A Principled Framework for Pliable and Secure Speculation in Operating Systems

Tae Hoon Kim

CMU-CS-24-161

December 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Dimitrios Skarlatos, Chair
Wenting Zheng

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Copyright © 2024 **Tae Hoon Kim**

Keywords: Operating Systems, Security, Speculative Execution, Virtualization

To my family

Abstract

Transient execution attacks present an unprecedented threat to computing systems. Protecting the operating system (OS) is exceptionally challenging because a *transient execution gadget* in the OS can potentially leak the *entire* memory.

In this work, we propose *Perspective*, a principled framework for building pliable and secure speculative execution defenses for the OS. Perspective offers a pliable interface that allows the OS to communicate its security requirements to hardware defenses, enabling tailored protection against transient execution attacks with little performance overhead. The design of Perspective is driven by a taxonomy of transient execution attacks in the OS kernel: (i) *active transient execution attacks*, where the attacker process exploits its own kernel thread to speculatively execute a transient execution gadget in the kernel, and (ii) *passive transient execution attacks*, where the attacker coerces the victim process's kernel thread to execute a transient execution gadget. Based on the taxonomy, Perspective introduces Data Speculation Views (DSVs) and Instruction Speculation Views (ISVs), to mitigate active and passive attacks, respectively. DSVs define the ownership of kernel data by a given execution context and block any speculative access to data outside the DSV. ISVs define the set of kernel functions that can be speculatively executed by a given execution context. Any transmitter instructions—whose execution could leak secrets, such as load instructions—that belong to kernel functions outside the ISVs are blocked from speculative execution. ISVs open up new opportunities of (i) swiftly patching gadgets in the OS, (ii) reducing the surface of passive attacks, and (iii) speeding up the process of auditing transient execution gadgets in the OS.

We build Perspective's software components in the Linux kernel and model the hardware components in gem5. We evaluate the security and performance of Perspective on a set of microbenchmarks and datacenter applications. Perspective has an execution overhead over an unprotected kernel of only 3.5% on microbenchmarks and only 1.2% on datacenter applications.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Dimitrios Skarlatos, for his outstanding mentorship and constant support throughout my graduate studies. Despite my limited background in systems and security when I first joined his lab as a junior three years ago, he saw potential in me and invested his time and expertise in guiding my development. His mentorship has been truly transformative and it has been an honor to work under his guidance.

Next, I am particularly grateful to Professor Neil Zhao (now at the University of Texas at Austin) for his invaluable mentorship and collaboration on Perspective. His patient guidance with gem5 and dedicated help in preparing for our ISCA presentation were instrumental to our project's success. The presentation was only possible because of the combined mentorship of Dimitrios and Neil, and I am deeply grateful for their support.

I would also like to thank my collaborators Kaiyang Zhao and David Rudo on Perspective. Kaiyang's expertise in operating systems and willingness to answer my questions have been invaluable to my understanding of complex systems. I am grateful to David for his significant contributions to the project, and presenting together at ISCA will remain a memorable highlight of my academic journey.

I am grateful to Professor Wenting Zheng for serving on my thesis committee and for her engaging lectures in the Secure Computer Systems course, which deepened my understanding of security principles.

Last but not least, I want to express my heartfelt thanks to my friends and family for their consistent encouragement and support throughout this journey. Their presence and support have made this accomplishment possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transient execution attacks [59, 60, 69, 69, 87, 107, 123, 123, 129, 135, 141, 145, 147, 149, 151] shattered the security isolation wall of modern processors. These attacks take advantage of *transient* instructions that may execute but not subsequently commit. An adversary can leverage *speculative execution gadgets* to leak sensitive data over microarchitectural covert channels [109, 157]. Such gadgets can leak any information within the address space in which they reside. Precariously, transient execution gadgets in the kernel can potentially leak *all* memory. This is because operating systems (OS), such as Linux, always map all physical frames [101] in the kernel address space for performance.

The unabating increase of the OS code base further exacerbates the threat of transient execution attacks. Manual code reviews by developers and security auditing become tremendously difficult, if not impossible. Case in point, the Linux kernel has reached 23 MLOC in 2023 [1]. Inevitably, this vast attack surface has led to a continuous stream of speculative execution CVEs [6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 21, 22, 23, 24, 26, 27, 28, 29, 32, 33, 34, 37, 38, 41].

A large body of work has focused on mitigating speculative execution vulnerabilities through software or hardware defense schemes. On the one hand, deployed software solutions are "spot mitigations" that provide limited coverage and often miss corner cases, requiring repeated efforts to resolve [6, 16, 18, 35, 43]. Furthermore, software approaches such as page table isolation [50, 56, 73, 80, 130, 155], Retpolines [142], LFENCEs [51], and even more heavyweight solutions such as core scheduling [99], cache flushing [100], and SLH [61] can have a significant impact on performance. For example, SLH shows a system call overhead of 65% on average [86]. Beyond performance, the deployment of software mitigations requires a lengthy procedure that involves kernel patches and even microcode updates. Given the requirement to manually add protection measures for identified gadgets [98], a principled way of mitigating transient execution attacks is needed.

On the other hand, several hardware solutions have been proposed [47, 62, 63, 72, 81, 85, 93, 97, 110, 112, 120, 126, 127, 139, 140, 150, 156, 158, 159, 163]. Predominantly, hardware defenses aim to be software transparent and backward compatible. However, without software information, hardware defenses are rigid, have to be always enabled, may miss corner cases, and may severely impact performance. Even some of the best-performing solutions [47, 159, 163] require intrusive changes in the core pipeline and especially in the cache hierarchy. Unfortu-

nately, such solutions are often impractical for adoption due to the significant verification costs inherited by their intricate interactions with the microarchitecture.

1.1 Pliable Secure OS Speculation

In this thesis, we propose *Perspective* [82], a principled framework for building efficient, lightweight speculative execution defenses for the OS kernel. Perspective offers a pliable interface between software and hardware that allows the OS to communicate its security requirements to the underlying hardware protection mechanisms. This interface opens up a new design space of tailored protection—the hardware protection mechanism selectively protects vulnerable instructions based on the security goals and the landscape of transient execution vulnerabilities, minimizing unnecessary protection. As a result, the hardware protection mechanism can be as simple as blocking speculative execution of vulnerable instructions while still attaining a low execution overhead.

The design of Perspective is driven by a taxonomy of transient execution attacks in the OS kernel. In this taxonomy, we identify two attack scenarios in the kernel: (i) *active transient execution attacks*, where the attacker process exploits its own kernel thread to speculatively execute a transient execution gadget in the kernel and (ii) *passive transient execution attacks*, where the attacker coerces the victim process's kernel thread to speculatively execute a kernel function containing transient execution gadgets. This taxonomy is agnostic to attack variants, such as Spectre v1 [87], Spectre v2 [87], Spectre RSB [88, 113], Retbleed [151], BHI [55], and others.

Based on our taxonomy, Perspective introduces two types of speculation views, Data Speculation Views (DSVs) and Instruction Speculation Views (ISVs), to mitigate active and passive attacks, respectively. A speculation view can be associated with an execution context, e.g., a process or a container.

