Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract

Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

Today's kernels pay a performance penalty for mitigations—such as KPTI, retpoline, return stack stuffing, speculation barriers—to protect against transient execution side-channel attacks such as Meltdown [21] and Spectre [16].

To address this performance penalty, this paper articulates the *unmapped speculation contract*, an observation that memory that isn't mapped in a page table cannot be leaked through transient execution. To demonstrate the value of this contract, the paper presents WARD, a new kernel design that maintains a separate kernel page table for every process. This page table contains mappings for kernel memory that is safe to expose to that process. Because a process doesn't map data of other processes, this design allows for many system calls to execute without any mitigation overhead. When a process needs access to sensitive data, WARD switches to a kernel page table that provides access to all of memory and executes with all mitigations.

An evaluation of the WARD design implemented in the sv6 research kernel [8] shows that LEBench [24] can execute many system calls without mitigations. For some hardware generations, this results in performance improvement ranging from a few percent (`huge page fault`) to several factors (`getpid`), compared to a standard design with mitigations.

1 Introduction

Over the last two years, transient execution has emerged as a powerful new side-channel attack technique. Vulnerabilities have proliferated [5, 12], with examples now including Meltdown [21], Spectre [16], L1 Terminal Fault [4], RIDL [29], Fallout [6], ZombieLoad [25], CrossTalk [23], and SGAxe [28]. In contrast with conventional timing-based side-channel attacks [17], where the victim must access its data in a specific pattern in order to leak it, transient execution attacks are more serious because they often allow an attacker to precisely control which memory locations are leaked, including memory that might not be accessed on the committed execution path. This is of particular concern to OS kernels, which have access to all of physical memory, and therefore could leak data from any process through transient execution bugs. In a public cloud, where it is common for mutually distrustful tenants to share a single machine [30, 35], the threat of transient execution is especially concerning.

A key challenge in addressing transient execution attacks lies in minimizing the performance overheads. CPU and OS designers have implemented a range of mitigations to defeat transient execution attacks, including state flushing, selectively preventing speculative execution, and removing observation channels [5]. These mitigations impose performance overheads (see §2): some of the mitigations must be applied at each privilege mode transition (e.g., system call entry and exit), and some must be applied to all running code (e.g., retpolines for all indirect jumps). In some cases, they are so expensive that OS vendors have decided to leave them disabled by default [2, 22]. Recent processor designs have also incorporated mitigations into hardware, which also reduces performance compared to earlier processor designs that do not perform such hardware mitigations.

To address the above challenge, this paper proposes a new hardware/software contract, called the *unmapped speculation contract*, or USC for short. 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. The benefit of USC is two-fold. From the OS designer perspective, it provides bounds on what data can be leaked through transient execution, and, as we show in the rest of this paper, can significantly reduce the cost of mitigations. From the hardware designer perspective, USC allows the CPU to keep many of the current speculative execution optimizations and their associated performance benefits. Most processor architectures already adhere to 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 [14, 15].

To demonstrate the benefits of the unmapped speculation contract, this paper 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. This paper presents 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 [8] 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 [24], which represents the most important system calls for a range of application workloads: Spark, Redis, PostgreSQL, Chromium, and building the Linux kernel. LEBench allows us to precisely measure the impact of mitigations on system calls that matter for applications. The most recent Intel CPUs (such as Cascade Lake) include hardware mitigations that cannot be fully disabled; however, some of these mitigations are not needed in WARD. To avoid the performance overhead of such unnecessary mitigations, we run experiments on the previous generation of Intel CPUs (Skylake).

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%, and 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.

2 Motivation

Transient execution mitigations harm kernel performance in two ways. First, they place overhead on code execution by disabling speculation. For example, the Linux Kernel uses a retpoline patch to mitigate Spectre V2, which replaces each indirect branch with a sequence of instructions that prevent the CPU from performing branch target speculation [13]. Second, these 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 [11, 20] switches to a separate page table before executing kernel code to prevent Meltdown attacks [21]. Workloads that are system call intensive (e.g., web servers, version control systems, etc.) are impacted by this type of overhead, while non-kernel intensive workloads see little performance impact [11].

