselves have not been getting substantially faster. The performance improvement for operating system workloads can be explained by no longer needing many of the most expensive mitigations.

What predictions can we make about mitigation overheads going forward? We cannot know for sure, but there is reason to be optimistic. None of the attacks discovered in the last several years have much performance impact and there is potential that new CPUs may be able to mitigate Spectre V1 or Speculative Store Bypass with lower overhead. If the recent change in Linux to use SSBD in fewer places is adopted broadly, then a hardware mitigation for the latter attack may not even be required.

9 Conclusions

This paper assesses the evolution of performance penalties for mitigations against transient execution attacks by measuring their overheads across several generation of Intel and AMD CPUs.

On post-2018 processors, overhead from the OS boundary has mostly been eliminated in hardware. This means the high mitigation costs have been largely resolved for server workloads. At the same time, JavaScript sandboxing is still expensive. Across both workloads, most overheads that remain are caused by a small number of software mitigations, all addressing attacks that were discovered in 2018 or earlier and attacks published since require mitigations with only minor performance impact for recent processors.

A further analysis of individual mitigations shows that the performance of most mitigation code sequences remains relatively unchanged, and that hardware fixes are responsible for nearly all of the speedup. Spectre V1 and Speculative Store Bypass mitigations are significant and haven't declined across processor generations. However, it may be possible to reduce these overheads of these mitigations with hardware changes too; for example, the Spectre V1 mitigation has a recognizable pattern of a conditional move followed by a load instruction, which could be detected by hardware to trigger special handling in the future.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Luigi Rizzo for their comments and suggestions. We also thank Nickolai Zeldovich and Mengjia Yan for the valuable feedback they provided on this work. This work was partially supported by a Facebook Research Award.

References

[1] Ainsworth, S., and Jones, T. M. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (2020), ISCA '20, IEEE Press, p. 132–144.

[2] Arm, Ltd. Cache speculation side-channels. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability, 2020.

[3] Barber, K., Bacha, A., Zhou, L., Zhang, Y., and Teodorescu, R. SpecShield: Shielding speculative data from microarchitectural covert channels. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, WA, Sept. 2019), pp. 151–164.

[4] Barberis, E., Frigo, P., Muench, M., Bos, H., and Giuffrida, C. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security* (Aug. 2022). Intel Bounty Reward.

[5] Behrens, J., Cao, A., Skeggs, C., Belay, A., Kaashoek, M. F., and Zeldovich, N. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Banff, Alberta, Canada, Nov. 2020).

[6] Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 785–800.

[7] Bölük, C. Speculating the entire x86-64 instruction set in seconds with this one weird trick. https://blog.can.ac/2021/03/22/speculating-x86-64-isa-with-one-weird-trick/, March 2021.

[8] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium* (Baltimore, MD, Aug. 2018), pp. 991–1008.

[9] Bulck, J. V., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., and Piessens, F. LVI: Hijacking transient execution through microarchitectural load value injection. *2020 IEEE Symposium on Security and Privacy (SP)* (2020), 54–72.

[10] Canella, C., Bulck, J. V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A systematic evaluation of transient execution attacks and defenses. *CoRR abs/1811.05441* (2018).

[11] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Bulck, J. V., and Yarom, Y. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)* (London, United Kingdom, Nov. 2019), pp. 769–784.

[12] Carruth, C. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[13] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. SgxPectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)* (2019), pp. 142–157.

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

[15] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is dead: Long live KASLR. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems* (Bonn, Germany, July 2017), pp. 161–176.

[16] Hill, M. D., Masters, J., Ranganathan, P., Turner, P., and Hennessy, J. L. On the Spectre and Meltdown processor security vulnerabilities. *IEEE Micro 39*, 2 (2019), 9–19.

[17] Horn, J. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, January 2018.

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

[19] IBM. Potential impact on the processors in the power family. https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/, 2019.

[20] Intel, I. Affected processors: Transient execution attacks & related security issues by CPU. https://software.intel.com/content/www/us/en/develop/topics/software-security-guidance/processors-affected-consolidated-product-cpu-model.html, 2021.

[21] Intel, Inc. Deep dive: Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-retpoline-branch-target-injection-mitigation.

[22] Intel, Inc. Indirect branch predictor barrier. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html, 2018.

[23] Intel, Inc. Software guidance: Rogue data cache load. https://software.intel.com/security-software-guidance/software-guidance/rogue-data-cache-load, 2018.