The intuition of DSVs is that in an active attack, the attacker process exploits its kernel thread to speculatively access and leak the data *owned* by other victim processes or the kernel. Therefore, a DSV defines the set of data that a given execution context owns. Based on the DSV, the hardware mechanism can block any speculative access to data that are outside the DSV of the current execution context, as such access would violate data ownership. As a result, DSVs eliminate active attacks. In this thesis, we propose one possible approach for associating the data to the execution context that allocates them.

However, DSVs do not mitigate passive attacks. This is because in passive attacks, the victim process's kernel thread speculatively executes kernel functions containing transient execution gadgets to access and leak its own data, which does not violate data ownership. Therefore, to thwart passive attacks, Perspective proposes the ISV interface. ISVs enable an execution context to define the set of kernel functions that are trusted by the context. Any transmitter instructions that belong to kernel functions outside the ISVs are blocked from speculative execution. In this thesis, we propose several approaches to generate ISVs by marrying concepts from system call interposition [3, 5, 67, 77, 83, 96, 136].

The security benefits of ISVs are manyfold. First, ISVs provide an interface to swiftly mitigate unforeseen vulnerable kernel functions that contain transient execution gadgets. This benefit is vital given the continuous discovery of speculative execution vulnerabilities in kernel func-

tions. Second, a victim program can exclude a large portion of kernel functions that are not used or infrequently used from its ISV. This use case is akin to kernel debloating [91, 92], reducing the victim's surface of passive attacks. Finally, since ISVs provide the guarantee that kernel functions outside ISVs are blocked from speculative execution, one only needs to audit kernel functions within ISVs for transient execution gadgets. This approach speeds up the auditing process, as ISVs often include only a small fraction of the whole OS kernel functions. Moreover, any gadgets discovered during auditing can be excluded from ISVs, enhancing security.

We build Perspective's software components in the Linux kernel and model the hardware components in gem5. We evaluate the security and performance of Perspective based on a set of microbenchmarks and datacenter applications. Compared to unmodified Linux running on unsafe hardware, Perspective's DSVs *eliminate* active speculative execution attacks. Furthermore, ISVs reduce speculatively accessible functions by 95.1%. We then leverage the reduced search space provided by ISVs to speedup a state-of-the-art speculative execution scanner between 1.14-2.23× and 1.57× on average, and further block all identified gadgets. Finally, Perspective has an execution overhead over an unprotected kernel of 3.5% on microbenchmarks and 1.2% on datacenter applications.

We make the following contributions:

- A taxonomy of transient execution attacks and CVEs in the kernel that generalizes transient execution attacks into active and passive attacks. The taxonomy is agnostic to specific attack variants such as Spectre v1 and v2.
- Perspective introduces a principled framework for pliable and secure speculative execution in the kernel through two types of *speculation views*, data speculation views (DSVs) and instruction speculation views (ISVs), which protect against active and passive attacks, respectively.
- We present several design points on top of Perspective. These design points are based on tracking data ownership through allocation for DSVs and system call interposition for ISVs.
- A comprehensive security and performance evaluation of Perspective using microbenchmarks and datacenter applications.

Chapter 2

Background

In this chapter, we provide a brief overview of modern processor execution, transient execution attacks, and operating system security.

2.1 Speculative Execution

Speculative execution is a fundamental performance optimization of high-performance processors. Modern processors employ this technique to maintain high throughput and hide latency of operations such as memory accesses and branch resolutions. When the processor encounters an instruction whose outcome depends on a previous, yet-to-complete operation (such as a branch instruction waiting for a condition evaluation), rather than stalling, it makes a prediction about the outcome and continues executing instructions along the predicted path. If the prediction is correct, the processor commits these speculatively executed instructions, resulting in improved performance. If the prediction is wrong, the processor discards, or sqaushes, the results and rolls back to the previous architectural state. With speculative execution, some instructions may execute speculatively but do not commit due to pipeline squashes. We refer to the execution of instructions that do not commit as *transient execution*.

2.2 Transient Execution Attacks

Although transient execution cannot alter architectural states, it can leave sensitive information in microarchitectural states, such as caches, which can be recovered by attackers through microarchitectural covert channels [49, 109, 157]. This class of attacks is known as *transient execution attacks* [59, 87, 107]. In this thesis, we focus on Spectre-type attacks [59] that exploit transient execution caused by various prediction mechanisms in the pipeline, such as control-flow misprediction or memory-dependency misprediction. Such attacks exploit fundamental performance optimizations, making them difficult to defend against.

Spectre v1. The most notable variant of Spectre-type attacks is Spectre v1 [87], which exploits conditional branch mispredictions. Listing 2.1 shows a Spectre v1 code snippet. The snippet takes an index variable `idx` as input, which is controlled by the user. The index is then tested

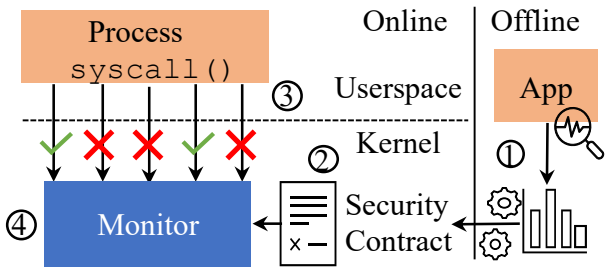

Figure 2.1: System call interposition overview.

against the size of `array1` to prevent out-of-bound accesses (Line 1). If the check passes, the code loads an element from `array1` (Line 2), which is in turn used as an index to access `array2` (Line 3).

```
1 if (idx < array1_size) { // mispredicted
2   u8 s = array1[idx]; // access
3   u8 y = array2[s*4096]; // transmit
4 }
```

Listing 2.1: Spectre v1 [87].

To carry out a Spectre v1 attack, the attacker first mistrains the branch responsible for performing the bounds check (Line 1). Then, the attacker re-invokes the code snippet with an out-of-bound index. The misstrained branch is taken and transiently executes the out-of-bound memory access on Line 2, potentially reading a secret value and leaking it through a cache-based covert channel [157]. This attack process can be generalized into two steps: (i) transiently *access* the secret (Line 2); and (ii) *transmit* the secret through a covert channel (Line 3). We refer to a code snippet that performs a combination of these two steps as a *transient execution gadget*.

Spectre v2 & Spectre RSB. Speculative control-flow hijacking attacks, coerce the victim to transiently execute unintended code paths by exploiting indirect branch and return address prediction, potentially leading to unauthorized data accesses. For example, the attacker can poison a branch target buffer (BTB) entry of a victim indirect branch and point the jump target to a transient execution gadget. This variant is known as *Spectre v2* [87]. Similarly, in *Spectre RSB* [88, 113] the attacker poisons a return stack buffer (RSB) entry to speculatively hijack the control flow of a victim return.

2.3 Operating System Security Isolation

In contemporary operating systems, such as Linux, virtual memory provides memory isolation between userspace processes and the kernel. To achieve high performance and avoid the high cost of context switches, the kernel is designed as a monolithic address space. A monolithic address space enables fast communication between different kernel components as memory can be directly accessed by any component without the heavy cost of context switching and interprocess communication. Furthermore, the kernel includes a direct map [101] area that maps all physical page frames in the system. Consequently, the kernel does not have to set up and tear

down mappings when it needs to copy data to and from userspace and can easily obtain physical addresses.