Collectively, these and other mitigations can result in large slowdowns. To better understand this problem, we run LEBench [24], a microbenchmark suite of system calls that impact application performance the most. 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 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 Skylake, and the newer Cascade Lake. On the older Skylake CPUs, 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 those made by Ren et. al.; they find that KPTI and Spectre V2 mitigations are the root cause of slowdowns in the Linux Kernel over the last two years [24].

The newer Cascade Lake CPUs exhibit lower relative overheads, partly because the processors include hardware mitigations for some of the transient execution vulnerabilities. However, these lower overheads are also in part due to the newer Cascade Lake CPUs being *slower* in the baseline case when software-controllable mitigations are disabled. Figure 2 shows the performance of the microbenchmark on Cascade Lake (Intel Xeon Silver 4210R) relative to the earlier Skylake CPU (Intel Xeon E5-2640 v4). Our experiment uses CPUs with identical clocks (2.4 GHz), and nearly identical other hardware (Dell PowerEdge T430 vs. T440), which allows the comparison to be meaningful. The results demonstrate that, although the new CPU is faster at some microbenchmarks, it is slower for many others: e.g., context-switching is about 20% slower. Although it is impossible for us to separate slowdowns

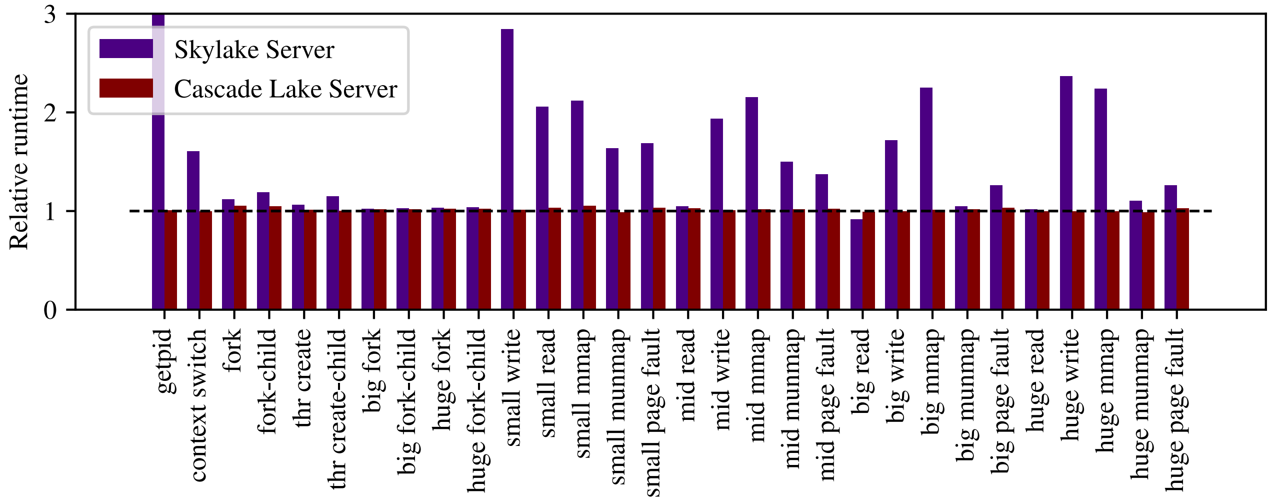

Figure 1: Linux slowdown due to mitigations on LEBench, for two generations of Intel CPUs: Skylake and Cascade Lake.

