Understanding and Improving the Performance of Mitigating Transient Execution Attacks

by

Jonathan Behrens

B.S., Cornell University (2016)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2022

Signature of Author...
Department of Electrical Engineering and Computer Science
January 26, 2022

Certified by ....
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Co-Certified by ....
Adam Belay
Assistant Professor of Electrical Engineering and Computer Science
Thesis Co-Supervisor

Accepted by ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Understanding and Improving the Performance of Mitigating Transient Execution Attacks

by
Jonathan Behrens

Submitted to the Department of Electrical Engineering and Computer Science on January 26, 2022 in Partial Fullfillment of the Requirements for the Degree of Doctor of Philosophy in Electrical Engineering and Computer Science.

ABSTRACT

This thesis makes two contributions: (1) a measurement study of the performance evolution of mitigations against transient execution attacks over generations of processors, and (2) the WARD kernel design, which eliminates as much as half the overhead of mitigations on older processors.

The measurement study maps end-to-end overheads to the specific mitigations that cause them. It reveals that hardware fixes for several transient execution attacks have reduced overheads on OS heavy workloads by a factor of ten. However, overheads for JavaScript applications have remained roughly flat because they are caused by mitigations for attacks that even the most recent processors are still vulnerable to. Finally, the study shows that a few mitigations account for most performance costs.

WARD is a novel operating system architecture that is resilient to transient execution attacks, yet avoids expensive software mitigations that existing operating systems employ when running on pre-2018 processors. It leverages a new hardware/software contract termed the Unmapped Speculation Contract, which describes limits on the speculative behavior of processors.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Adam Belay
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

I want to extend my thanks to my advisors Frans Kaashoek and Adam Belay for their guidance and mentorship, to my other committee members Nickolai Zeldovich and Mengjia Yan for their valuable feedback, and to all the past and present members of PDOS who I have learned so much from over these years. Thank you.

I would also like to thank all of my collaborators including Jack Cook, Jules Drean, Anton Cao, Cel Skeggs, Amy Ousterhout, Joshua Fried, Hari Balakrishnan, Jon Gjengset, Malte Schwarzkopf, Lara Timbó Araújo, Martin Ek, Eddie Kohler, Robert Morris, Sagar Jha, Ken Birman, and Edward Tremel. Your insights and support have meant a ton.

I am extraordinarily thankful to my friends and family have supported me every step of this journey. You have shaped my experiences, given me meaning and purpose, and helped me drive positive change. I would not be where I am today without you.

⋆ ⋆ ⋆

This dissertation includes work from the following papers:

- Efficiently mitigating transient execution attacks using the unmapped speculation contract. Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. *OSDI 2020*.

- Performance Evolution of Mitigating Transient Execution Attacks. Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. To appear. *EuroSys 2022*.

Contents

Chapter 1

Introduction

Side-channel attacks leak information between protection domains outside of the normal information flow of a system. In late 2017 a new class of side-channel attacks was discovered impacting CPUs from all major vendors [32, 41]. These attacks—termed transient execution attacks—exploit details of how modern processors use speculative execution to run more quickly.

Transient execution attacks represent a concern for system developers because providing isolation is a key security responsibility for many kinds of software. For instance, an operating system must not allow processes running on the same machine to inadvertently leak information between one another, and web browsers must ensure that JavaScript running on one website cannot access state belonging to other sites. System developers have deployed a range of techniques devised to mitigate the impact of these transient execution attacks, but unfortunately they can introduce significant performance costs.

The overhead caused by mitigations is present on vulnerable processors dating from before the discovery of the attacks and also at least to some degree on all major commercial CPUs released after their discovery. This thesis (i) measures how overhead has evolved over subsequent generations of processors, and (ii) since processors from 2017 and earlier will remain in use for years to come, presents a new operating system design to reduce OS level overheads those CPUs.

1.1 Transient Execution Attacks

Speculative execution is an optimization used in the design of modern processors to drastically improve their performance, by enabling them to do useful work during times when they would otherwise be stalled waiting to load information from memory. When the CPU reaches a conditional branch instruction that it doesn't yet know whether will be taken or not, it predicts the outcome and then starts executing instructions that would follow. The prediction is based on prior executions of that same instructions and any other heuristics the processor may have.

Usually the branch prediction is correct and the speculatively executed instructions can be committed. When the prediction is incorrect, the CPU rolls back the architecturally visible results of the instructions executed after the branch. However, in general when reverting transiently executed instructions, their microarchitural effects—like inserting or evicting lines from the cache—are not undone. Observing the microarchitural effects of speculatively executed instructions enables transient execution attacks.

1.1.1 Example Attack

```
if (index < array_size) {
    int v = array2[array1[index] * 256];
    ...
}
```

Figure 1.1: Example code for a Spectre V1 attack

Spectre V1 was one of the earliest attacks discovered and leaks information between different processes, between a process and the operating system kernel, or between a sandbox and the code running within. The attack relies on a specific sequence of instructions called a “gadget” to be located in the victim’s code region (e.g., in the code for a system call). The gadget consists of a bounds check followed by two array accesses as shown in Figure 1.1, for which the index value can be controlled by the adversary.

To trigger the attack, the adversary will repeatedly call the gadget with in bounds indexes (e.g., by invoking the system call that has the gadget). This trains the CPU branch predictor that the branch is usually taken. Then, the attackers forces the array size to be evicted from the CPU cache and calls the gadget with an out of bounds index.

When execution reaches the bounds check, the processor will initiate a load to read the array size. However before that load completes, it will speculatively start executing the body of the if statement because the branch predictor has previously learned that the index is usually in bounds. Because the index is out of bounds, the first array access will read past the end of the array into arbitrary memory (e.g., allowing the attacker to access any part of memory) and pull the value into a CPU register. The second array access then loads a specific cache line whose index will depend on the first read.

Eventually the branch misprediction will be discovered and rolled back, but the cache contents will not be. This enables the attacker to later time how long it takes to access each cache entry and infer which value must have been pulled in by the victim application. Refer to the Project Zero article [23] and paper appendix [32] for a full attack description.

1.1.2 Other Attacks

After the discovery of the original transient execution attacks, a wide range of others were identified. They can be divided into three broad categories [22]:

- **Spectre type** attacks like the one described above rely on processor misprediction to incompletely roll back executed instructions.
- **Meltdown type** attacks achieve the same effect using a trapping instruction to trigger the roll back.
- **Microarchitural Data Sampling (MDS) type** attacks exploit the processor's forwarding logic to cause the processor to erroneously run instructions with data fed from a sibling hyperthread or previously running task, rather than the correct values.

1.1.3 Threat Model

To understand this wide range of attacks, it is useful to have a framework to describe them all. Based on the outline from Kiriansky, et al. [31] we can give a unifying definition: a transient execution attack involves an attacker and victim application co-located on the same physical machine in which the attacker (i) guides speculative control flow to cause, (ii) an access to a victim secret, (iii) and for it to be transmitted via a microarchitectural covert channel, (iv) to a receiver in the attacker's protection domain. These steps are depicted in Figure 1.2.

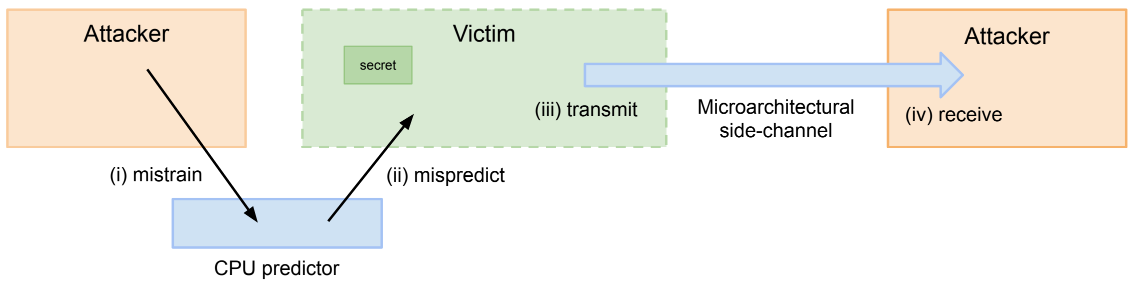

Figure 1.2: Steps involved in a transient execution attack.

In Spectre V1 for instance, the attacker (i) poisons the branch predictor, (ii) which causes an out of bounds load to pull the victim secret into a register, (iii) so a second memory operation pulls in a cache line whose index is determined by the secret, (iv) so that the attacker can recover the secret by probing which entries are present in the cache. Exactly how each step is conducted varies between attacks. For instance, there are many prediction structures within a modern CPU that can be leveraged to reroute speculative control flow, and the access in (ii) can be performed via a gadget in the victim application or entirely using instructions located within the attacker application. The final receive step is typically noisy and requires some degree of decoding of the observed value to recover the original secret.

Transient execution attacks are possible in settings with code running for two mutually-distrusting entities on the same CPU. This can be a user application attacking the OS, a user application targeting other user applications, a guest OS exploiting the hypervisor, and so forth.

1.2 Evolution of Mitigation Cost

Mitigations for transient execution attacks can introduce significant performance costs as surveys by Phoronix have demonstrated [35, 36, 38]. One goal of this thesis is to gain a more detailed understanding compared to prior work, including by measuring how specific mitigations impact performance.

We start with end-to-end benchmarks to identify which mitigations are relevant to performance. In selecting workloads, we direct our attention primarily towards security boundaries. This is because transient execution attacks in one way or another involve leaking information across a boundary and most of the mitigations to prevent them involve performing extra work each time execution crosses a boundary. Each selected workload stresses a different boundary: we use LEBench [48] to measure the OS boundary, Octane 2 [19] to profile JavaScript sandbox overhead, run a few virtual machine benchmarks, and verify that there isn't significant overhead for a few CPU intensive workloads running entirely within a single process.

By varying which mitigations are enabled during each experiment run, we're able to attribute overheads back to the specific mitigations that cause them. Our test systems vary on many dimensions unrelated to transient execution attacks (like core count, clock speed, and cache size) so our direct comparisons focus on relative differences between configurations of the same system.

On workloads that stress the Linux kernel interface (which have received the most attention) we find there have been substantial improvements with overhead on LEBench going from over 30% to less than 3%, and all measured overhead now attributable to a single attack. By contrast, the performance of JavaScript applications running inside Firefox are impacted by an almost entirely different set of mitigations, which on Octane 2 has caused overhead to remain roughly flat at 20%.

We aim to understand why some mitigation costs have declined while others have not, and to understand whether moving mitigations from software to hardware truly makes them faster. Therefore, we also conduct a detailed breakdown of individual mitigation code sequences to investigate their precise cycle costs. For each mitigation identified by the end-to-end benchmarks, we attempt to measure execution time of the associated instruction or instruction sequence on each of our impacted systems. Our experiments show some variations between processors in how long individual mitigations take, but demonstrate that the main source of improvement is that some costly mitigations are completed avoided on newer CPUs.

1.3 Ward

Mitigations are important for OS kernels because they make a particularly good target for transient execution attacks. First, an adversary can cause the kernel to speculatively execute code that leads to leakage of sensitive data. Even though the adversary cannot inject their own code to execute in the kernel, they can often have significant influence on what existing kernel code gets executed in speculative execution, by specifying particular system call arguments or setting up micro-architectural CPU state such as the branch predictor. Secondly, an OS kernel has access to all of the state on the computer. This means that an adversary running in one process can trick the kernel into leaking state from any other process on the same computer.

When running on processors designed prior to 2018, the only way for operating systems to prevent transient execution attacks is to use expensive software mitigations. Such processors will remain in use for years to come and cannot simply be ignored. WARD is a novel operating system architecture that is resilient to transient execution attacks, yet avoids much of the overhead caused by software mitigations when running on these processors. It is based on the Unmapped Speculation Contract, which describes limits on a processor's speculative behavior: namely that physical memory not mapped into a current or previous page table cannot impact microarchitectural state.