System Call Interposition. System call interposition is a powerful security technique that reduces the number of systems calls available to userspace processes. Such techniques [2, 3, 4, 67, 119] have been widely adopted by internet browsers [79], cloud deployments including Amazon's Firecracker [46, 53], Google's gVisor [70], and container technologies [66, 90, 114, 133, 143]. When execution transitions into the kernel, a security monitor inspects the system call and decides if its execution should proceed. Figure 2.1 shows an overview of system call interposition. First, in step ①, the application binary is analyzed to create a deny/allow system call list. This can be performed either statically through binary analysis or dynamically with tracing. During deployment in step ②, the list defines the security contract between the userspace process and the kernel. Every time a process performs a system call, shown in step ③, the security monitor intercepts it. Based on the security contract, the monitor decides to block or allow it in step ④.

Linux implements system call interposition through its Secure Computing mode (seccomp). Seccomp allows processes to restrict their system call interface by specifying which system calls can be made and what argument values are allowed. These restrictions, or *filters*, are expressed through Berkeley Packet Filter (BPF) programs, which operate on system call numbers and their arguments [102]. When a process makes a system call, the kernel evaluates it against these filters before execution, either allowing the call to proceed or taking action such as terminating the process or returning an error. By design, BPF programs cannot dereference pointers, which prevents time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks.

Chapter 3

Threat Model

Perspective's threat model considers environments where mutually distrusting parties share hardware resources, such as multi-tenant datacenters. We assume an unprivileged adversary who can trigger transient kernel execution and illegally access the victim's secret data. For example, the attacker can carefully prime microarchitectural structures, craft system call arguments, and exfiltrate the secret through transient execution such as Spectre v1 [87], Spectre v2 [87], Spectre RSB [88, 113], Retbleed [151], BHI [55], and so on. The goal of Perspective is to block transient execution attacks in the OS kernel that enable information leakage between different userspace processes or between the kernel and the userspace. Additionally, we assume a strong adversary that can exploit any side channel. Perspective enforces the Spectre threat model [156]. We do not consider attacks such as Meltdown [107] and MDS [144], as they are fixed in the newest processors. Non-transient side channels e.g., cache timing side-channels and attacks that leak non-speculatively accessed data are out of scope. Attacks such as Rowhammer [84], electromagnetic attack [131], and power side-channels [108] are out of scope.

Chapter 4

Motivation and Scope

This chapter presents the motivation and scope for developing defenses against transient execution attacks in operating systems. It describes a taxonomy of transient execution attacks in the OS (Section 4.1), and our systematic study of transient execution vulnerabilities in the Linux kernel (Section 4.2).

4.1 A Taxonomy of Transient Execution Attacks in the OS

Operating systems, including Linux, utilize a monolithic address space for high performance. For example, Linux uses a direct map [101] that maps all physical page frames in the system. However, from a security standpoint, a monolithic OS is more susceptible to transient execution attacks. This is because an attacker can circumvent protections, such as bounds checks, to speculatively access and leak all system memory from the OS address space. To better comprehend the risks, we present a taxonomy of transient execution attacks in the OS in two main scenarios. Note that this taxonomy is agnostic to variants of transient execution attacks, such as Spectre v1 [87], Spectre v2 [87], Spectre RSB [88, 113], Retbleed [151], BHI [55], and others.

Active Transient Execution Attacks. In an active transient execution attack, unauthorized memory accesses and subsequent disclosures are performed from the kernel thread of the attacker's process. Consider the example of Figure 4.1. The upper half shows two userspace processes, $Proc_a$ and $Proc_v$, representing the attacker and the victim. The goal of $Proc_a$ is to steal private data owned by $Proc_v$, denoted by $Secret_v$. The lower half represents the monolithic kernel, which is divided into two parts: (i) the kernel code, containing a transient execution gadget located on the path of a system call, with $r1$ being an argument of the system call; and (ii) the kernel data, encompassing the entire system memory, including $Secret_v$ owned by $Proc_v$.

In the example of Figure 4.1, similar to the generic Spectre v1 attack discussed in Section 2.2, $Proc_a$ first mistrains the branch responsible for performing the bounds check (Line 1 in Figure 4.1). This can be achieved by repeatedly making the system call with an $r1$ value that passes the bounds check, thereby biasing the branch toward being taken. Subsequently, $Proc_a$ initiates a system call with an out-of-bound $r1$ value (①), enters the kernel space, and speculatively executes the transient execution gadget. Due to previous mistraining and the out-of-bound index $r1$, the gadget speculatively accesses $Secret_v$ (②), which is then transmitted to the attacker through

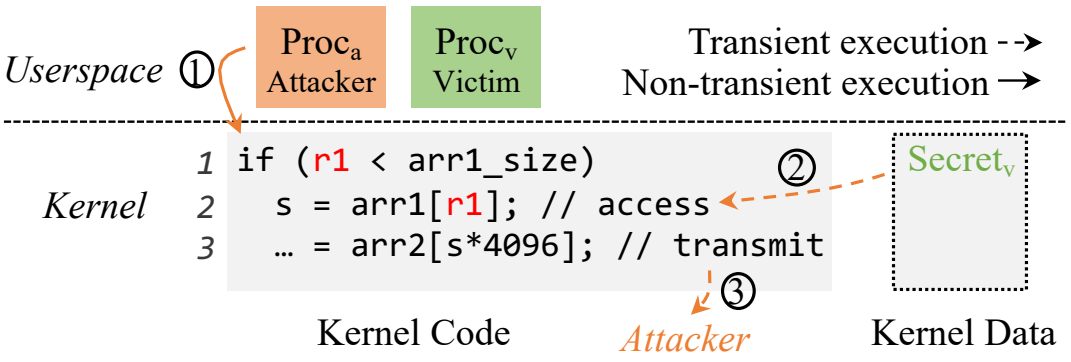

Figure 4.1: Example *active* transient execution attack in the OS. Input $r1$ is controlled by the attacker.

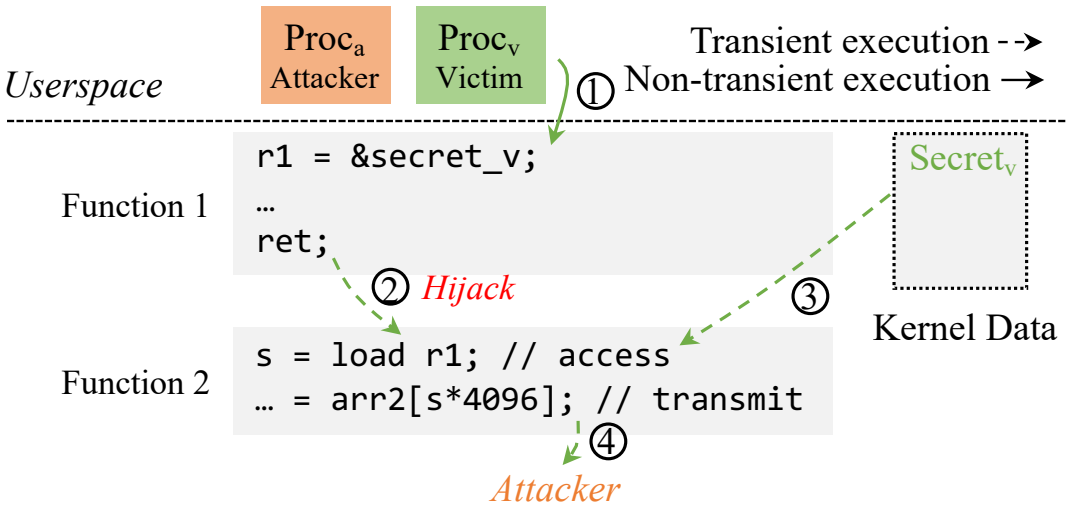

Figure 4.2: Example *passive* transient execution attack in the OS.

a cache-based covert channel (③).