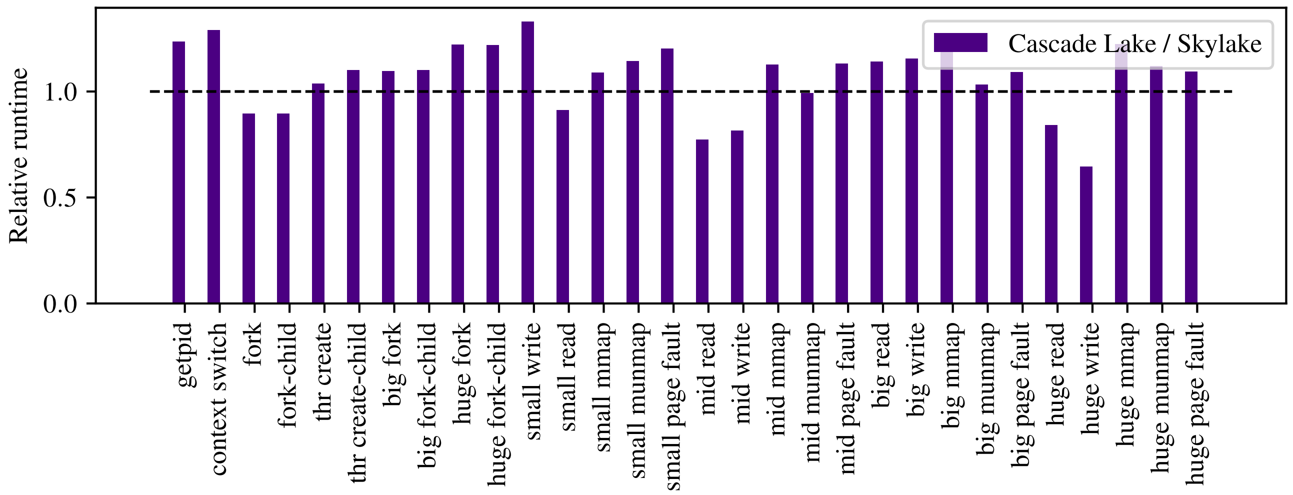

Figure 2: Performance regression on the newer Cascade Lake CPU, compared to the older Skylake CPU, for LEBench on Linux, with all software-controllable mitigations disabled.

due to mitigations from speedups due to architectural improvements, the results suggest that the overheads of mitigations implemented in hardware (e.g., for Meltdown, L1TF, or MDS) could still be significant. [1]

3 Goal and threat model

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 costly mitigations in hardware. Practically, however, it is difficult for us to disable hardware mitigations in the newest processors. Therefore, this paper focuses on reducing the overhead of software mitigations, and experimentally measures their effect on the previous generation of CPUs, where we can avoid mitigations altogether. We hope that WARD's design can allow processor designers to regain some of the absolute performance lost due to hardware mitigation costs.

Our threat model targets scenarios where the adversary and the victim are both running code on the same computer. This might arise either in a server setting, where both are running on a cloud computing platform, or in a client device, where the adversary code is a malicious application or web site.

Canella et al. [5] discuss transient execution attacks in detail, but the salient points of the attack boil down to four steps. First, the processor speculatively executes some code, which accesses sensitive data that the victim wants to keep secret. Second, during the speculative execution, the processor updates microarchitectural state in a way that depends on the sensitive data (e.g., bringing in cache lines into a shared L3

[1] One indication that this regression may be related to hardware mitigations is that measured branch mispredictions are around 40% higher on LEBench.

cache whose addresses depend on the sensitive data). Third, the processor aborts the speculative execution, but does not fully roll back all of its side effects (e.g., changes to the L3 cache), because doing so would be prohibitively expensive in hardware. Fourth, the adversary observes these side effects (e.g., using timing measurements), which allows the adversary to infer the sensitive data.

What makes transient execution attacks challenging to mitigate in an OS kernel is a combination of two factors. The first is that an adversary can trigger an OS 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, the adversary 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. The second factor is that 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.

Current OS kernel designs, such as Linux, have two approaches for mitigating transient execution attacks. The first approach is to make sure that the CPU does not speculatively execute any code that could end up accessing sensitive data. This approach includes techniques such as retpolines and other speculation barriers. The second approach is to make sure that sensitive data is flushed from microarchitectural state, such as flushing CPU caches and buffers when returning from a system call or when context-switching between processes. Both incur significant performance overheads.

Transient execution attacks can leak data across many protection domain boundaries, such as 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 of this paper 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 for this paper, 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), power side channels, etc.