WARD leverages the USC by dividing the kernel into multiple domains. The K domain includes all information accessible to the operation system, while each process has its own associated Q domain consisting only of the userspace memory and kernel space data structures related to that one process. At any given point in time the processor is either executing in userspace, in a Q domain, or in the K domain.

Notably when executing in a Q domain, WARD is able to avoid many of the expensive software mitigations that would ordinarily be required. And since WARD is able to handle many system calls and traps entirely in the Q domain, it can achieve considerably better performance on many workloads compared to conventional operating system designs.

1.4 Contributions

One primary contribution of this thesis is to draw attention to the performance critical areas for improving transient execution mitigations, driven by (1) a survey mapping end-to-end overheads to the specific mitigations that cause them, and (2) detailed microbenchmarking of individual mitigations. To analyze hardware mitigations for Spectre V2, this thesis also contributes a new technique to measure speculation using ideas from Bölük [8].

Another contribution is articulating the Unmapped Speculation Contract, which describes upper limits of what speculative execution attacks can and cannot do. This thesis then demonstrates the benefits of the contract by presenting WARD, a novel kernel architecture that uses selective kernel memory mapping to avoid some of the costly transient execution mitigations required on older Intel processors.

1.5 Outline

The following chapter presents an end-to-end performance evaluation and goes into detail on each major attack describing both background on how it works as well as analyzing its impact on each evaluated system (§2). We then proceed to introduce the Unmapped Speculation Contract, which encapsulates some security guarantees that we believe even old processors are able to provide, and describe the WARD kernel design which improves the performance of OS intensive workloads by leveraging it (§3). Afterwards is an overview of related work (§4). We follow up with some discussion and ideas for future work (§5), and then conclude (§6).

Chapter 2

Performance Analysis

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. We go further and attribute overheads to individual mitigations to understand which ones matter to overall performance. We also study internal JavaScript runtime mitigations to understand whether the performance impact on browsers is different from operating systems.

In this thesis, the focus is 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 microarchitural state or waiting for in-flight operations to complete. Each of our workloads are chosen to stress a different security boundary. §2.2.2 measures the OS boundary, §2.2.3 the boundaries between JavaScript sandboxes, §2.2.4 between a guest OS and a hypervisor, and §2.2.5 confirms that mitigation overheads are low in the absence of security boundaries. Another possible boundary would be between WASM sandboxes, but we found that other than Swivel [43], which is already well studied, production WASM engines seem to either rely on site isolation [47] or neglect to mitigate transient execution attacks at all [3].

Our experiments look at a range of processors including both some that predate the discovery of Spectre and Meltdown but which are still in active use, newer ones which incorporate some mitigations, and even more recent models with still more mitigations. We evaluate five processors from Intel and three AMD processors so we can compare across vendors as well. For each processor and workload, we characterize the total overhead caused by mitigations and further attribute how much of the overall slowdown is caused by each individual mitigation. This informs our microbenchmarking of individual mitigations. For mitigations that incur meaningful overhead, we investigate in more detail to understand their performance characteristics.

We seek to answer the following questions: Which attacks are primarily responsible for the performance impact, and does that vary across processors or workloads? (§2.2) What drives the cost of mitigations for those attacks? (§2.3) What mitigations would benefit from hardware support to lower their cost and what predictions can we make about mitigation overheads going forward? (§2.5)

In doing so, this chapter seeks 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 [8].

There are some limitations. We are limited in the number of processor generations to 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. Additionally the conclusions we can draw are constrained by the lack of public details on how hardware mitigations are implemented. Finally, it is inherently uncertain what impact yet to be discovered attacks will have.

2.1 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. [11], which focus on enumerating and classifying the space of possible attacks.

2.1.1 Meltdown-Type Attacks

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

Meltdown [41] The original Meltdown attack is caused by speculatively translating kernel addresses for supervisor 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 of discovery, existing processors from Intel as well as some from IBM and ARM were vulnerable [4, 25, 30].

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 2.1: Default mitigations used by Linux on each processor. 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.

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 [61] On certain Intel processors, the present bit in PTEs 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 PTEs 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 [54] 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 special instructions for saving and restoring this state (e.g., `xsaveopt`) [42]. As a result, the trap handling overhead is often higher than the cost of unconditionally saving and restoring the registers.

2.1.2 Spectre-Type Attacks

Spectre-type attacks exploit speculative execution following a misprediction. They typically involve a specific gadget involving two memory loads; see Figure 2.1 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 seeing which array entry was pulled into the cache.

Notice that the code in Figure 2.1 matches the body of the `if` statement from Figure 1.1 (which shows a Spectre V1 attack). Other Spectre attacks have different supporting code but this same core gadget.

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

Figure 2.1: 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 [32] 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 [32] 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 [28] that halts further speculative execution, plus additional kernel logic to flush the BTB on context switches to protect user processes from one another.

Speculative Store Bypass [24] This attack—originally known as Spectre V4—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`.

2.1.3 Microarchitectural Data Sampling (MDS)

Microarchitectural Data Sampling describes class of attacks involving leaks from various microarchitectural buffers within the CPU [12, 51, 57]. 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.

2.2 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.

Later sections will go into more detail on how individual attacks work and the characteristics of their respective mitigations, but for now we wish only 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 two 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

Vendor	Model	Microarchitecture	Power (W)	Clock (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 2.2: Information about each of the CPUs we evaluate. All except the Ryzen 3 1200 have 2-way SMT ("hypertheads" in Intel terminology).

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.

2.2.1 Methodology

In the following benchmarks we evaluate across eight different CPU microarchitectures from two vendors. Considering different microarchictures enables us to observe design improvements between successive releases. Table 2.2 lists out detailed information on each CPU.

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 microarchictures (Ice Lake Client and Server respectively) and AMD microarchicture (Zen 3). Despite sharing the same name, Ice Lake Client and Ice Lake Server are different microarchitectures and were designed separately. All machines have an up-to-date kernel: either version 5.11, the 5.14 release, or the 5.4 long-term maintenance release.

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.

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.

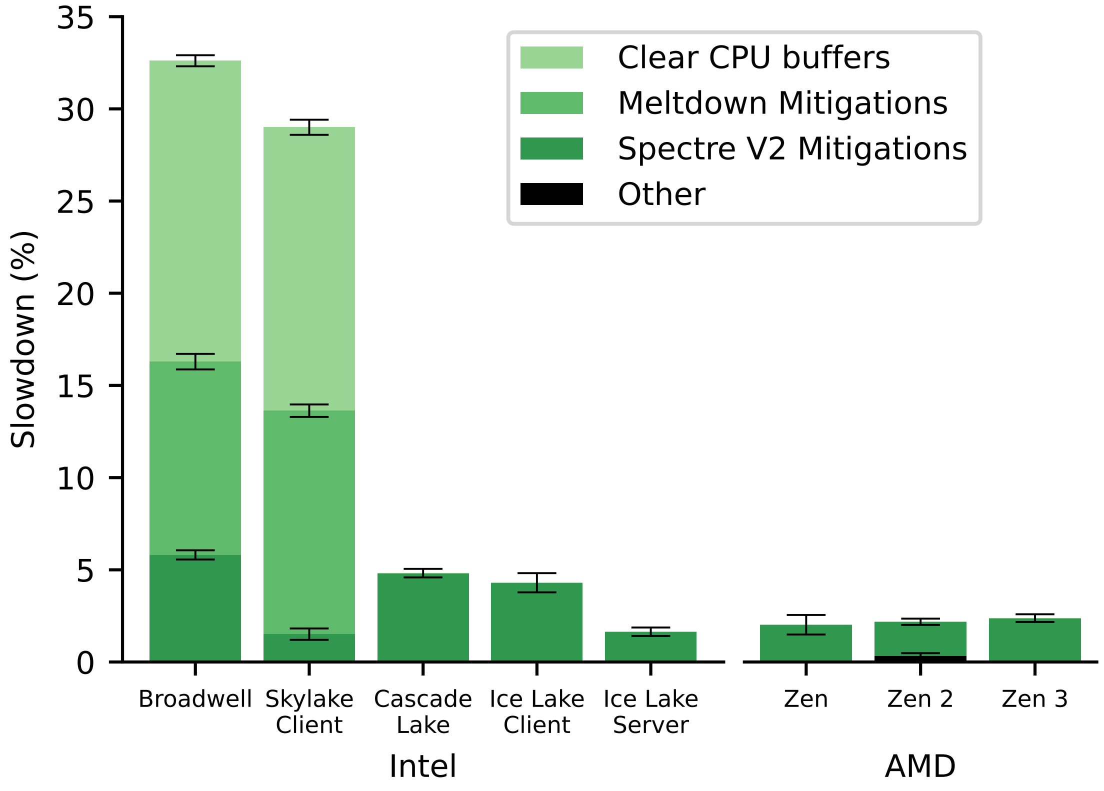

Figure 2.2: The overhead of mitigations on the LEBench benchmark suite which stresses the operating system interface. Error bars show 95% confidence intervals.

2.2.2 LEBench

LEBench [48] 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 2.2 the overhead has decreased sharply for newer processors: CPUs that incorporate hardware mitigations (for Intel) or from a vendor whose CPUs were not vulnerable to all attacks in the first place (AMD) exhibit substantially smaller overheads.

[1]To align with experiments from elsewhere in this thesis, we use the version of the LEBench benchmarks distributed with WARD [6] .

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.

2.2.3 Octane 2

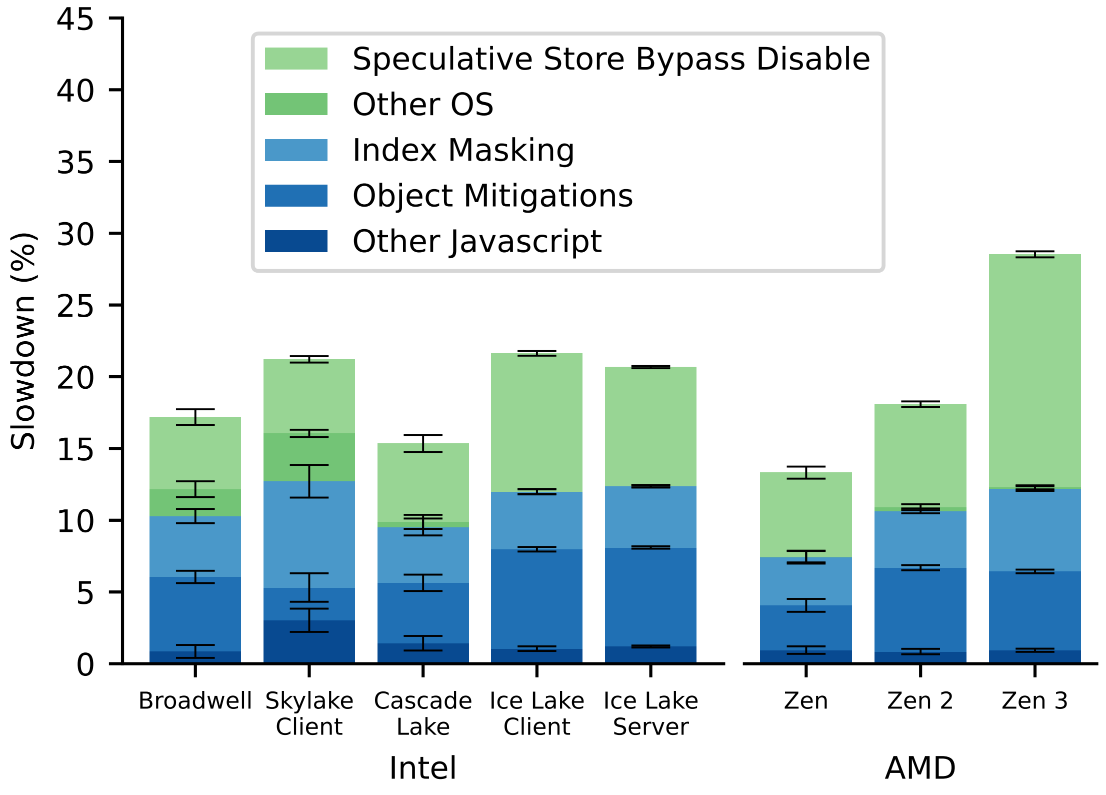

Figure 2.3: Slowdown on the Octane 2 browser benchmark caused by JavaScript and operating system level mitigations. Error bars show 95% confidence intervals.