Since active attacks exploit the monolithic nature of kernel data, one of Perspective's defense mechanisms, discussed in Chapter 5, is to identify kernel data ownership by different contexts—e.g., different processes or containers. By blocking speculative memory accesses that violate ownership, Perspective effectively eradicates active transient execution attacks.

Passive Transient Execution Attacks. In a passive transient execution attack, the victim speculatively executes both *access* to the secret and *transmission* of the secret. This scenario is analogous to "cross-domain transient execution attacks" in Intel's refined speculative execution terminology [76]. Since the speculative access to the secret is initiated by the victim, the access does not violate data ownership. Consequently, mechanisms that block speculative accesses violating data ownership are ineffective against passive attacks.

Figure 4.2 illustrates an example of passive attacks. The victim-issued system call executes Function 1 (①), which loads a reference of $Secret_v$ to $r1$. Since function 1 does not access $Secret_v$, its execution does not leak the secret. However, when the victim returns from Function

Table 4.1: A collection of speculative execution related vulnerabilities targeting the Linux kernel[2]

	Attack Primitives	Insufficient Mitigation	CVEs and Papers	Description	Origin of Vulnerabilities
1	Unauthorized speculative data access (Spectre v1)	n/a	[38]	Array index is not validated	Xilinx USB Driver
2		Misuse	[11]	Reintroduced Spectre vulnerabilities in backporting	ptrace
3		n/a	[20, 26, 27, 28, 29]	Out-of-bounds speculation on pointer arithmetic	eBPF verifier
4		n/a	[21, 86]	Speculative type confusion [86]	eBPF verifier
5	Speculative control-flow hijacking (Spectre v2, Spectre RSB, and more)	Hardware	[30, 31, 36, 55]	Branch history injection [55]	Indirect calls and jumps
6		Software	[18]	LFENCE/JMP is insufficient on AMD	Indirect calls and jumps
7		Software	[39, 40, 152]	Retbleed [152]	Retpoline [142]
8		Misuse	[42]	Missing retpolines or IBPB	KVM
9		Misuse	[12, 16, 35, 44]	Improper use of hardware mitigations	Indirect calls and jumps

1, the attacker can employ techniques such as Spectre v2 [87] and Spectre RSB [88, 113] to *hijack the victim's speculative control flow* to *any* kernel function, such as Function 2, which contains a transient execution gadget (②). This step serves two purposes: first, it coerces the victim to speculatively execute a transient execution gadget; second, it creates a speculative type confusion [86], causing the value in $r1$ prior to the hijacking, which is a memory reference to $Secret_v$, to be used out-of-context by the memory access in Function 2. Consequently, the hijacked execution speculatively accesses $Secret_v$ (③) and transmits the secret through a covert channel (④).

Compared to active attacks, passive attacks are challenging to orchestrate, since the attacker has no control over the victim's system calls and their arguments. As a result, passive attacks are generally assisted by hijacking the victim's speculative control flow to a kernel function that contains transient execution gadgets [76], as exemplified by Step ② in Figure 4.2.

Since passive attacks largely exploit the monolithic nature of kernel code, the second defense

[2]Table 4.1 in this thesis was developed in collaboration with David Rudo, as part of our join research work published in Perspective [82]

mechanism of Perspective, which will be discussed in Section 5.3, is to limit the victim's speculative control flow to a small subset of kernel functions and drastically reduce the attack surface for passive attacks.

4.2 Studying of Transient Execution Attacks in the Kernel

To gain a deeper understanding of the threats posed by transient execution attacks, we studied transient execution vulnerabilities in the Linux kernel. Our study focuses on two types of attack primitives that aim to (1) enable unauthorized speculative data access like Spectre v1; and (2) enable hijacking of the victim's speculative control flow such as Spectre v2 [87], Spectre RSB [88, 113], Retbleed [151], BHI [55]. These attack primitives are the basic building blocks of *both* active and passive transient execution attacks in the kernel. Moreover, these primitives have not yet been fully mitigated or are attacks whose mitigations introduce overhead.

Table 4.1 summarizes our findings. The first column of Table 4.1 illustrates the impact of the vulnerability, falling into two main categories: (i) enabling unauthorized speculative data access (Spectre v1), or (ii) enabling speculative control-flow hijacking (Spectre v2 and Spectre RSB). The second column indicates whether the vulnerability results from insufficient mitigations. If so, it further specifies whether it is caused by insufficient hardware or software mitigations or misused mitigations. The third column lists the relevant CVEs and academic papers. Finally, the fourth and fifth columns provide the vulnerabilities' descriptions and origins in the Linux kernel.

Unauthorized speculative data access (Spectre v1). Rows 1 to 4 list vulnerabilities that enable unauthorized speculative data access through Spectre v1-like attacks. From top to bottom, Row 1 represents CVE-2022-27223 [38], which identifies a transient execution gadget in the Xilinx USB peripheral driver with an unprotected index that allows transient outbound accesses. Due to the vast size of the Linux kernel, such vulnerabilities are often deeply buried within infrequently used modules, making it challenging to detect.

Then, Row 2 represents a vulnerability [11] that reintroduces Spectre vulnerabilities to `ptrace` when backporting Spectre mitigations to earlier Linux versions. This vulnerability highlights the possibility of accidentally creating transient execution gadgets in the Linux kernel.

Lastly, Rows 3 and 4 list a series of vulnerabilities in the eBPF program verifier. By exploiting these vulnerabilities, an attacker can inject transient execution gadgets into the Linux kernel by loading malicious eBPF programs, thereby achieving unauthorized speculative data access. This category of vulnerabilities is mitigated by fixing the verification logic of the eBPF program and disallowing unprivileged users to load eBPF programs by default [138].

Speculative control-flow hijacking (Spectre v2, Spectre RSB, and more). Rows 5 to 9 list vulnerabilities that enable speculative control-flow hijacking through Spectre v2- or RSB-like attacks. These vulnerabilities are caused by inadequate mitigations. Row 5 shows the Branch History Injection (BHI) attack [55] that bypasses hardware mitigations [75], which replace software mitigations like Retpoline [142]. Rows 6 and 7 list vulnerabilities that originate from insufficient software mitigations. Row 6 shows a vulnerability caused by the false belief that inserting an `lfence` before indirect jumps can defend against Spectre v2 on AMD processors. However, there is a race condition where an `lfence` is ineffective. Row 7 represents the Retbleed attack [152] that targets Retpoline, the most widely used software defense against Spectre v2.

Lastly, Rows 8 and 9 list a series of misuses of mitigations. These vulnerabilities arise because of the numerous hardware and software defenses against Spectre v2 on different CPUs that lead to confusion about which defense should be enabled.