4 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 paper 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, $\text{step}(\cdot)$, which includes any speculative execution done by the processor on that cycle, and require that unmapped pages cannot influence it:

$$\begin{aligned}&\forall s, s',\\ &\text{if } A(s) = A(s') \text{ and } M(s) = M(s'),\\ &\text{then with } S := \text{step}(s) \text{ and } S' := \text{step}(s'),\\ &\text{it must be that } A(S) = A(S')\end{aligned}$$

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 ($S := \text{step}(s)$ and $S' := \text{step}(s')$ 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 [23]), 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. On the one hand, hardware can keep the high performance provided by out-of-order execution, because the contract allows almost all forms of speculative execution, as long as data during speculative execution is accessed through non-speculative TLB entries. On the other hand, software can 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.

Although the contract is aspirational, one appealing property of the contract is that modern computer architectures already effectively aim to provide such a guarantee. 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 [6, 25, 29]. 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, as we demonstrate in WARD, by avoiding kernel mappings of sensitive memory.

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.

More generally, the contract helps categorize existing attacks based on which part of the system state they leak, as shown in Figure 3. 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

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

Figure 3: Transient execution attacks categorized based on the state leaked by the attack.

attacks in Figure 3 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. 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 through hardware and microcode revisions [14, 15], as confirmed by Canella et al. [5].[2]

5 Design

Under the assumption of the unmapped speculation contract, this section describes how WARD can reduce the cost of mitigations for system calls. §5.1 provides an overview of WARD's design with subsequent sections providing more detail about WARD's switch between protection domains (§5.2), about the mitigations used by WARD when mitigations are necessary (§5.3), WARD's kernel text (§5.4), WARD's memory management modifications (§5.5), WARD's process management split (§5.6), and WARD's file system split (§5.7).

5.1 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 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

[2]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.

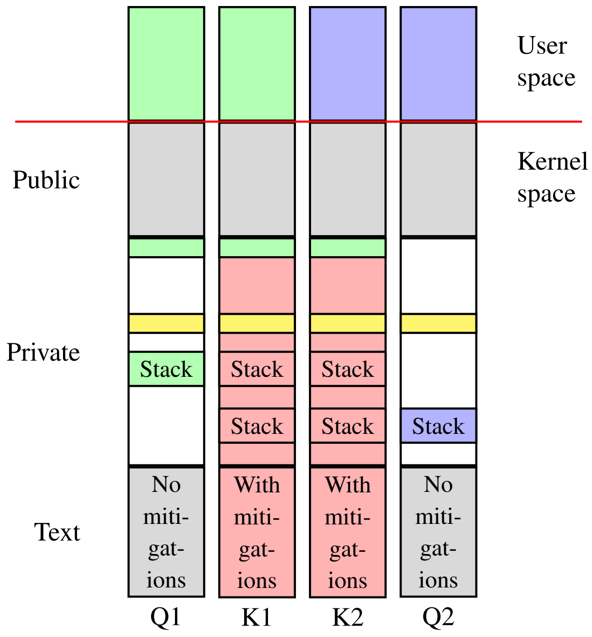

Figure 4: Overview of WARD's address space layout with two processes (indicated by the colors green and purple). Each process has a Q and K domain. Q domains have access to public data (the grey color) and per-process kernel data; the white private region is unmapped kernel data. Each domain also has 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.

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: it requires access to the process's own page table and to the memory allocator, but not other processes' page tables or pages.

Figure 4 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.

5.2 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 §5.4).

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 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.

Transient execution vulnerability	**U/Q**	**K**	**Ctx**
L1TF	x	x	
V1 (Bound Check Bypass)		x	
V1.1 (Bounds Check Bypass Store)		x	
V3 (Meltdown)		x	
V4 (Speculative Store Bypass)		x	
V2 (Branch Target Injection)		x	x
Microarchitectural Data Sampling (Fallout, RIDL, Zombie Load, etc.)		x	x
LazyFPU			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		
V1.2 (Read-only Protection Bypass)	Not applicable		

Figure 5: The mitigations implemented in software by WARD.

5.3 Mitigations

Figure 5 shows the known transient execution attacks [5, 12], organized by the mitigations needed to address those attacks in WARD's design. The columns (U/Q, K, and Ctx) indicate where the mitigations are needed: respectively, while executing in user-mode or 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), 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 microcode from Intel fixes 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. Spectre v2 and MDS attacks 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 LazyFPU 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).

5.4 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 [9], 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.

5.5 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.

5.6 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 save 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.

5.7 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.

5.8 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.