Octane 2 is a benchmark for JavaScript performance, which we run from within Firefox. Figure 2.3 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.

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 [37]. Applications can still enable the mitigation manually, but Firefox releases so far don’t override the kernel setting.

This 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.

2.2.4 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 $\pm 3\%$ on all systems, signalling 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 [50] 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.

2.2.5 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 $\pm 0.5\%$ 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. §2.3.5 explores this in more detail.

2.2.6 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 [38], 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 [2, 5, 60, 65, 66], 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 remains true).

2.3 Performance of Individual Mitigations

This section explores the individual mitigations that contributed 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.

2.3.1 Meltdown

Meltdown mitigations account for one of the most substantial performance impacts on LEBench, singlehandedly 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 evaulated, only Broadwell and Skylake are vulnerable to Meltdown.

As seen in Table 2.3, 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.

Vendor	CPU	`syscall`	`sysret`	swap `cr3`
Intel	Broadwell	49	40	206
	Skylake Client	42	42	191
	Cascade Lake	70	43	0
	Ice Lake Client	21	29	0
	Ice Lake Server	45	32	0
AMD	Zen	63	53	0
	Zen 2	53	46	0
	Zen 3	83	55	0

Table 2.3: Average cycles to execute a `syscall` or `sysret` instruction, and for vulnerable processors, to swap page tables.

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.

2.3.2 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 2.4 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.

2.3.3 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.

Vendor	CPU	Clear Cycles
Intel	Broadwell	610
	Skylake Client	518
	Cascade Lake	458
	Ice Lake Client	0
	Ice Lake Server	0
AMD	Zen	0
	Zen 2	0
	Zen 3	0

Table 2.4: Cycles required to clear microarchitectural buffers using the `verw` instruction. Processors not vulnerable to MDS are listed as zero cycles because they do not require any microarchitectural buffers to be cleared before returning to user space.

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.

The cycle cost of doing this MSR write on every system call was viewed as unacceptably high [56], so production operating systems investigated alternative approaches, ultimately settling on retpolines for any processor not supporting eIBRS.

Retpoline Retpolines are the primary software mitigation for Spectre V2 today. They 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 does not work on Intel: code using it would still be vulnerable to Spectre V2.

```
generic_retpoline:
    call 2f
1:  pause
    lfence
    jmp 1b
2:  mov %r11, (%rsp)
    ret

amd_retpoline:
    lfence
    call *%r11
```

Figure 2.4: Assembly sequences for the two kinds of retpolines

Table 2.5 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.

Vendor	CPU	Baseline	IBRS	Generic	AMD
Intel	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
AMD	Zen	30	N/A	+25	+28
	Zen 2	3	+13	+14	+0
	Zen 3	23	+19	+13	+18

Table 2.5: Baseline cycles to perform an indirect branch, and the added cost of doing indirect branches with IBRS enabled, the added cost of an indirect branch via a generic repoline, and via an AMD retpoline.

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 runs an Indirect Branch Prediction Barrier to clear the branch target buffer.

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. Oddly, 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 BTB to point to a specific harmless gadget rather than simply clearing them.

Table 2.6 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.

Vendor	CPU	IBPB cycles
Intel	Broadwell	5573
	Skylake Client	4537
	Cascade Lake	340
	Ice Lake Client	2455
	Ice Lake Server	836
AMD	Zen	7370
	Zen 2	1088
	Zen 3	808

Table 2.6: Cycles to execute an Indirect Branch Prediction Barrier.

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 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 2.7 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 2.7: Cycles to stuff the RSB.

Return stack buffer filling also provides protection against variations of SpectreRSB [33], 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.

2.3.4 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 2.8 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.

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

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

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, 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.

2.3.5 Speculative Store Bypass

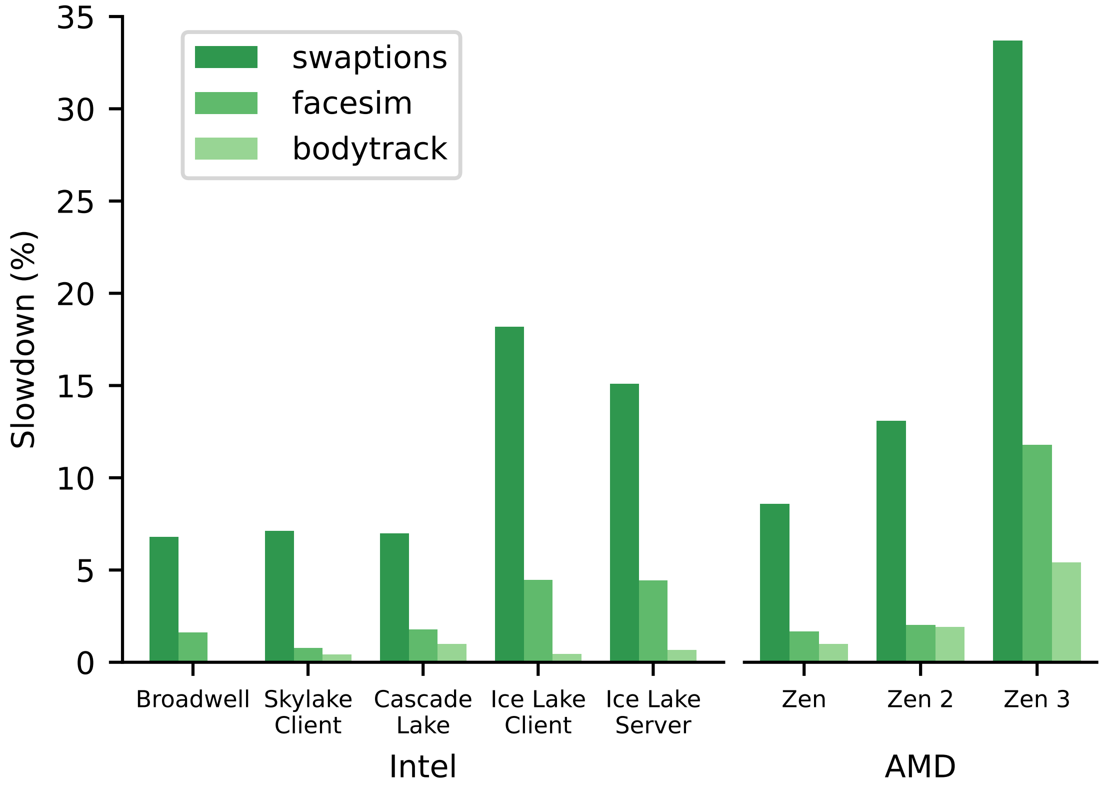

Figure 2.5: The slowdown caused by Speculative Store Bypass Disable on three benchmarks from the PARSEC suite.

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 the swaptions benchmark from PARSEC. Figure 2.5 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 this benchmarks is well under one percent (§2.2.5).

2.3.6 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.

2.3.7 Other attacks

The attacks discussed so far are hardly the only transient execution attacks discovered. Many others like System Register Read, 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.