Takeaways. Despite the tremendous efforts of kernel developers and hardware vendors, numerous transient execution vulnerabilities continue to arise. In our study, we find that transient execution gadgets are often deeply buried within infrequently used code, hindering their detection. Furthermore, active kernel development can inadvertently reintroduce new vulnerabilities. Adding to the issue, the OS defenses against speculative control-flow hijacking often fall short due to insufficient or misuse of mitigations. As a result, an attacker can directly exploit, or coerce the victim to exploit, an unpatched transient execution gadget deep within the kernel. This enables the attacker to perform active or passive transient execution attacks to steal secrets. These observations emphasize the necessity of blocking speculative memory accesses that violate data ownership and limiting the set of functions that can be speculatively executed by a context to a small subset of all kernel functions.

Chapter 5

Perspective Design

Perspective introduces a principled framework for building pliable and secure speculative execution defenses for the OS Perspective offers a pliable interface that allows the OS to communicate its security requirements to underlying hardware protection mechanisms. At a high level, Perspective consists of two types of Speculation Views, Data Speculation Views (DSVs) and Instruction Speculation Views (ISVs), which aim to thwart active and passive transient execution attacks in the kernel as presented by our taxonomy of attacks in Section 4.1. Perspective's design is rooted in this taxonomy, instead of attack variants, and hence it can mitigate all variants.

5.1 The Perspective Framework

Data Speculation Views (DSVs). In active transient execution attacks, unauthorized memory accesses and subsequent disclosures are performed from the kernel thread of the attacker's process to leak data *owned* by other victim processes or the kernel. The Perspective framework mitigates this class of attacks by isolating the kernel state based on ownership. Specifically, Perspective associates each process with a DSV. A DSV defines the set of data that a given execution context owns. Any speculative access to data outside the DSV is protected. For simplicity, we assume a simple but aggressive protection mechanism that *blocks* the speculative memory access until it becomes non-speculative. As the majority of accesses during kernel execution do not violate ownership, DSVs impose very low execution overhead even if the protection mechanism is as aggressive as blocking speculative accesses. In Section 5.2, we discuss one potential DSV design by defining ownership based on memory allocations.

Figure 5.1 shows an example of DSVs. The userspace processes live on the upper side of the figure, while the monolithic kernel space contains the kernel code and data. The kernel code is represented as a function call graph, where each circle represents a kernel function. DSVs are depicted in Figure 5.1 as a gray pattern with a colored line that represents the owner process. For example, Figure 5.1 shows that during the execution of $Proc_a$, the speculative access of the function Ⓒ is blocked as it tries to access data that belong to $Proc_v$'s DSV. Similarly, a process is only allowed to transiently access data within its own DSV, e.g., the transient access from node Ⓕ.

Instruction Speculation Views (ISV). In passive transient execution attacks, both the *access*

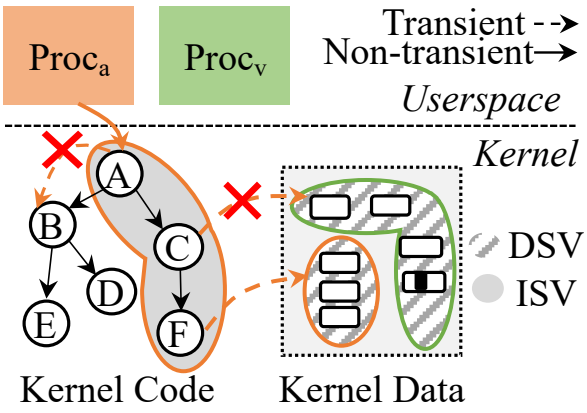

Figure 5.1: An example of data speculation views (DSVs) and instruction speculation views (ISVs).

to and the *transmission* of the secret are speculatively carried out by the victim's kernel thread. To thwart passive attacks, Perspective proposes the ISV interface that defines the set of kernel code that can be speculatively executed by a given execution context. To simplify the discussion, we assume that ISVs are defined at a kernel-function granularity, however, in practice ISV protection is applied at the instruction granularity. Any transmitter instructions—whose execution could leak secrets, such as load instructions—that belong to kernel functions outside the ISVs are protected (e.g., blocked from speculative execution). The ISV becomes active when a process transitions from userspace into the kernel, e.g., due to a system call. Figure 5.1 illustrates the ISV of $Proc_a$ using the call graph, where the kernel functions that belong to the ISV are enclosed in the orange line. For example, consider the control-flow edge from node Ⓐ to node Ⓑ. Because node Ⓑ is outside the defined ISV, the program cannot speculatively execute transmitter instructions from node Ⓑ. Section 5.3 introduces multiple possible ways to generate an ISV for a given victim program.

The security benefits of ISVs are manyfold. First, ISVs provide an interface to swiftly mitigate unforeseen vulnerable kernel functions that contain transient execution gadgets. This benefit is vital given the continuous discovery of transient execution vulnerabilities in kernel functions. Second, a victim program can exclude a large portion of kernel functions that are not used or infrequently used from its ISV. This use case is akin to kernel de-bloating [91, 92], reducing the victim's surface of passive attacks. Finally, since ISVs provide the guarantee that kernel functions outside ISVs are blocked from speculative execution, one only needs to audit kernel functions within ISVs for transient execution gadgets. To this end, we augment with Perspective's ISVs a state-of-the-art speculative execution gadget scanner [78] to dramatically reduce the scanner's gadget search time and produce strict ISVs that block all identified gadgets. Note that an attacker can hijack the victim's speculative control flow to the middle of a function that is within the ISV, Perspective builds on top of control flow integrity (CFI) techniques [89] to defend against these types of attacks.

Outlook. The interfaces of DSVs and ISVs open up a new design space of protection schemes against transient execution attacks. By changing the approaches of defining data ownership or generating ISVs, one can create various schemes tailored to their security requirements and

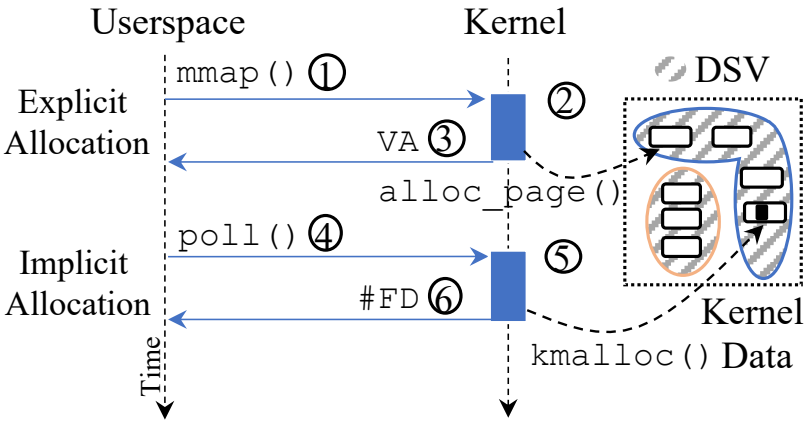

Figure 5.2: Tracking explicit and implicit allocations.

performance goals. In the following sections, we present several design points for DSVs and ISVs.