5.9 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.

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.

6 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 [8].

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.

Transient execution variant	Strategy	Support
V1 (Bound Check Bypass)	bounds clipping	partial
V1.1 (Bounds Check Bypass Store)	lfence	partial
V1.2 (Read-only Protection Bypass)	lfence	n/a (no in-kernel software sandbox)
V2 (Branch Target Injection)	retpoline	yes
—"—	speculation barrier	yes
—"—	return stack buffer filling	yes
—"—	disable spec before BIOS calls	n/a (no calls to BIOS in WARD)
V3 (Meltdown)	Kernel page table isolation (KPTI)	yes
V3a (System Register Read)	microcode	yes
V4 (Speculative Store Bypass)	disable spec. or ctx. switch	yes
LazyFPU	hardware-assisted save/restore	yes
SpectreRSB	return stack buffer filling	yes
L1TF	cache flush, no SMT	n/a (no VM entry in WARD)
—"—	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. bypass)	lfence	n/a (no bounds check instructions)

Figure 6: Transient execution mitigations implemented in WARD.

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 §5.2). 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 §5.7).

Mitigations. WARD implements side-channel mitigations for known transient execution attacks [5, 12], as shown in Figure 6. WARD mostly copies the mitigation strategies and their implementation from the Linux kernel [19]; the most interesting exception is that WARD does not apply some of these mitigations to the Q domain, as described in Figure 5.

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 Spectre V3, WARD uses separate page tables (as described in §5.1) 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 LazyFPU, 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.

7 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? (§7.2)
- How do mitigations affect the cost of a world switch? (§7.3)
- What are the memory overhead associated with WARD's design? (§7.4)

7.1 Experimental methodology

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

- Baseline: WARD with no mitigations against side channels.
- Linux-style: WARD with standard mitigations against side channels, mirroring the approach taken by the Linux kernel. This configuration does not use separate Q domains; all system calls directly enter the K domain.
- 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 [24], a collection of system call workloads representative of a range of real applications. 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 §2. 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.

7.2 WARD's USC-based fast mitigations

LEBench. Figure 7 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 8.

Many microbenchmarks (`getpid` through `huge pagefault` in Figure 8) 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 mitiga-

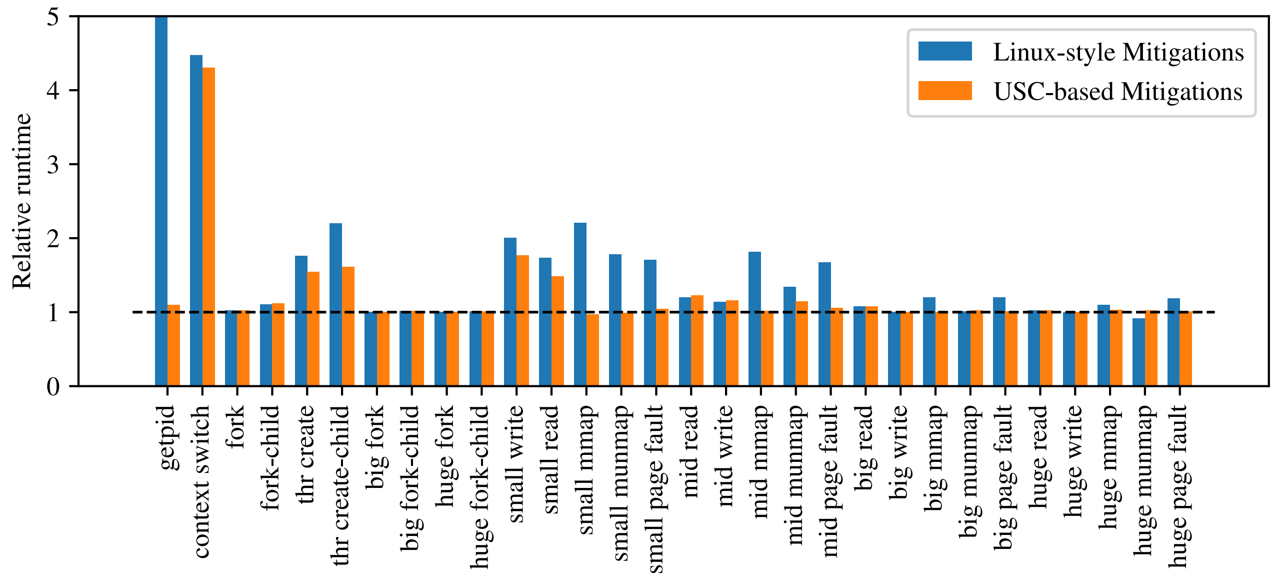

Figure 7: Performance of WARD with fast USC-based mitigations and with Linux-style mitigations, normalized against the baseline performance of WARD without any mitigations.