2.4 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 a multitude of hardware and software mitigations, yet remains a non-trivial expense on every CPU we've tested. In this section we attempt to understand the attack better by determining under which conditions the Branch Target Buffer is used to speculatively execute instructions and when it is not.

```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 2.6: 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`.

2.4.1 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 [8] 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 explained next.

Processor 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 is active, and 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 2.6 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 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 both are 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. 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.

2.4.2 Results

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

Table 2.9: Whether the processor will speculatively execute an indirect branch in the given configuration when IBRS is disabled. A checkmark 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.

Table 2.9 and Table 2.10 show the results produced using this methodology. The columns indicate the mode that the attacker and victim run in respectively (e.g., u→k 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

		With intervening system call			No system call	
Vendor	**CPU**	u→k	u→u	k→k	u→u	k→k
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 2.10: Same as Table 2.9 but with IBRS *enabled*. IBRS always prevents problematic cases like u→k, but on many processors blocks all speculation including predicting the target of userspace indirect branches based on prior branches done by the same process (u→u).

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.

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 [26], 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 (reported in Table 2.10) revealed that 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.

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 2.10, Cascade Lake and the two Ice Lake processors (the microarchictures 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.) having 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.

2.4.3 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 or 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.

We suspect that the designers of eIBRS may have been aware of this risk and taken precautions against it. The documentation for eIBRS doesn't make any promises, but the slow kernel entries suggest that additional work is happening in connection with the feature.

2.5 Discussion

2.5.1 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 six week release cycle, any new hardware could be leveraged quickly.

2.5.2 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.

2.6 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 show up as causing 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.

Chapter 3

Ward

To address the severe performance overhead associated with OS level mitigations on older processors, we propose a new hardware/software contract, called the *unmapped speculation contract*, or USC for short. The USC allows the OS kernel to significantly reduce the overhead of mitigating a particular subset of transient execution attacks—namely, those that leak arbitrary memory contents. The USC says that physical memory that is unmapped (i.e., physical memory that has no virtual address) cannot be accessed speculatively. Although not specifically guaranteed by the x86 architecture, this property seems to be true even on pre-2018 CPUs and thus provides a theoretical baseline for what information transient execution attacks can leak on those processors. By bounding what data can be leaked, the USC can significantly reduce the cost of mitigations.

We have evidence that most processor models already adhere to the USC. AMD states that “AMD processors are designed to not speculate into memory that is not valid in the current virtual address memory range defined by the software defined page tables” [1, pg.

2], and Intel issued hardware and microcode fixes for bugs that violate USC [29, 30].

To demonstrate the benefits of the unmapped speculation contract, this thesis presents WARD, a novel kernel architecture that uses selective kernel memory mapping to avoid the costs of transient execution mitigations. WARD maintains separate kernel memory mappings for each process, and ensures that the memory mapped in the kernel of a process does not contain any data that must be kept secret from that process. As a result, privilege mode switches (e.g., system call entry and exit) no longer need to employ expensive mitigations, since there are no secrets that could be leaked by transient execution. When the WARD kernel must perform operations that require access to unmapped parts of kernel memory, such as opening a shared file or context-switching between processes, it explicitly changes kernel memory mappings, and invokes the same mitigation techniques used by the Linux kernel today.

A key challenge in the WARD design lies in re-architecting the kernel and its data structures to allow for per-process views of the kernel address space. For example, a typical `proc` structure in the kernel contains sensitive fields, such as the saved registers of that process, which should not be leaked to other processes. At the same time, every process must be able to invoke the scheduler, which in turn may need to traverse the list of `proc` structures on the run queue. We present several techniques to partition the kernel: transparent switching of kernel address spaces when accessing sensitive pages through page faults; using temporary mappings to access unmapped physical pages; splitting data structures into public and private parts; etc.

To evaluate the WARD design, we applied it to the sv6 research kernel [16] running on x86 processors. The sv6 kernel is a monolithic OS kernel written in C/C++, providing a

POSIX interface similar to (but far less sophisticated than) Linux. The simplicity of sv6 allowed us to quickly experiment with and iterate on WARD's design, since some aspects of WARD's design require global changes to the entire kernel. Since sv6 is a monolithic kernel, our prototype was able to tackle hard problems brought up by kernel services such as a file system and a POSIX virtual memory system.

We evaluate the performance of our WARD prototype using LEBench [49]. On the Broadwell CPU, WARD can run the LEBench microbenchmarks with small performance overheads compared to a kernel without mitigations. For 18 out of the 30 LEBench microbenchmarks, WARD's performance is within 5% of the benchmark's performance without any mitigations (but at the cost of some extra memory overhead). In the worst case, the overhead is 4.3× (context switching between processes, where mitigations are unavoidable). In contrast, standard mitigations incur a median overhead of 19% with a worst case of nearly 7×. To confirm that LEBench results translate into application performance improvements, we measured the performance of `git status`, which incurs 11.2% overhead in WARD, compared to 24.6% with standard mitigations.

One of the limitations of USC is that it does not cover all possible transient execution attacks. In particular, attacks where the sensitive information is already present in the architectural or microarchitectural state of the CPU are not covered by USC. For instance, the Spectre v3a attack can leak the sensitive contents of a system register (MSR), instead of leaking sensitive data from memory. USC does not cover sensitive data that is stored outside of memory, and WARD applies other mitigations (e.g., as in Linux) to address those attacks.

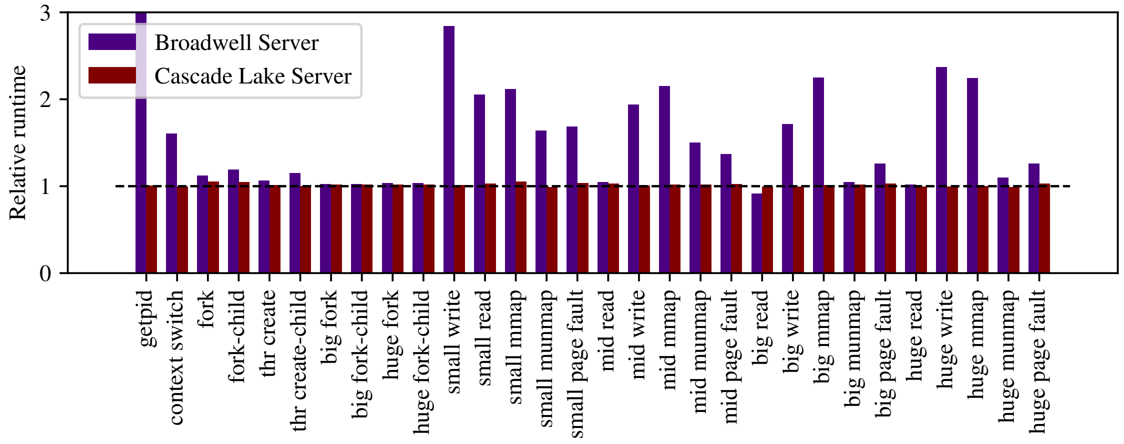

Figure 3.1: Linux slowdown due to mitigations on LEBench, for two generations of Intel CPUs: Broadwell and Cascade Lake.

3.1 Motivation

Transient execution mitigations harm kernel performance in two ways. First, they place overhead on code execution by disabling speculation, like how the Linux kernel uses retpolines to mitigate Spectre V2 [28]. Second, mitigations increase the privilege mode switching cost incurred during each system call: upon entry into the kernel, they either flush microarchitectural state or reconfigure protection mechanisms. For example, KPTI [21, 40] switches to a separate page table before executing kernel code to prevent Meltdown [41]. As described before, workloads that are system call intensive (e.g., web servers, version control systems, etc.) are impacted by this type of overhead, while compute-intensive workloads see little performance impact.

Collectively, these and other mitigations can result in large slowdowns. To better understand this problem, we revisit the LEBench [49] microbenchmark suite from earlier which consists of system calls that impact application performance the most, this time examining individual system calls instead of only the geometric mean over all of them. We evaluate the Linux kernel (version 5.6.13), comparing two configurations: one where all mitigations are disabled and one where all are enabled. Figure 3.1 shows the relative slowdown between the two configurations for 13 kernel operations of LEBench that don't involve networking (i.e., without `send`, `recv`, `epoll`). There are two sets of bars, representing two generations of Intel CPUs: the older Broadwell, and the newer Cascade Lake. On Broadwell, system calls that perform the least kernel work are impacted the most (e.g., `getpid`), but a wide range of operations are impacted significantly (25%-100% slowdowns). These observations are similar to what we saw in §2.2.2 and to the observations made by Ren et. al. [49].

3.2 Goals

WARD's goal is to reduce the performance cost of mitigations for transient execution attacks. In principle, WARD's techniques can reduce not only the cost of software mitigations, but also allow processor designers to avoid mitigations in hardware. Practically, however, WARD is most applicable to older Intel processors, which incur the largest costs on OS-level mitigations.

Transient execution attacks can leak data across many protection domain boundaries, including leaking secrets from the kernel to an adversary's process, or leaking secrets from one process to a different process, or even leaking secrets within a single process that implements its own internal protection domains. Much like in the Linux kernel, the focus of WARD is on preventing leakage between processes, as well as preventing leakage from the kernel to a process. WARD's approach to preventing cross-process leakage is the same as Linux (flushing state), but WARD has a novel approach for efficiently preventing kernel-to-process leakage of memory contents, as we describe in the next section.

Although WARD addresses all known transient execution attacks, the focus here is on attacks that allow the adversary to leak the contents of arbitrary memory, which is especially important in an OS kernel. WARD handles other transient execution attacks, such as leaking the contents of sensitive data already present in the CPU (e.g., x86 MSRs), in the same way as Linux does.

Attacks that do not leverage transient execution to leak data are also out of scope, since they are orthogonal to the key challenge of transient execution leakage. In particular, we do not consider attacks that leverage physical side channels (such as Rowhammer or RAMbleed), cache side channels (such as cache timing attacks), interrupt side channels, power side channels, etc.

3.3 Approach: Unmapped speculation contract

WARD's design for mitigating transient execution attacks relies on page tables. Specifically, if a page of physical memory is not referenced by any entry in the current page table or TLB, speculative execution cannot access any sensitive data stored in that page, because the page doesn't have a virtual address to access it by.

A contribution of this thesis lies in articulating a hardware/software contract—which we call the *unmapped speculation contract*—that captures the above intuition. The contract aims to provide a strong foundation for keeping data confidential, which is typically stated as non-interference. Non-interference can be thought of by considering two system states, s

and s', that differ only in sensitive data, which should not be observable by an adversary. A system ensures non-interference if an adversary cannot observe any differences in how the system executes starting from either s or s'.

Single-core USC. To formally state the unmapped speculation contract, we start with a single-core definition. We use $A(\cdot)$ to refer to the state of the CPU, including all architectural and micro-architectural state, but excluding the contents of memory, and we use $M(\cdot)$ to refer to the contents of *mapped* memory, i.e., the contents of every valid virtual address based on the committed page table in that state. We define the contract by considering a single clock cycle of the processor's execution, step$(\cdot)$, which includes any speculative execution done by the processor on that cycle, and require that unmapped pages cannot influence it:

$\forall s, s'$,
if $A(s) = A(s')$ and $M(s) = M(s')$,
then with $S :=$ step(s) and $S' :=$ step(s'),
it must be that $A(S) = A(S')$

In plain English, the definition considers a pair of starting states s and s' that should look the same, as far as speculative execution is concerned, because they have the same CPU state and the same contents of mapped pages. They might, however, differ in the contents of some unmapped physical pages, which contain sensitive data that we would like to avoid leaking. The definition then considers the state of the CPU at the next clock cycle ($S :=$ step(s) and $S' :=$ step(s') respectively), and requires that the CPU architectural and micro-architectural state $A(\cdot)$, which the adversary might observe, continues to be the same in those two states. As a result, the microarchitectural state could not have been influenced by any sensitive data not present in $M(s)$.

If the OS kernel does not change the mapped memory in that clock cycle, $M(\cdot)$ remains the same, and the contract will continue to hold on the next cycle too. However, if the OS kernel changes the mapped memory, the contract allows speculative execution from that point on to use the newly mapped memory, and the kernel will need to use other mitigations to defend against transient execution leaks from the newly mapped memory, if necessary.

The contract specifies how the micro-architectural state, $A(\cdot)$, can evolve, but does not say anything about how $M(\cdot)$ can change. This is because the focus of the contract is on transient execution, which cannot affect the committed architectural state of the system; the contents of memory is described by the ISA, since it is architectural state. In other words, changing the memory requires committing the execution of some instruction, at which point this is no longer a transient execution.