5.2 Defining Data Ownership for DSVs Through Allocations

The Perspective framework has many potential design points of DSVs by varying the definition of data ownership. We propose defining data ownership for DSVs based on the context of execution that the kernel operates on behalf of during memory allocation. Our insight is that allocations provide a strict security guarantee as they separate resources based on contexts. This approach simplifies Perspective's integration with real-world operating systems as allocations are commonly tracked for resource control.

Implicit and Explicit Memory Allocations. To guide our design, we identify two main types of kernel allocations, *explicit* and *implicit* allocations. We define as explicit all the allocations where the kernel allocates resources explicitly requested by userspace, e.g., memory, sockets, or files. The upper part of Figure 5.2 shows an example of an explicit allocation. Arrows represent the transitions between userspace and the kernel. Time traverses vertically from top to bottom. In step ①, the userspace process executes the `mmap()` system call.[1] Then in step ②, the kernel allocates available pages on behalf of the process. During this time, Perspective assigns the allocated pages to the DSV of the process. Finally, in step ③ the kernel returns the virtual address (VA) to userspace. Similarly, in the case of a page fault, the allocated page will be associated to the DSV of the faulting process. In the case of `munmap()`, Perspective removes pages from the DSV.

Implicit allocations are usually allocated by the kernel on behalf of the process for bookkeeping and metadata management. An implicit allocation example is shown in the lower part of Figure 5.2. The userspace process executes in step ④ the `poll()` system call. Within the system call, the kernel may allocate memory, e.g., through `kmalloc()`, to store metadata about the file descriptors being polled. This is shown in step ⑤. Such allocations belong to the process, and hence Perspective protects them by adding them to the process DSV. Finally, in step ⑥, the

[1]We assume the $MAP\_POPULATE$ flag is set for simplicity.

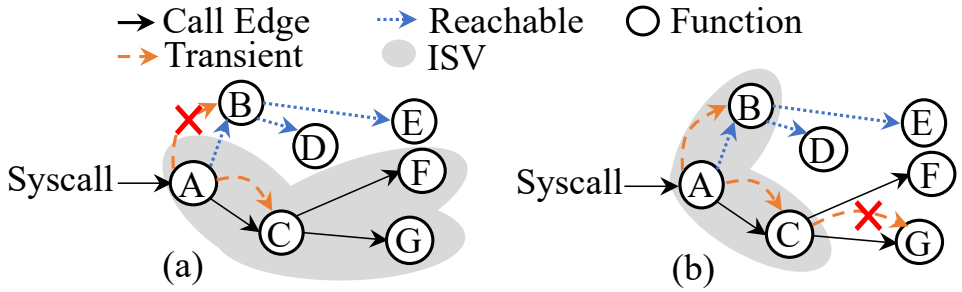

Figure 5.3: Perspective's (a) static, and (b) dynamic construction of ISVs.

return value of the number of file descriptors with events is returned to the userspace.

In practice, most implicit allocations are handled through slab allocations, e.g., through interfaces such as `kmalloc()` in Linux. The goal of the slab allocator is to maximize memory utilization and therefore tries to pack the allocations together [58, 105]. As a result, data belonging to mutually distrusting processes may get allocated even within the same cache line. For example, Linux slab allocations can be as small as 8 bytes [103, 104], much smaller than the contemporary cache line sizes of 64 bytes. However, this approach introduces a major challenge to security isolation. This is because Perspective would need to track data ownership at a granularity as small as 8 bytes, leading to high bookkeeping overhead. Perspective resolves implicit allocations by designing a secure slab allocator that isolates different execution contexts. We further discuss our implementation in Section 6.1.

5.3 Generating ISVs with System Call Interposition

We leverage the principled security guarantees of ISVs provided by the Perspective framework to propose a methodology on how to realize ISVs. Our proposal marries concepts from system call interposition (Section 2.3) used for application sandboxing with speculative execution. Specifically, Perspective introduces two techniques based on static and dynamic system call interposition to define personalized ISVs for each context.

Similarly to how system call interposition techniques define an allow list of system calls for an application, we identify potential entry functions to the kernel. Then, we use static or dynamic analysis to find the set of kernel functions used by these entry functions. These functions are trusted and form the basis of the ISV of the application.

Our approach avoids one of the common pitfalls of system call interposition. In particular, in conventional system call interposition enforcement, blocking a system call can have detrimental effects on applications as they may lead to irrecoverable errors and crashes. As a result, the system call allow list ends up being overly permissive to avoid usability problems. Instead, Perspective's insight is that even if kernel functions are excluded from the ISV, execution can continue non-speculatively. As a result, Perspective's ISVs can dramatically reduce the barrier for adoption.

Static ISVs. One approach to define ISVs is by statically analyzing the application binary. The first step is to identify the system calls that may be used by a given program. Next, Perspective statically analyzes the kernel binary to identify the set of functions that could be invoked during

each system call. Then, it combines the set of system calls with the set of functions that could be invoked by each system call to form a static ISV at the granularity of functions for the application.

Figure 5.3(a) shows an example of generating ISV with static analysis. The kernel code is represented as a call graph composed of function nodes, where the node Ⓐ is the system call entry point. A call edge, shown as a solid line arrow, represents calling relations between functions. From the entry point Ⓐ, the static analysis recursively follows the call edge to traverse and record all kernel functions that might be called during the execution of the system call. These functions form the static ISV, which is marked as a gray cloud that engulfs a subset of function nodes.

As with any technique that relies on an accurate call graph, indirect jumps pose a major challenge because they are hard to reason with statically, making it hard to identify functions are *reachable* via only indirect jumps. Figure 5.3(a) shows reachable nodes connected with a dotted blue arrow. Such reachable nodes are not included in the static ISV. Therefore, any speculative indirect jumps to these reachable nodes are blocked from speculative execution even if these reachable nodes are safe to speculatively execute, incurring unnecessary overhead. Shown in Figure 5.3(a) with an orange dotted line, a transient transition from node Ⓐ to Ⓒ is allowed in the static ISV but not from Ⓐ to Ⓑ, as the latter is a reachable node.

Dynamic ISVs. While static ISVs present a viable solution, to further reduce the attack surface of the kernel, Perspective introduces a dynamic technique to define ISVs. Specifically, Perspective leverages kernel-level process tracing to identify the set of actively used system calls and kernel function paths. Based on this information, Perspective generates a personalized dynamic ISV consisting of the kernel functions that are traced. Figure 5.3(b) shows the same high-level call graph but depicts a dynamic ISV that now includes a reduced set of nodes compared to the static ISV.

Compared to static ISVs, there are two benefits of using dynamic ISVs. First, dynamic ISVs can further reduce the attack surface over static ISVs by including only functions that are actually used by the application, not the functions that could be used. Second, dynamic ISVs generally offer better performance. This is because dynamic ISVs capture functions that are reachable via only indirect jumps, which otherwise would not be included in a static ISV. As shown in the example, in contrast to the static ISV, node Ⓐ can transiently reach node Ⓑ, potentially improving performance. In addition, node Ⓖ is now excluded from the ISV, improving security.