[24] Kirzner, O., and Morrison, A. An analysis of speculative type confusion vulnerabilities in the wild. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 2399–2416.

[25] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy* (San Francisco, CA, May 2019), pp. 19–37.

[26] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies* (USA, 2018), WOOT'18, USENIX Association, p. 3.

[27] Kroah-Hartman, G. Linux 5.15.28 release announcement. https://lwn.net/Articles/887638/, Mar. 2022.

[28] Larabel, M. The performance impact of MDS / Zombieload plus the overall cost now of Spectre/Meltdown/L1TF/MDS. https://www.phoronix.com/scan.php?page=article&item=mds-zombieload-mit, 2019.

[29] Larabel, M. Looking at the linux performance two years after spectre / meltdown mitigations. https://www.phoronix.com/scan.php?page=article&item=spectre-meltdown-2, 2020.

[30] Larabel, M. Linux 5.16 loosens the spectre defaults around ssbd / stibp. https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.16-Spectre-SECCOMP-To-P, Nov. 2021.

[31] Larabel, M. A look at the cpu security mitigation costs three years after spectre/meltdown. https://www.phoronix.com/scan.php?page=article&item=3-years-specmelt&num=1, 2021.

[32] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium* (Baltimore, MD, Aug. 2018), pp. 973–990.

[33] Lutomirski, A. [patch] x86/fpu: Hard-disable lazy fpu mode. https://lore.kernel.org/lkml/CALCETrV9rXJOgdBY9Wyardo0NETA1meCEM_C4-e+SYsZAoUU7A@mail.gmail.com, 2016.

[34] Milburn, A., Sun, K., and Kawakami, H. You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection. *arXiv preprint arXiv:2203.04277* (2022).

[35] MITRE Corporation. Cve-2018-3640. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3640, 2018.

[36] Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D. M., and Stefan, D. Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium* (2021).

[37] Pizlo, F. What spectre and meltdown mean for webkit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018.

[38] Prout, A., Arcand, W., Bestor, D., Bergeron, B., Byun, C., Gadepally, V., Houle, M., Hubbell, M., Jones, M., Klein, A., Michaleas, P., Milechin, L., Mullen, J., Rosa, A., Samsi, S., Yee, C., Reuther, A., and Kepner, J. Measuring the impact of spectre and meltdown. In *2018 IEEE High Performance extreme Computing Conference (HPEC)* (2018), pp. 1–5.

[39] Ragab, H., Milburn, A., Razavi, K., Bos, H., and Giuffrida, C. CrossTalk: Speculative data leaks across cores area real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy* (San Francisco, CA, May 2021).

[40] Reis, C., Moshchuk, A., and Oskov, N. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1661–1678.

[41] Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., and Yuan, D. An analysis of performance evolution of Linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)* (Huntsville, Ontario, Canada, Oct. 2019), pp. 554–569.

[42] Rosenblum, M., and Ousterhout, J. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)* (Pacific Grove, CA, Oct. 1991), pp. 1–15.

[43] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)* (London, United Kingdom, Nov. 2019), pp. 753–768.

[44] Simakov, N. A., Innus, M. D., Jones, M. D., White, J. P., Gallo, S. M., DeLeon, R. L., and Furlani, T. R. Effect of meltdown and spectre patches on the performance of HPC applications. *CoRR abs/1801.04329* (2018).

[45] Stecklina, J., and Prescher, T. Lazyfp: Leaking FPU register state using microarchitectural side-channels. *CoRR abs/1806.07480* (2018).

[46] Torvalds, L. Re: Create macros to restrict/unrestrict indirect branch speculation. https://lkml.org/lkml/2018/1/21/192, 2018.

[47] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy* (San Francisco, CA, May 2019), pp. 88–105.

[48] van Schaik, S., Minkin, M., Kwong, A., Genkin, D., and Yarom, Y. CacheOut: Leaking data on intel cpus via cache evictions. *CoRR abs/2006.13353* (2020).

[49] Wagner, L. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018.

[50] Weisse, O., Neal, I., Loughlin, K., Wenisch, T. F., and Kasikci, B. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, Oct. 2019), pp. 572–586.

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

[52] Xiong, W., and Szefer, J. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv. 54*, 3 (May 2021).

[53] Yu, J., Mantri, N., Torrellas, J., Morrison, A., and Fletcher, C. W. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), pp. 707–720.

[54] Yu, J., Yan, M., Khyzha, A., Morrison, A., Torrellas, J., and Fletcher, C. W. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, Oct. 2019), pp. 954–968.