Multi-core USC. In a multi-core setting, the CPU state can be thought of as consisting of per-core state (e.g., registers, execution pipeline, and root page table pointer), which we denote with $A_i(\cdot)$ for core i, and the uncore state (e.g., the hardware random number generator [46]), which we denote with $U(\cdot)$, shared by all cores. Similarly, since each core has its own page table, we index the mapped memory by the core i whose page tables we are considering, $M_i(\cdot)$. Finally, we consider the multi-core system executing a clock cycle on one core at a time, $\text{step}_i(\cdot)$. We assume that $\text{step}_i(\cdot)$ does not change $A_j(\cdot)$ for any $i \neq j$. With this notation, the multi-core contract says:

$\forall s, s', i,$

if $A_i(s) = A_i(s'); U(s) = U(s')$; and $M_i(s) = M_i(s')$,

then with $S := \text{step}_i(s)$ and $S' := \text{step}_i(s')$,

it must be that $A_i(S) = A_i(S')$ and $U(S) = U(S')$

This means that speculative execution on core i is allowed to depend on the state of core i, the uncore state, and the memory mapped by core i. This multi-core formulation allows transient execution to affect both the core state $A_i(\cdot)$ as well as the uncore state $U(\cdot)$, at the micro-architectural level. However, transient execution cannot affect either of these states in a way that depends on unmapped memory.

Although hardware threads appear to provide separate execution contexts, with a separate page table for each hardware thread, they have extensive sharing of core resources. To capture that, we consider $A_i(\cdot)$ to include the state of all hardware threads on core i, $\text{step}_i(\cdot)$ to include the execution of any hardware thread on core i, and $M_i(\cdot)$ to be the union of memory mapped by all of the hardware threads on core i (i.e., the union of the page tables of the threads). With this model, the contract allows leakage of mapped memory across hardware threads.

Benefits of the USC. The contract helps reconcile security and performance of speculative execution. It enables software to precisely specify what data can and cannot be used for speculative execution, by configuring page tables. For example, if the mapped pages never contain sensitive data, then no mitigations are needed to defend against transient execution vulnerabilities. Finally, because OS developers expect page faults and TLB misses to be quite expensive (compared to memory references), USC doesn't change their performance expectations: developers already have adapted their designs to avoid excessive page faults or TLB invalidations.

AMD explicitly states in bold font that their “processors are designed to not speculate into memory that is not valid in the current virtual address memory range defined by the software defined page tables” [1, pg. 2]. Intel has no explicit position about this contract, but it appears that they treat violations of this contract as bugs to be fixed in hardware or microcode, as evidenced by their fixes for Meltdown and L1TF, described below.

USC and attacks. The contract captures a common pattern that emerges in many transient execution attacks: an adversary can only leak micro-architectural state that is already present on the CPU, as well as the contents of mapped memory, but not the contents of memory that is not present in a page table. As one example, consider the MDS family of attacks [12, 51, 57]. These attacks allow an adversary to trick the kernel into leaking the contents of mapped memory, through careful orchestration of transient execution. Linux prevents this class of attacks by clearing CPU buffers when crossing the user-kernel boundary. This is needed because, when executing in kernel mode, all system memory is mapped and therefore could be leaked through transient execution. The contract, however, captures the fact that only mapped memory is at risk with this attack. This allows for a more efficient mitigation of such attacks by avoiding kernel mappings of sensitive memory, as we demonstrate with WARD

In contrast to the example of MDS attacks, which leak sensitive data from memory, the USC does not help mitigate attacks that leak sensitive data already present in the CPU state. For instance, the Spectre variant that leaks the contents of x86 MSRs (Spectre 3a) is not precluded by the contract, since the sensitive data being leaked is not present in memory at all. As a result, an OS kernel must apply other mitigations to deal with such attacks.

Attack	**Leaked state**	**Mitigated by USC**	**Consistent with USC**
Spectre variants	Memory	Yes	Yes
Meltdown		Yes	Yes (depending on PTE contents)
MDS		Yes	Yes
PortSmash		Yes	Yes
L1TF		Yes	Yes (depending on PTE contents)
Spectre variants	Core state	No	Yes
LazyFP		No	Yes
System reg. read		No	Yes
Spectre variants	Uncore state	No	Yes
CrossTalk		No	Yes
SGAxe		No	Yes

Figure 3.2: Transient execution attacks categorized based on the state leaked by the attack.

More generally, the contract helps categorize existing attacks based on which part of the system state they leak, as shown in Figure 3.2. For attacks that leak core or uncore state, the contract has little to say in terms of how those attacks can be mitigated, as shown in the "Mitigated by USC" column. As a result, WARD defends against these attacks much in the same way as Linux. In contrast, for attacks that leak the contents of memory, the contract gives a more efficient mitigation approach: simply avoid mapping memory that contains sensitive data. This allows WARD to efficiently mitigate attacks such as some variants of Spectre and MDS.

As shown in the "Consistent with USC" column, all of the attacks in Figure 3.2 are consistent with the contract's requirements on the underlying hardware. This is good in two ways. First off, this means that none of the known attacks violate the contract, and thus, the contract is a reasonable approach for mitigating transient execution attacks. Second, this means that USC can mitigate the class of attacks that it covers—namely, attacks that leak memory contents.

There are two special cases: Meltdown and L1TF. When originally discovered, these attacks bypassed the page table protections and allowed an adversary to obtain the contents of memory that was not mapped. Subsequent guidance clarified that the USC still holds in these cases provided that software avoids certain bit patterns in page table entries. In both of these cases, the hardware manufacturer (Intel) considered them to be hardware bugs, as evidenced by the fact that both of them were fixed in subsequent CPU generations [29, 30], as confirmed by Canella et al. [11].[1]

3.4 Design

Under the assumption of the unmapped speculation contract, this section describes how WARD can reduce the cost of mitigations for system calls. §3.4.1 provides an overview of WARD's design with subsequent sections providing more detail about WARD's switch between protection domains (§3.4.2), about the mitigations used by WARD when mitigations are necessary (§3.4.3), WARD's kernel text (§3.4.4), WARD's memory management modifications (§3.4.5), WARD's process management split (§3.4.6), and WARD's file system split (§3.4.7).

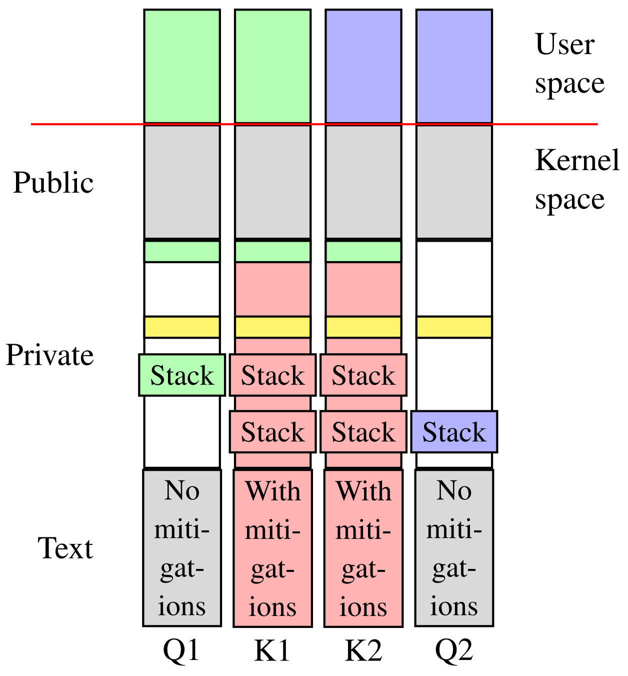

Figure 3.3: Overview of WARD's address space layout with two processes (indicated by the colors green and purple). Each process has a Q and K domain. Q domains have access to public data (the grey color) and per-process kernel data; the white private region is unmapped kernel data. Each domain also has its own stack and kernel text. In the Q domain, the kernel text has no mitigations. The K domains map all memory, including sensitive memory (indicated by red); all K domains have the same memory layout. Data structures that are shared across processes, such as pipes or file pages, can be mapped in multiple Q domains, as indicated by the yellow color.

3.4.1 Overview

WARD's design maintains two page tables per process. One page table defines a process-specific view of kernel memory. When a process is running with that page table, we say

[1]Canella et al. state that some variants of the Meltdown attack, such as Meltdown-BR, are still possible even with the most recent microcode. Those variants, however, are bypassing software checks, rather than the hardware page table, and therefore do not violate the unmapped speculation contract.

it is running in its *quasi-visible* domain (or Q domain for short), and with its Q page table. Following the unmapped speculation contract, WARD assumes any kernel memory mapped by the Q page table can be leaked to the currently running process. Instead of using mitigations to prevent leaks of kernel memory, WARD arranges for the mappings in the Q page table to be such that they contain no sensitive data of other processes.

When the process needs to access data that is not mapped in the Q page table, it can switch to its other page table, which maps all physical memory, including memory that contains sensitive data. When a process is running with this page table, we say the process is executing in its *K* domain with its K page table. In its K domain, the process runs with the same mitigations as Linux currently uses.

This design allows many system calls to execute in the Q domain, with no mitigation overhead. As a simple example, `getpid` does not access any sensitive data; it needs access to only the kernel text and its own process structure. A more interesting example is mapping anonymous memory: this requires access to the process's own page table and to the memory allocator, but not other processes' page tables or pages.

Figure 3.3 shows the address space layout in WARD in more detail. Each process has a Q and K view of memory. When a process is running in user space it runs in its Q domain (with no secrets mapped in the Q page table). When a user process makes a system call it enters the kernel but stays in its Q domain. The Q domain maps public kernel memory, Q-visible kernel memory, the process's Q domain stack, and the kernel text without mitigations.

If a system call needs access to memory in the K domain, WARD performs a switch from its Q domain to its K domain. We refer to the switch from a Q domain to a K domain as a *world switch*, because kernel code in a Q domain runs without most mitigations and the kernel code in the K domain runs with full mitigations. Furthermore, the process switches from its Q domain stack to its K domain stack. The K domain, with access to all kernel memory, can then execute the rest of the system call with full mitigations.

Achieving good performance in WARD depends on avoiding world switches. To reduce the number of world switches, WARD maps kernel data structures that contain no sensitive data into every Q domain. For example, all Q domains map x86 configuration tables (IDT, GDT), some memory allocator state, etc. On the other hand, kernel data structures that contain application data, such as process memory or saved register state, are not mapped into Q domains unless that process should have access to that data.

3.4.2 World switch

One of the challenges in WARD's design is that a system call often does not know upfront whether it will need to execute in the Q domain or in the K domain. For example, a `read` system call might be able to execute purely in the Q domain, or might need access to sensitive data from the K domain, depending on the file descriptor that the process is reading from, and depending on whether this Q domain already has some sensitive data mapped or not. To support this, WARD's design allows a system call to start executing in the Q domain, and switch to the K domain later as needed.

WARD allows the Q domain to trigger a world switch either *intentionally* or *transparently*. If the code determines that it must switch to the K domain, it can intentionally invoke the function, `kswitch()`, to perform a world switch. When `kswitch()` returns, the kernel thread is now executing in the K domain, and has access to all memory. If the Q domain needs access to specific sensitive data which might or might not be already mapped, the Q domain can attempt to access the virtual address of that data. If the data is already mapped in the Q domain, the access will succeed, no world switch happens, and the Q domain can continue executing. If the data is not mapped, the Q domain triggers a page fault, which transparently triggers a world switch. Once the page fault returns, the kernel thread is now executing in the K domain, as if it called `kswitch()`. Compared to making an intentional call to `kswitch()`, the transparent approach incurs a slight overhead for executing the page fault, but allows large sections of the kernel to be kept completely unmodified, and allows the Q domain to elide a world switch altogether if the data is already mapped in the Q domain.

The above design requires that a kernel thread can start executing in the Q domain and transparently switch to executing in the K domain. This means that any addresses that the kernel thread is referencing, including pointers to data structures, stack addresses, and function pointers, remain the same. To achieve this, WARD ensures that the layout of the Q domain and the K domain match. In particular, all data structures in the Q domain must appear at the same address in the K domain, and the kernel code (text) is located at the same address (even though the code is slightly different, as described in §3.4.4).

The stack requires particular care because a kernel thread that is processing sensitive data in the K domain could inadvertently write that data to the stack. For example, a `read()` system call from `/dev/random` needs to switch to the K domain to access the system-wide randomness pool. However, the pseudo-random generator code might spill some of its state to the stack, depending on the compiler's choices. If the stack is accessible from the Q domain, the sensitive data could in turn be leaked during the next entry into the Q

Transient execution vulnerability	**User/Q domain**	**K domain**	**Context Switch**
L1TF	x	x	
Spectre V1		x	
Bounds Check Bypass Store		x	
Meltdown		x	
Speculative Store Bypass		x	
Spectre V2		x	x
Microarchitectural Data Sampling (Fallout, RIDL, Zombie Load, etc.)		x	x
LazyFP			x
SpectreRSB			x
PortSmash		Not applicable	
Load Value Injection		Not applicable	
Meltdown-PK (protection key bypass)		Not applicable	
Meltdown-BR (bounds check instr. bypass)		Not applicable	
Read-only Protection Bypass		Not applicable	

Figure 3.4: The mitigations implemented in software by WARD.

domain by any thread within the same process. At the same time, if the K domain stack was separate from the Q domain stack, pointers to stack locations before a world switch would no longer work after a world switch. To reconcile these constraints without having to rely on any dedicated compiler support, WARD maps a distinct kernel stack for each domain at the virtual address range and copies the Q domain stack contents to the K domain stack during a world switch.

3.4.3 Mitigations

Figure 3.4 shows the known transient execution attacks [11, 22], organized by the mitigations needed to address those attacks in WARD's design. The columns indicate where the mitigations are needed: while executing in user-mode or the Q domain; while executing in the K domain; and when context-switching between processes.

The L1TF attack allows leaking the contents of the L1 cache if there are partially-filled-in entries in the page table. We think of this attack as a violation of the USC (see Figure 3.2), but a simple microcode fix, as well as clearing unused page table entries, makes the system agree with the USC, and avoids the L1TF attack. Since L1TF allows leaking the contents of any data, WARD applies the mitigations both in user-space, Q domain, and K domain.