tions, and have the same overhead. These system calls also perform much work in the kernel and the overhead of the 1 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.

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

	# 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 8: 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 9: The costs of transparent and intentional world switches for different configurations.

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.

7.3 World switch

§7.2 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 9 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.

7.4 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 10 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.

7.5 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

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 10: Memory overhead of different WARD components.

succeeds in exfiltrating secret kernel data. However, when our 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.

8 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 1 (§2) 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 7). 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 [7], while Linux just inserts the new region into a list. Linux may see a bigger pay 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 completely 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 the data structures in the Linux kernel to fit WARD's design.

9 Related work

This paper is motivated by the papers that show how secret kernel data can be leaked through micro-architectural state (e.g., [4, 6, 16, 21, 25, 29]). In particular, two survey papers were helpful by categorizing the known attacks [5, 12].

This paper relies heavily on the mitigation work in the Linux community [19]. WARD adopts Linux's techniques and their optimized implementation in the K domain. WARD uses, for example, 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 [9].

In addition to the software/microcode approach currently used by Linux and other production operating systems, there are several proposed hardware-only defenses that delay the use of speculative data until it is safe [3, 31, 34]. While these defenses are more comprehensive, they have higher overheads that impact performance whenever speculation occurs. By contrast, the USC constrains speculation in a more targeted way based on memory mappings. ConTExT also proposes constraining speculation based on memory mappings, but introduces a new PTE bit to explicitly mark pages that contain secret data [26]. WARD instead keeps secrets in separate address spaces, and allows speculation after employing its defenses to switch to the K domain. Finally, SpecCFI proposes to enforce control-flow integrity during speculative execution [18]. This idea strengthens Spectre defenses, and is complementary to WARD.

The Q page table is inspired by the shadow page table in KAISER [11] and KPTI [20]. 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 paper.

The use of virtual-memory to partition the kernel address space has a long history in operating systems research. One example is Nooks [27], 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 [32] to isolate Linux kernel modules in different protection domains within the kernel address space [33]. The most recent example is Mike Rapoport's work on kernel address space isolation [10] 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.

10 Conclusion

This paper articulates the unmapped speculation contract (USC) for a division of labor between hardware and software. This contract allows hardware to speculate on many values (but not the values of page table entries) and provides software with a mechanism to prevent leaking secrets through micro-architectural state. 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 . Although WARD is research kernel, we are hopeful that its ideas can carry over to production monolithic kernels.

Acknowledgments

We'd like to thank the anonymous reviewers and our shepherd, Chris Hawblitzel, who provided comments that helped improve this paper. We also want to thank our artifact evaluators for their diligent examination of our artifact submission.

Artifact

Source code and directions for using WARD are available at `https://github.com/mit-pdos/ward`.

References

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

[2] Apple, Inc. Additional mitigations for speculative execution vulnerabilities in Intel CPUs. `https://support.apple.com/en-us/HT210107`, August 2019.

[3] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, pages 151–164, Seattle, WA, September 2019.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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`.

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

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

[16] 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.

[17] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*, pages 104–113, Santa Barbara, CA, August 1996.

[18] 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.

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

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

[21] 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.

[22] Microsoft Corporation. Windows guidance to protect against speculative execution side-channel vulnerabilities. `https://support.microsoft.com/en-us/help/4457951/`, November 2019.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. `https://sgaxe.com`, 2020.

[29] 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.

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM EuroSys Conference*, pages 18:1–18:17, Bordeaux, France, April 2015.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM EuroSys Conference*, pages 379–391, Prague, Czech Republic, April 2013.