5.4 Discussion of ISVs

Dynamically Reconfigurable ISVs. An ISV provides an interface for *dynamically configurable security* at runtime. While an ISV is built offline and later provided to the OS at application startup, ISVs can become stricter later on. Specifically, during the runtime of the application, one can shrink the ISVs as certain system calls or function paths are no longer needed, reducing the attack surface. Furthermore, in a scenario where new vulnerabilities are found in kernel functions, the ISV can be adjusted to exclude affected functions, effectively mitigating new vulnerabilities without kernel patches and potentially expensive server downtime. Finally, it enables system administrators to install ISVs that could be applied to all or selected applications.

Accelerating Security Auditing with ISVs. Perspective's ISVs unlock the potential to efficiently identify gadgets using security auditing techniques such as static analysis, taint tracking, and fuzzing [74, 78, 121]. This is because functions excluded from ISVs cannot speculative execute, the auditing only needs to examine functions within ISVs, drastically reduce the search space We demonstrate this point in Chapter 8 with Kasper [78], a state-of-the-art speculative execution gadget scanner that uses taint analysis and fuzzing. This approach significantly improves the overall gadget discovery rate, as we demonstrate in Chapter 8.

Enhancing ISVs with Auditing. Perspective can leverage the accelerated security auditing to further enhance the security of ISVs. Specifically, after Perspective produces ISVs through system call interposition (Section 5.3), it can use them to bound the search space of kernel auditing. Perspective then uses the results of the ISV auditing to exclude all the vulnerable functions identified by the audit from the ISVs. We describe Perspective's implementation in Chapter 6 and present the results in Chapter 8.

Chapter 6

Perspective Implementation

This chapter presents the implementation of the Perspective architecture. It describes the operating system support for Perspective's speculation views (Section 6.1), and the architectural extensions required to efficiently control speculation (Section 6.2).

6.1 Operating System Support

The design principles of Perspective are general and applicable to most operating systems. For our implementation prototype, we build Perspective on top of the Linux kernel and control groups (`cgroups` [106]) for resource tracking. However, alternative implementations based on processes are also simple to implement, allowing Perspective to be applicable to most commercial operating systems, including Mac OS X, FreeBSD, and Windows. Our implementation includes about 400 LOC of changes for DSV management and the secure slab allocator. We further modify the kernel (about 400 LOC) to expose speculation view metadata to the simulator. To generate ISVs on top of radare2 [122] we require about 150 LOC. Finally, our modifications to Kasper [78] are about 150 LOC. Next, we discuss the implementation of instruction and data speculation views of Perspective.

Data speculation views with cgroups. The primary requirement to support DSVs in the kernel is the ability to track the ownership of resources. In Perspective we use `cgroups` [106] to associate allocations with a DSV per container. For kernel threads, each of them belongs to distinct DSVs to improve isolation. The kernel buddy allocator (through `alloc_pages()`) obtains the cgroup ID of the current process context during allocations and associates the allocated physical frames to a DSV for the corresponding page in the direct map. As a result, Perspective associates each page in the direct map with appropriate DSVs fully protecting the direct map. This approach is sufficient for mapping a frame into the userspace (e.g., in the page fault handler) or using a frame solely in the kernel. When a physical frame is freed, Perspective disassociates it from its DSV.

A Secure Slab Allocator. As discussed in Chapter 5, slab allocations are particularly challenging in current OS designs as they pack memory across mutually distrusting processes [103, 104]. To address this challenge Perspective introduces a secure slab allocator that isolates different contexts of execution at the granularity of pages. Specifically, for each slab serving a particular

object size or type, Perspective maintains separate lists of physical pages to store objects for each `cgroup`, eliminating collocation.

Resolving Unknown Allocations. Perspective tracks resource ownership through the page and the slab allocator, but on rare occasions memory used by the kernel does not use these two primary interfaces. We classify such allocations as *unknown*. By default, unknown memory does not belong to a DSV, hence Perspective conservatively blocks speculation. We identify three main sources of unknown allocations. First, unknown allocations may originate from global variables defined in the kernel code. For example, some are structs of function pointers describing operations that can be performed on a file. When possible Perspective redesigns the allocation mechanisms to replicate such structs dynamically per process. Second, some allocations are per-cpu variables allocated at boot-time. Finally, the per-process kernel stack is allocated from `vmalloc` during fork. Perspective tracks it and adds it to the process DSV.

Generating instruction speculation views. To generate ISVs, we have designed a static and dynamic analysis framework. Specifically, for static ISVs we identify system call paths used by a given binary by extending Radare2 [122], a state-of-the-art binary analysis tool. Second, we identify the function path for a given system call. Our framework combines the two to statically define ISVs at the instruction granularity per program. For dynamic analysis, we rely on the tracing subsystem of Linux to dynamically identify the system calls and their function paths in the kernel on a per-process and container basis. The result is a dynamic ISV profile.

Improving Kernel Auditing. Perspective can significantly improve the performance of kernel auditing by drastically reducing the search space only within ISVs. There are various techniques to identify gadgets using static analysis, taint tracking, and fuzzing [74, 78, 121]. We select Kasper [78] for our implementation, a state-of-the-art speculative execution gadget scanner that uses taint analysis and fuzzing. Specifically, we improve Kasper's kernel auditing performance by modifying Kasper and the underlying kernel fuzzer Syzkaller [71] to bound their search space within the ISVs. The result is a drastically reduced search space (Chapter 8).

Enhancing ISVs with Auditing. Perspective can further leverage the results of kernel auditing to generate even stricter ISVs. Specifically, after Perspective generates ISVs and Kasper performs its gadget analysis, we leverage the results of the analysis to exclude all identified vulnerable functions from the ISV, further enhancing their security. While the kernel analysis is bound by the capabilities of the underlying toolchain, Kasper and Syzkaller for our implementation, Perspective's approach can be leveraged for other analysis techniques as well [74, 115, 121]. We present the security enhanced ISVs in Chapter 8.

6.2 Architectural Extensions

The goal of the hardware design of Perspective is to expose the speculation view metadata to the hardware and efficiently control speculation.

Exposing ISVs to Hardware. Perspective's ISVs are at the granularity of instructions, enabling Perspective to block the speculative execution of any instruction that does not belong to the ISV irrespective of its type. To expose ISVs to hardware, we propose a simple design that associates a page of code in the kernel with an ISV page through a fixed virtual address offset, enabling

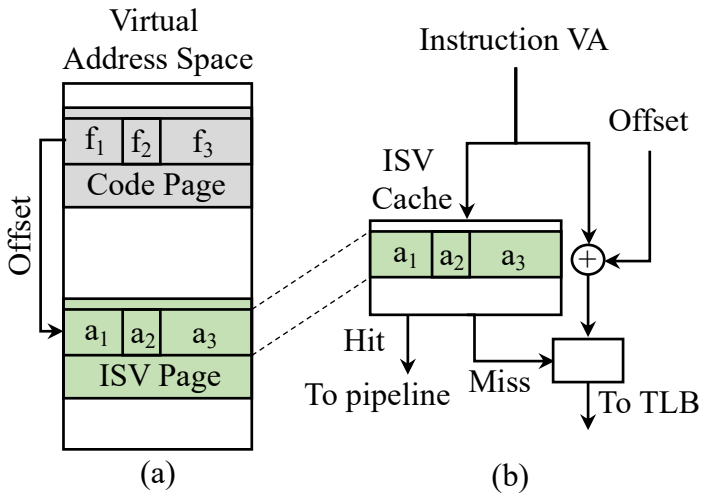

Figure 6.1: Perspective's (a) ISV VA layout and (b) ISV hardware cache.