The next category of attacks requires no mitigations in either user-space or Q domain. Specifically, Spectre variants that bypass bounds checks require mitigation in the K domain, since there is sensitive memory contents that could be leaked as a result of a speculative check bypass. However, there is no sensitive data that can be leaked in the Q domain, owing to USC. Similarly, no mitigations are required on a context switch, since these attacks can only leak data from the current protection domain.

Meltdown also falls in this category, but for a different reason. Meltdown allows an adversary to bypass the user-kernel boundary check in the page table. WARD's use of a separate page table for the Q and K domains ensures that Meltdown cannot leak any confidential data, since no confidential data is available in the Q domain. Recent fixes from Intel resolve the Meltdown attack in a way that avoids the need for software mitigations.

The next category of attacks require mitigation both in the K domain and on context switch. Both Spectre V2 and MDS can allow an adversary to obtain sensitive data either from the OS kernel or from another process. However, no mitigations for these attacks are needed in the Q domain due to USC: there is no sensitive data to leak in the Q domain of the currently running process.

For some attacks, such as LazyFP and SpectreRSB, mitigations are only required on context switch, because the attacks involve process-to-process leakage.

Finally, a number of attacks are not applicable to WARD's simpler design, in contrast to Linux. For example, WARD does not support SGX, does not support running virtual machines, and does not use certain hardware features (such as hardware bounds-check instructions or protection keys).

3.4.4 Kernel text

Some of the mitigations involve changes to the executable kernel code (text), such as the use of retpolines in place of indirect jumps. These mitigations impose a performance cost, but they are not needed when executing in the Q domain.

A naïve approach might be to compile the kernel code twice, with different compiler flags for mitigations, and load the two different kernel binaries in the Q and K domains respectively. However, this would break WARD's page fault triggered world switches because after completing the switch, execution would resume with the same instruction pointer and stack contents from before the switch but neither would be meaningful in the new text segment.

Instead we need the two version to have matching instruction addresses and stack layouts. WARD achieves this by compiling the kernel only once, but then making two copies of the code at runtime. One copy is mapped into all the K domains, and the other into all the Q domains *but at the same virtual address as in the K domains*. Switching between the two is seamless.

At boot time, in a process inspired by Linux's ALTERNATIVE macro [17], WARD locates each `call` or `jmp` in the Q text segment pointing to a retpoline thunk, and replaces them with the instruction that retpoline emulates. One complication is that indirect call instructions are only 2 or 3 bytes, compared to the 5 that a direct call instruction takes. If we tried to pad with a `NOP` instruction before or after, the pair would not execute atomically, so instead we prepend indirect calls with several repetitions of the CS-segment-override prefix, which is always ignored in 64-bit mode.

3.4.5 Memory management

Memory allocation in WARD is complicated by the fact that the contents of free pages may contain sensitive data. In particular, if a page was freed by one process, its contents must be erased before the page can be mapped in another Q domain. Zeroing out pages on every allocation would be costly, in particular when allocating kernel data structures, which do not otherwise require the memory to be zero-filled.

To avoid the overhead of repeatedly zeroing kernel pages, WARD implements a sharded allocator for kernel memory. Each Q domain has its own pool of pages for allocation, and the K domain keeps all of the kernel memory that is not part of any Q domain. WARD transfers memory between these shards in batches to amortize the world switch overhead. Keeping a pool of kernel memory in a Q domain allows the kernel to repeatedly allocate and free memory within a Q domain with little overhead.

The other category of memory managed specially by WARD is public memory. WARD maintains a single pool of public pages, with separate functions, `palloc()` and `pfree()`, for allocating and freeing in that pool. All public-pool pages are mapped in every Q domain.

3.4.6 Process management

When the WARD kernel switches from executing one process to another, it must perform a world switch, to ensure that confidential data does not leak across processes (such as the saved CPU registers that the kernel might keep on the stack). However, if a multi-threaded application is running, there is no security reason to perform a world switch when switching between multiple threads in the same process—all of the threads have the same privileges and have access to the same process address space.

To avoid mitigation overhead when switching between threads in the same process, WARD splits the process descriptor, `struct proc`, into two parts. The first part stores sensitive process state, such as the saved CPU registers, and is not public. The second part stores metadata about the process, such as the PID, the run queue, the scheduler state, etc. This part is public and is used by the scheduler when deciding what thread to execute next. As a result, the scheduler can pick the next thread without incurring a world switch. Furthermore, if the next thread happens to be from the same process, the context switch code can also avoid performing a world switch. Existing scheduler policies that favor picking threads from the same process mesh well with this approach.

3.4.7 File system

File system workloads involve access to several kernel data structures, including the inode cache and the page cache (containing file data). Inodes are challenging for WARD to deal with because they are smaller than a page, so it is not feasible to map them individually into a Q domain. However, achieving good performance for file system operations requires being able to access an inode without a world switch. To reconcile this conflict, we chose to make all inode structures public in WARD, similar to our approach for splitting the `proc` structure above. If the inode had sensitive data (such as extended attributes), that part of the inode structure would need to be split off into a separate private structure, along the lines of how we split off the part of the `proc` structure storing saved CPU registers.

File data pages are not public, because their contents might be sensitive. WARD implements an optimization that allows it to access file contents without a world switch. In particular, after WARD checks the permissions on a file, it reads or writes the contents of a file page by temporarily mapping the corresponding physical page of memory into its Q domain's address space. This allows the Q domain to access that specific memory page without the risk of leaking other pages; as a result, no mitigations or world switches are needed. When the Q domain is finished with the file read or write, it unmaps the page and issues a TLB shootdown, in case the file is later truncated and the page gets reused for other data.

3.4.8 Pipes

Pipes are different from many of the other kernel data structures discussed so far in that their contents shouldn't be visible globally, but their state can be associated with multiple processes at a time. WARD's goal is to ensure that if a reader and writer of a pipe run on different cores, then they don't incur world switches when they access the pipe. To achieve this, we store a pipe's data structures in shared memory regions between Q domains. These shared regions are lazily mapped into Q domains the first time a process accesses a pipe (doing the mapping on `fork` would cause unnecessary overhead), and unmapped when the last reference to the pipe within a Q domain is closed.

When a pipe becomes full or empty, the caller blocks on a condition variable. Subsequent reads or writes can observe which processes are blocked and add them to the scheduler run queue if appropriate. Neither of these operations requires access to any secret data so no world switch is triggered until a new process is scheduled. Thus, if the core remains idle until the blocking thread is added back to the run queue, the cost of a world switch is avoided.

3.4.9 Discussion

WARD's design assumes that there are no secrets in the Q domain that need to be hidden from the user-level process. For many secrets, they can be protected by placing them in the K domain, such as the seed of a system-wide randomness generator. However, address-space layout randomization (ASLR) for the kernel address space is difficult to protect in this fashion, because kernel addresses must be used in the Q domain, and the addresses must match up between the Q domain and the K domain in order for world switches to work. (Note that the initial seed that is used to randomize layout could be protected in the K domain, but the resulting randomized layout cannot be protected.) As a result, kernel ASLR in WARD is susceptible to leakage of addresses through transient execution side-channels.

Our WARD prototype does not include an optimized in-kernel network stack, but a reasonable approach might be to treat all network data as public, leaving it up to the application to encrypt any sensitive information sent over the network. This meshes well with the recent trends in widespread use of TLS for network security, and allows for network operations to achieve high performance in WARD because no mitigations or world switches are required, and all network processing can stay in the Q domain.

Transient execution variant	**Strategy**	**Support**
Spectre V1	bounds clipping	partial
Bounds Check Bypass Store	lfence	partial
Read-only Protection Bypass	lfence	n/a (no kernel sandbox)
Spectre V2	retpoline	yes
"	speculation barrier	yes
"	return stack buffer filling	yes
"	set IBRS before BIOS call	n/a (no BIOS calls)
Meltdown	Kernel page table isolation	yes
System Register Read	microcode	yes
Speculative Store Bypass	disable spec. or ctx. switch	yes
LazyFP	hardware FP save/restore	yes
SpectreRSB	return stack buffer filling	yes
L1TF	cache flush, no SMT	n/a (no VM entry)
"	no invalid PTEs	yes
PortSmash	no SMT	no
Microarchitectural Data Sampling	CPU buffer clearing	yes
(Fallout, RIDL, Zombie Load, etc.)	no SMT	no
Load Value Injection	lfence	n/a (no SGX in WARD)
Meltdown-PK (protection key bypass)	address space isolation	n/a (no protection keys)
Meltdown-BR (bounds check instr.)	lfence	n/a (no MPX instrs.)

Figure 3.5: Transient execution mitigations implemented in WARD.

Hyperthreading is a source of many possible transient execution leaks, because a significant amount of microarchitectural state is shared between the execution contexts. However, many Linux systems continue to run with hyperthreading *enabled*, despite these risks, because of the high performance overhead they would incur if hyperthreading was entirely disabled. WARD does the same.

3.5 Implementation

To demonstrate the feasibility of the WARD design, we implemented a prototype of WARD starting from the sv6 research kernel. The kernel is monolithic, implementing traditional

OS services such as virtual memory, processes and threads, file systems, fine-grained concurrency using RCU-like techniques, etc. The sv6 kernel, is written in C/C++, runs on x86 processors (both AMD and Intel), and has decent uniprocessor performance and great multicore performance and scalability [16].

Kernel changes. WARD's design affects most core kernel subsystems, including the memory allocator, virtual memory, context switching and the scheduler, and the file system. The simplicity of sv6 allowed for rapid experimentation with kernel designs to enable WARD, which would have been challenging to do in a more complex kernel like Linux, since it is time-consuming to make changes to core subsystems in the Linux kernel, which would have made design iterations far slower.

To help partition the kernel data structures across Q domains, we developed Warden, a tool for tracking down the cause of world switches. Warden instruments page faults from the Q domain that lead to a world switch, and records a stack trace for each of them. Examining the profile of these world switches allows the kernel developer to quickly understand what kernel data structures need to be partitioned or sharded to reduce the number of world switches, as well as the operations that need to be supported on these data structures within a Q domain. Although Warden identifies the data structures that are causing world switches, it is up to the kernel developer to identify an appropriate plan for partitioning the data structure so that no sensitive data can leak through side channels.

To run applications on top of the WARD prototype kernel, we changed the WARD system call interface, including system call numbers, data structure layout, etc, to match that of Linux. This allows unmodified Linux ELF executables to run on top of WARD, and ensures that WARD implements (a subset of) the same system calls that are available on

Linux.

We modified sv6 to use PCIDs to reduce the cost of switching page tables (see §3.4.2). To improve TLB shootdown performance, we modified sv6 to use Linux's shootdown strategy. This is important, for example, for removing temporary mappings in a `read` and `write` systems calls (see §3.4.7).

Mitigations. WARD implements side-channel mitigations for known transient execution attacks [11, 22], as shown in Figure 3.5. WARD mostly copies the mitigation strategies and their implementation from the Linux kernel [39]; the most interesting exception is that WARD does not apply some of these mitigations to the Q domain, as described in Figure 3.4.

For Spectre V1, WARD, adds an `lfence` instruction when copying from user code, and when taking an interrupt, exception, and NMI entry. WARD uses bounds clipping in fewer cases than Linux for two reasons: WARD has less code and we haven't performed a careful audit of the complete source code. For Spectre V2, we compile WARD to use retpolines (by specifying the "-mretpoline-external-thunk" flag to `clang`). WARD also uses Linux's `FILL_RETURN_BUFFER` macro to fill the return stack buffer, and issues an indirect branch predictor barrier `IBPB` instruction on a context switch. For Meltdown, WARD uses separate page tables (as described in §3.4.1) and uses process-context identifiers (`PCID`s) to avoid TLB flushes.

For Spectre V4, WARD issues an `lfence` on context switch. (If WARD supported generating code at runtime, the JITs would also have to be hardened.) For LazyFP, WARD uses the `xsaveopt` instruction to safe/restore floating point state. For SpectreRSB, WARD fills the return stack buffer on context switch. For L1TF, WARD avoids invalid PTEs. Like

Linux, WARD doesn't address PortSmash; the default for the Linux kernel is to allow SMT, and WARD does too. For microarchitectural data sampling attacks, WARD issues the `verw` instruction for clearing CPU buffers.

Some attacks aren't applicable to WARD, because WARD doesn't support virtualization, secure enclaves, and hardware transactional memory; does not call into the BIOS; and does not implement in-kernel software sandboxes such as BPF.

Like Linux, WARD also zeroes unused CPU registers on kernel entry, to reduce the avenues of attack available to an adversary. To determine whether mitigations are necessary, WARD maintains a special variable called `secrets_mapped` whose value is 0 in the Q domain and 1 in the K domain; this allows the rest of the kernel code to determine if it needs to perform mitigations just by using `if (secrets_mapped) ...` (as long as interrupts are disabled, to avoid races). To help evaluate the performance impact of side-channel mitigations, WARD's implementation allows switching individual mitigations on and off at runtime, rather than at compile time or boot time.

To improve performance, a few system calls invoke the world switch intentionally to avoid the extra overhead of a transparent world switch. For example, `open`, and `fork` always invoke world switch intentionally. The `read` and `write` system calls invoke a world switch intentionally when they are reading or writing large amounts of data, since the cost of a world switch is less than the cost of shooting down the temporary mappings for that many file pages. A page fault on a Copy-On-Write (COW) page also intentionally invokes a world switch.

Lines of code. The WARD prototype consists of about 34,000 lines of C++ code (for `kernel/` and `include/`), compared to 24,000 lines of C++ code for the sv6 kernel that

WARD was derived from. `git diff -stat` reports roughly 17,000 lines of insertions and 5,000 lines of deletions between sv6 and WARD. It is difficult to further break down WARD's lines of code, since many aspects of WARD's design required small changes throughout the kernel's source code. For example, splitting up the kernel memory allocator required the use of C++ placement `new` in many parts of the kernel. Similarly, implementing the Linux binary compatibility layer required making changes to the implementation of many system calls.

3.6 Evaluation

To demonstrate the benefits of WARD's design, this section answers the following questions:

- Do WARD's techniques reduce the overhead of mitigations for system calls? (§3.6.2)
- How do mitigations affect the cost of a world switch? (§3.6.3)
- What are the memory overhead associated with WARD's design? (§3.6.4)

3.6.1 Experimental methodology

To answer these questions, we consider three different configurations of WARD:

- Baseline: WARD with no mitigations against transient execution attacks.
- Linux-style: WARD with standard mitigations against transient execution attacks, mirroring the approach taken by the Linux kernel. This configuration does not use separate Q domains; all system calls directly enter the K domain.

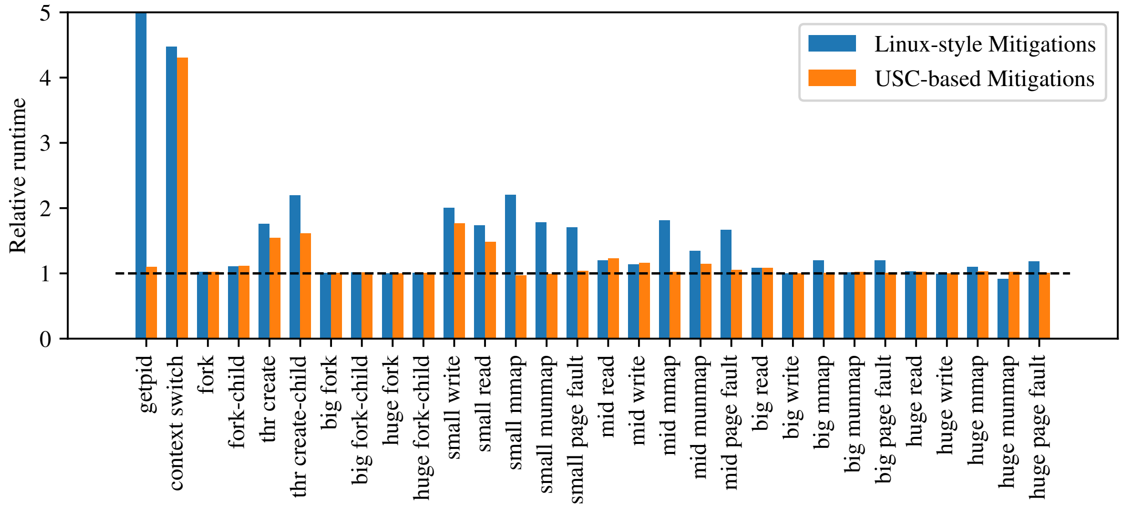

Figure 3.6: Performance of WARD with fast USC-based mitigations and with Linux-style mitigations, normalized against the baseline performance of WARD without any mitigations.

- USC-based: WARD with fast mitigations that take advantage of the split between the Q domain and the K domain, leveraging the USC. The K domain implements the same mitigations as in Linux-style.

WARD's design is aimed at reducing the overhead of mitigations associated with system calls. To zoom in on the system call overhead, we evaluate WARD's performance using LEBench [49]. This allows us to precisely report and explain the effect of WARD's techniques on individual system calls. We don't report results for the networking benchmarks in LEBench, because the WARD prototype doesn't have a suitable in-kernel network stack.

All benchmarks were run on a Dell PowerEdge T430 with two E5-2640 v4 CPUs and 64 GB of RAM.

One potential concern with the use of recent microcode is that it makes the baseline slower, which in turn makes the cost of mitigations appear lower than they really are.

This is similar to the significant effect we observed with newer CPUs, as described in §3.1. However, with newer microcode, we find that the performance of the baseline is not significantly affected: it achieves similar performance even when we use old microcode. The reason for this is that the recent microcode updates add mitigations that can be specifically enabled (e.g., through the `SPEC_CTRL` MSR), but almost nothing is enabled by default. The Linux and Ward baseline experiments do not enable these mitigations, and thus the performance effect is minimal.

For the Linux measurements of LEBench, we use the 5.4.0 kernel on Ubuntu 20.04.

3.6.2 WARD's USC-based fast mitigations

LEBench. Figure 3.6 shows the benefit of WARD's fast mitigations on LEBench. The figure compares WARD with USC-based and Linux-style mitigations, relative to the baseline with no mitigations. As shown, WARD with fast USC-based mitigations is often able to match the unmitigated baseline. The reason is that many of the microbenchmarks can execute with no or very few world switches, as shown in Figure 3.7.

Many microbenchmarks (`getpid` through `huge pagefault` in Figure 3.7) have nearly 0 transparent and intentional world switches. They execute completely in the Q domain. The reason that some have near 0 world switches, but not exactly 0, is that during the measurement they were interrupted by a timer interrupt, which requires a world switch to the K domain to run the scheduler (the remainder of the syscall is then executed in the K domain too).

Another cause for fractional numbers of transparent world barriers is that some operations might have a slow path that requires secrets but only gets triggered infrequently (i.e. because a memory allocator pool ran empty). A strength of the WARD approach is that these sorts of cases don't have to be manually annotated and in fact it is harmless to completely ignore them provided they are executed infrequently enough.

There are several microbenchmarks (e.g., the bigger `read` and `write` ones) that perform one intentional world switch per system call. These system calls immediately enter the K domain and thus perform identical to WARD with full mitigations, and have the same overhead. These system calls also perform much work in the kernel and the overhead of the one world switch is amortized by that work.

The `thr create` and `thr create-child` do multiple syscalls per iteration, but average one world barrier per iteration. Specifically, the `thr create` microbenchmark makes three systems calls: one `clone` that requires a world switch and a call to each of `sigprocmask` and `set_robust_list` which don't. The `thr create-child` microbenchmark includes an additional call to (`sigprocmask`) from the child process, for which WARD can also avoid the world switch.

The `fork` and `fork-child` benchmarks each do a single syscall with an intentional world barrier that takes the vast majority of execution time, but also raise a handful of page faults to populate page table entries (which need secrets if they are copy-on-write related or if the kernel runs out of zeroed memory pages and has to prepare more).

An interesting case is the `context switch` microbenchmark. This microbenchmark measures context switching by writing and reading a byte over a pipe between two processed pinned to the *same* core. The `write` calls avoids a world switch because the scheduler can wake other processes while in the Q domain, but the `read` call causes a context switch and (since the two processes are mutually distrusting) thus requires a world switch.

	# sys calls	**World switches**		
		T	**I**	**Sum**
getpid	1	0	0	0
small write	1	0	0	0
small read	1	0	0	0
small mmap	1	0	0	0
small munmap	1	0	0	0
small page fault	1	0	0	0
mid mmap	1	0	0	0
mid munmap	1	0	0	0
mid page fault	1	0	0	0
big mmap	1	0	0	0
big page fault	1	0	0	0
huge mmap	1	0	0	0
huge page fault	1	0	0	0
context switch	2	0	1	1
thr create	3	0	1	1
thr create-child	4	0	1	1
mid read	1	0	1	1
mid write	1	0	1	1
big read	1	0	1	1
big write	1	0	1	1
big munmap	1	1	0	1
huge read	1	0	1	1
huge write	1	0	1	1
huge munmap	1	1.001	0	1.001
fork	2	0	2	2
big fork	2	0	2	2
huge fork	2	0	2	2
huge fork-child	17	0	7	7
big fork-child	17	0.006	7.02	7.026
fork-child	17	0.012	7.065	7.077

Figure 3.7: The microbenchmarks, sorted by the sum of the number of transparent (**T**) and intentional (**I**) world switches per iteration, along with the number of system calls invoked (including page faults).

Configuration	Transparent	Intentional
None	2457 cycles	1082 cycles
SpectreV2	2453 cycles	1075 cycles
MDS	3337 cycles	1980 cycles
MDS+SpectreV2	3363 cycles	1992 cycles
MDS+SpectreV2+Q_retpoline	3406 cycles	2014 cycles

Figure 3.8: The costs of transparent and intentional world switches for different configurations.

When we modify the microbenchmark to pin the two processes to *different* cores we observe that it runs without world switches and that the overhead is about 25 times lower than Linux-style mitigations.

Application: git. To confirm that the improved performance of WARD's fast mitigations seen in LEBench translates into application-level performance improvements, we evaluated the performance of `git`. For this benchmark, we ran `git status` in a 100 MB repository that we cloned from GitHub; all of the file system state was cached in memory. The average runtime for Linux-style mitigations took 24.6% longer than the unmitigated baseline, and USC-style mitigations took 11.2% longer than the unmitigated baseline. Much of the speedup is due to the fact that `git status` invokes frequent `lstat` system calls, which can execute in the Q domain. The remaining overhead is due to system calls like `openat` that require a world barrier for accessing potentially sensitive file contents.

3.6.3 World switch

§3.6.2 shows that the mitigation overhead is dominated by the cost of a world switch. This section breaks down this cost.

An intentional world switch via `kswitch()` takes around 644 cycles on a shallow stack, plus 50 cycles or so for every KB of stack used (the cost of a `memcpy`). A transparent world switch using a page fault adds 1372 cycles.

Figure 3.8 measures the cost of a null system call that invokes an intentional or a transparent world switch, and returns. It shows the cost for different configurations: no mitigations, MDS mitigations, SpectreV2 mitigations, and with `retpoline` in Q domain. The configuration with Q_retpolines runs with retpolines in both the Q and K domains. It shows the benefit of WARD patching them out at runtime: the retpoline that disables branch prediction for indirect jumps through the system call table costs 22 cycles.

3.6.4 WARD memory overhead

Because the memory protection mechanisms that WARD uses to expose non-secret data to Q domains operates on a 4KB or 2MB granularity, WARD's approach incurs some additional memory overhead. Figure 3.9 lists some of these cases. In general we face a trade-off when filling small dynamic memory allocations for Q domain state: either we use an entire page each time, or we tolerate higher memory fragmentation because all chucks of memory on a page must be only used by the same Q domain.

3.6.5 Security

To validate that WARD's mitigations work, we implemented a demonstration program that attempts to execute a Spectre V2 attack against the WARD kernel. While running with applicable mitigations disabled (i.e. each Q and K domain retpoline replaced with a normal indirect jump) the attack succeeds in exfiltrating secret kernel data. However, when our

Component	Overhead	Explanation
Kernel text	2 MB	Separate text segments for Q and K domains
Public kernel data	< 4 KB	Padding to a page boundary
Process structure	4 KB / process	Allocated on its own page
Thread structure	~6 KB / thread	Split between a Q domain page and a K domain page
Q domain stack	32 KB / thread	Smaller stacks possible by avoiding deep recursion
Page tables	*varies*	Q domain mappings require additional PTEs
Inodes	–	Many public allocations packed into a single page
Scheduler state	–	

Figure 3.9: Memory overhead of different WARD components.

Spectre V2 mitigations are re-enabled (by re-enabling retpolines in the K domain) the attack is thwarted. It is of course impossible to be certain that all variations on the attack would be blocked, but this test provides some confidence both that the unmitigated baseline is vulnerable to transient execution attacks, and that WARD is able to prevent them.

3.7 Discussion

Future vulnerabilities. It is likely that there are further transient execution attacks either under embargo or yet to be discovered. Based on trends in the existing attacks, we believe that WARD should be well positioned to address them: so far, mitigations developed for Linux have been suitable to directly copy into WARD. Since many need to run only at K domain entry/exit instead of every user-kernel boundary crossing, the same defenses in WARD might be cheaper to apply than they would be for Linux.

Linux. We are optimistic that Ward's techniques could also benefit monolithic production kernels for two reasons. First, WARD and Linux are in the same ballpark in terms of system call performance on LEBench. Out of the 30 microbenchmarks, WARD is faster than Linux on 18 of them, and slower on 12. Second, as shown in Figure 3.1 (§3.1) Linux incurs a significant overhead for mitigations on LEBench and that overhead is in line with the overhead that WARD's Linux-style mitigations incur on LEBench (see Figure 3.6). Some systems calls experience more overhead in WARD, because they implement less functionality (e.g., `getpid`), but the corresponding calls in Linux also incur significant overhead. Some systems calls in WARD have less overhead than Linux, because they are not as efficient; for example, big and huge `mmap` in WARD requires an update of its radix-tree VM data structures [15], while Linux just inserts the new region into a list. Linux may see a bigger payoff for those system calls with WARD's design than WARD.

A question is how much effort is required to incorporate WARD's techniques into a production kernel such as Linux. Our preliminary efforts have proven encouraging: we found that we could leverage existing infrastructure for KPTI to maintain Q domain and K domain page tables. We implemented a `switch_world` function in Linux, which switches to the K domain and copies the Q stack to the K stack. We modified the Linux page-fault handler to call this function when it encounters a page fault while running with the Q page table. This allows the Linux kernel to run as normally with a transparent world switch on each system call. We refactored the `struct task_struct` into a Q-private and secret part, allowing the `gettid` system call to run entirely in the Q domain. This gives us some indication that the basic approach of WARD could be made to work in Linux, although an open question is how to best re-design Linux's data structures to fit WARD's design.

Chapter 4

Related Work

This thesis is motivated by the papers that show how secret kernel data can be leaked through micro-architectural state, which started with the discovery of Meltdown [41] and the original Spectre [32] variants. These were rapidly followed by the discovery of more attacks targeting transient execution, including MDS [12, 51, 57], Speculative Store Bypass [24], and many others [7, 9, 10, 14, 33, 46, 54, 58, 61]. Several survey papers categorize the known attacks [11, 22, 64].

4.1 Measuring performance

Simakov [53] and Prout [45] conducted early performance studies on the impact of transient execution attacks, but the most comprehensive results come from the many articles published by Phoronix [35, 36, 38]. 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 [56], KPTI [20], and MDS [35]. Their efforts has played a role in both understanding and driving down the performance overheads.

4.2 Mitigation approaches

Linux makes heavy use of software and microcode-based mitigations on older processors [39], and WARD adopts the same techniques and optimized implementations for the K domain. These include Linux's `nospec` macro for bounds clipping, `FILL_RETURN_BUFFER` to fill the return buffer, and retpoline. WARD's hotpatching of its kernel text to remove retpolines in the Q domain was inspired by Linux's ALTERNATIVE macro [17].

Many attacks have now been fixed in production hardware [27], but the known hardware techniques for addressing other attacks require more substantial changes [2, 5, 60, 65, 66]. Generally these techniques 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.

User-space sandboxing requires its own set of techniques. Swivel [43] is a compiler framework which hardens WASM bytecode against attack, while Firefox's and Chrome's WASM engines rely on Site Isolation [47]. Production JavaScript engines deploy more targeted mitigations like Pointer Poisoning and Index Masking [44], and also reduce the overall timer precision [44, 59]. Compiler techniques like Speculative Load Hardening [13] ensure binaries are completely immune to Spectre, albeit at considerable overhead.

4.3 WARD

The Q page table is inspired by the shadow page table in KAISER [21] and KPTI [40]. In Linux, when a process executes in user space, the process runs with a shadow page table, which maps only minimal parts of kernel memory: the kernel memory to enter/exit the kernel on a system call. As soon as the process enters the kernel, it switches to the kernel page table that maps all of physical memory. WARD, however, executes complete system calls while running under the Q page table; this requires a significant redesign of the OS kernel, which is a major focus of this work.

WARD and ConTExT both constrain speculation based on memory mappings, but ConTExT uses a new PTE bit to explicitly mark pages that contain secret data [52] while WARD instead keeps secrets in separate address spaces. SpecCFI proposes to enforce control-flow integrity during speculative execution [34]. This idea strengthens Spectre defenses, and is complementary to WARD.

The use of virtual-memory to partition the kernel address space has a long history in operating systems research. One example is Nooks [55], which runs device drivers in separate protection domains with their own page table in kernel space to provide fault isolation between drivers and the kernel. Another example is the use of Mondrian Memory Protection [62] to isolate Linux kernel modules in different protection domains within the kernel address space [63]. The most recent example is Mike Rapoport's work on kernel address space isolation [18] in Linux. These designs use similar techniques to introduce isolation domains within the kernel, but focus on traditional attacks (e.g., code execution through a buffer overflow) as opposed to transient execution.

Chapter 5

Discussion and Future Work

The ongoing impact to performance of transient execution attacks is continuing to decline on newer processors, and on some workloads isn't even measurable at all. Watching the stream of new transient execution attacks being published might not give this impression, but that comes from looking at the quantity of attacks independent of their performance drain.

One aspect in particular that is easily overlooked is that production operating systems don't try to provide perfect isolation in the first place. This is a challenge for the security community because it is much harder to model, but a tremendous opportunity to the systems community that can ignore low severity side channels (or accept incomplete fixes for more severe ones). It means that a proposed method by a security researcher to mitigate a specific transient execution attack might be deemed excessive by the developer community.

One example is disabling hyperthreading. While theoretically necessary to mitigate certain attacks, Linux by default choses to just ignore that advice and run mutually untrusting processes on adjacent hyperthreads. An even more pronounced case is Spectre V1: the only way to be 100% sure that no gadgets are present is to insert speculation barriers of some sort after every single branch in the kernel, because there isn't a programmatic way to know which are vulnerable. Instead Linux developers just annotate all the specific code sequences they can find to prevent them from being exploited. From a theoretical sense this is unsatisfying since there are almost surely places that have been missed. However, the constant stream of memory safety and logic bugs being discovered in the kernel makes any comparatively difficult to exploit Spectre vulnerabilities of limited concern.

If strong security against transient execution attacks is needed, critical applications can achieve much higher guarantees of isolation by running on dedicated hardware because transient execution attacks only work if the attacker and victim are sharing the same physical machine. This makes them immune even to attack variants that haven't been discovered yet, but naturally comes with its own downsides. The combination of added cost, reduced utilization, or degraded usability mean that this approach is usually reserved only for applications that absolutely require it.

It is also worth considering performance impacts in context. The Zen 3 processor we experimented with might look like it has the worst overhead from Speculative Store Bypass Disable, but that is only on relative terms. In fact, it is so much faster than any of the other machines we tested on the benchmarks, to the point that with SSBD enabled it outperformed all the other processors when they had SSBD disabled. That partly comes down to clock speed, but also the generational improvement in CPUs, which almost invariably exceeds the at this point roughly 3% overhead that OS-level mitigations cause to syscall workloads.

With the context discussed so far, the major area most applicable for future work is

JavaScript isolation. Across all the processors we studied, overheads from mitigations have been high and unchanging across processor generations. A big chunk of JavaScript overhead is from SSBD, which may be solvable by hardware at low cost or avoided by simply not enabling the mitigation. However, that requires drawing attention to it: most public documentation emphasizes that SSBD isn't used by default and neglects to mention that critical applications like web browsers frequently run with it enabled.

The second facet of possible future work is addressing Spectre V1 from sandboxed JavaScript code. JIT compilers can readily insert appropriate bounds check instructions and speculation barriers to prevent the common variants of the attack. However, existing processors are not tuned to accelerate these code patterns, so the cumulative impact of running them thousands or millions of times can have a detrimental impact on performance. This doesn't need to be the case. The computer architecture community has already studied ways for a CPU to perform memory operations without introducing Spectre gadgets [2, 65, 66]. Those techniques can be painfully slow when applied to every memory operation in a program, but could be far more reasonable when applied only to memory instructions flagged by the JIT.

Chapter 6

Conclusion

This thesis measures the performance penalties for mitigations against transient execution side-channel attacks across generations of Intel and AMD processors, and introduces the WARD kernel design, which eliminates as much as half this overhead for OS heavy workloads on old processors.

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.

The WARD design shows how USC can be used to reduce the performance costs of mitigations on system calls using per-process Q domains and global K domains. WARD transparently switches from Q to K domain through page faults, uses temporary mappings to access unmapped physical pages, and splits data structures into public and private parts. An evaluation shows that WARD can run the microbenchmarks of LEBench with small performance overhead compared to a kernel without mitigations: for 18 out of 30 LEBench microbenchmarks, WARD's performance is within 5% of the performance without mitigations. On Broadwell, WARD overall has a geometric mean slowdown of 16% on LEBench compared to 32% for Linux. Although WARD is research kernel, we are hopeful that the ideas of this thesis can drive further progress in production kernels and web browsers.

Bibliography

[1] Advanced Micro Devices, Inc. Speculation behavior in AMD micro-architectures. `https://www.amd.com/system/files/documents/security-whitepaper.pdf`, 2019.

[2] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 132–144. IEEE Press, 2020.

[3] Bytecode Alliance. Cranelift code generator. `https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/README.md`, January 2022.

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

[5] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Spec-Shield: Shielding speculative data from microarchitectural covert channels. In *Pro-*

ceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques, pages 151–164, Seattle, WA, September 2019.

[6] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. 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, November 2020.

[7] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Can Bölük. 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.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, Baltimore, MD, August 2018.

[10] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI:

Hijacking transient execution through microarchitectural load value injection. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72, 2020.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *CoRR*, abs/1811.05441, 2018.

[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 769–784, London, United Kingdom, November 2019.

[13] Chandler Carruth. Speculative load hardening. `https://llvm.org/docs/SpeculativeLoadHardening.html`, 2018.

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 142–157, 2019.

[15] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM EuroSys Conference*, pages 211–224, Prague, Czech Republic, April 2013.

[16] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for

multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, November 2013.

[17] Jonathan Corbet. SMP alternatives. `https://lwn.net/Articles/164121/`, 2005.

[18] Jonathan Corbet. Generalizing address-space isolation. `https://lwn.net/Articles/803823/`, November 2019.

[19] Google. Octane 2. `https://developers.google.com/octane/benchmark`, 2020.

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

[21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems*, pages 161–176, Bonn, Germany, July 2017.

[22] Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. On the Spectre and Meltdown processor security vulnerabilities. *IEEE Micro*, 39(2):9–19, 2019.

[23] Jann Horn. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, 2018.

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

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

[26] Inc. Intel. https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf, 2018.

[27] Inc. Intel. 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.

[28] 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.

[29] Intel, Inc. Software guidance: L1 terminal fault. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault, 2018.

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

[31] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, CA, May 2019.

[33] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, page 3, USA, 2018. USENIX Association.

[34] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SpecCFI: Mitigating Spectre attacks using CFI informed speculation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 39–53, San Francisco, CA, May 2020.

[35] Michael Larabel. 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.

[36] Michael Larabel. Looking at the linux performance two years after spectre /

meltdown mitigations. https://www.phoronix.com/scan.php?page=article&item=spectre-meltdown-2, 2020.

[37] Michael Larabel. 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, November 2021.

[38] Michael Larabel. 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.

[39] Linux Kernel Maintainers. Hardware vulnerabilities. https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/, 2020.

[40] Linux Kernel Maintainers. Page table isolation. https://www.kernel.org/doc/Documentation/x86/pti.txt, 2020.

[41] 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. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, August 2018.

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

[43] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*, 2021.

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

[45] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther, and Jeremy Kepner. Measuring the impact of spectre and meltdown. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5, 2018.

[46] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores area real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2021.

[47] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, Santa Clara, CA, August 2019. USENIX Association.

[48] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In

Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), pages 554–569, Huntsville, Ontario, Canada, October 2019.

[49] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 554–569, Huntsville, Ontario, Canada, October 2019.

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

[51] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 753–768, London, United Kingdom, November 2019.

[52] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *CoRR*, abs/1905.09100, 2019.

[53] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, and Thomas R. Furlani. Effect of meltdown and spectre patches on the performance of HPC applications. *CoRR*, abs/1801.04329, 2018.

[54] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking FPU register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018.

[55] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

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

[57] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 88–105, San Francisco, CA, May 2019.

[58] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on intel cpus via cache evictions. *CoRR*, abs/2006.13353, 2020.

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

[60] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*, pages 572–586, Columbus, OH, October 2019.

[61] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[62] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, San Jose, CA, October 2002.

[63] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, Brighton, United Kingdom, October 2005.

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

[65] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 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)*, pages 707–720, 2020.

[66] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, Columbus, OH, October 2019.