fast ISV access. The offset is defined by the OS at process creation time. An ISV page holds an entry for each instruction at the same offset as in the code page. Each entry is a single bit that identifies whether that instruction belongs to the ISV. With this design Perspective only needs to populate ISV pages on demand based on usage when a first access to an ISV page is performed. Figure 6.1(a) shows the virtual address space and how the code pages are mapped to the ISV pages through the offset. Furthermore, Perspective maintains a hardware structure near the processor pipeline, called the ISV cache, that stores ISV entries. Figure 6.1(b) shows the ISV cache. The ISV cache is indexed by the instruction VA. On a hit, a response is returned to the pipeline to identify if Perspective needs to block speculative execution. In an ISV cache miss, the speculative execution of the instruction is blocked, and the instruction VA combined with the offset is sent to the TLB to locate the physical address of the ISV page. As we will see in the evaluation, due to the small amount of code accessed by applications during kernel execution, the ISV cache achieves a very high hit rate of 99%. To avoid ISV cache flushes on context switches, we tag entries with the address space identifier (ASID) similar to other tagged structures. Similar implementations have been proposed in prior work [137, 162] for other purposes.

Exposing DSVs to Hardware. Our goal is to minimize hardware changes in existing processors. To this end, Perspective implements DSV based on virtual ranges on a per-context basis inspired by upcoming secure processor extensions such as Intel's TDX [25]. Specifically, Perspective introduces a data structure called the Data Spection View Metadata Table (DSVMT), which is similar to TDX, and it is accessed in parallel to the TLB. Perspective leverages the DSVMT during kernel execution to identify DSV metadata. The DSVMT is organized as a three-level tree structure supporting the three contemporary page sizes (4KB, 2MB, 1GB). The leaf entry corresponds to 4KB pages. Each entry defines if a given page belongs in the DSV, and hence the DSVMT entries are only a single bit. Perspective further maintains a small DSVMT hardware cache located close to the processor pipeline that is checked on access. On a miss, instead of waiting for a refill, Perspective conservatively blocks speculation. Because DSVMT entries are very small, Perspective enjoys very high hit rates close to 99% as shown in Section 9.2. We believe that a realistic adoption path is for Perspective to extend secure architecture extensions

such as TDX. We opted against storing the DSV information in page tables because that would require intrusive changes in how the kernel currently uses page tables for the direct map, adding significant complexity to managing page tables and context switching.

Controlling Speculation. To control speculation, in Perspective the hardware adds a fence ahead of the instruction based on DSV and ISV. In this way, an attacker cannot observe side effects from speculative execution. A fenced instruction is allowed to proceed when it reaches the Visibility Point [156]: at the head of the ROB or when no older instructions can squash it. On DSV and ISV misses, instructions are blocked, and request is sent to the cache hierarchy after the instruction reaches its VP. On a hit, DSV and ISV LRU bits are not updated until the instruction reaches its VP.

Chapter 7

Methodology

Full-system Simulation. We model an out-of-order processor shown in Table 7.1 using cycle-level simulation with gem5 [57]. We evaluate the following defense schemes: (i) UNSAFE is the baseline architecture, which is not protected against transient execution attacks; (ii) FENCE is a hardware-only defense that delays all speculative loads until all prior branches are resolved; (iii) PERSPECTIVE-STATIC is the FENCE scheme augmented with the proposed Perspective hardware, running our modified Linux kernel. PERSPECTIVE-STATIC utilizes static ISVs; (iv) PERSPECTIVE is the same as PERSPECTIVE-STATIC, except it uses dynamic ISVs. (v) PERSPECTIVE++ extends PERSPECTIVE with stricter, ISV++, as discussed in Chapter 8.

Benchmarks and Applications. We evaluate the performance of Perspective on LEBench [125], a microbenchmark suite for Linux, as well as four datacenter applications, Redis [94], httpd [14] (Apache web server), nginx [124], and memcached [68]. Each workload runs within a container and Perspective leverages cgroups as discussed in Chapter 6. For LEBench, we run the ROI for each microbenchmark, and use the evaluation methodology of the original work [125]. For datacenter applications, the client and the server communicate over the loopback interface, which is less likely to bottleneck applications on I/O and represents the worst case for Perspective. We run httpd and nginx for 40K requests using ab [52]. We run Redis for 20K requests using redis-benchmark [95] and average over all individual tests. We run memcached for 160K requests using memslap [164]. We measure the percentage of time that applications spend in the OS during our experiments by tracking userspace/kernel time in the simulator: 50% for httpd, 65% for nginx, 65% for memcached, and 53% for Redis.

Table 7.1: Full-System Simulation Parameters.

Parameter	**Value**
Architecture	2 out-of-order x86 cores at 2.0 GHz
Core	8-issue, out-of-order, 62 Load Queue entries, 32 Store Queue entries, 192 ROB entries, L-TAGE branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 hardware prefetcher
Private L1-D Cache	32 KB, 64 B line, 8-way, 2 cycle RT latency, 1 hardware prefetcher
Shared L2 Cache	Slice: 2 MB, 64 B line, 16-way, 8 cycles RT latency
Coherence	Directory-based MESI protocol
DRAM	50 ns RT latency after L2
ISV Cache	128 entries, 32 sets, 4-way; 57 bits / entry
DSV Cache	128 entries, 32 sets, 4-way; 53 bits / entry
OS Kernel	Linux v5.4.49

Chapter 8

Security Evaluation

As discussed in Chapter 4, speculative execution attacks in the kernel can be categorized into active and passive attacks. This taxonomy is agnostic to a specific attack variant, as any variant such as Spectre v1, v2, BHI, Retbleed and others, can be exploited to perform either active or passive attacks. Since Perspective's design is rooted in this taxonomy, not attack variants, it can mitigate all variants.

8.1 Active Attacks Security Analysis

Active attacks are the most potent scenario as they are initiated directly by the attacker's kernel thread and are relatively simple to perform. Perspective completely eliminates active attacks. This is because Perspective's DSVs provide a strong security guarantee that an attacker cannot speculatively access kernel memory that has been allocated to other contexts. As a result, *any* speculative execution attack variant is blocked from performing active attacks. We leverage the identified CVEs in Table 4.1 to launch Proof-of-Concept (PoC) active attacks. Perspective's DSVs effectively mitigates all active attacks.

8.2 Passive Attacks Security Analysis

In passive attacks the victim process's kernel thread speculatively executes kernel functions containing transient execution gadgets to access and leak its own data. To this end, ISVs enable an execution context to define the set of kernel functions that are trusted by the context. Any transmitter instructions—whose execution could leak secrets, such as load instructions—that belong to kernel functions outside the ISVs are blocked from speculative execution.

Attack Surface Reduction. To assess the passive attack surface reduction of Perspective, we measure the surface as the number of functions that can be speculatively executed in the Linux kernel. Table 8.1 reports the relative reduction in the attack surface after applying ISVs to different applications. Perspective's personalized ISVs reduce the attack surface by at least 90.9% across all evaluated benchmarks and applications. Using static ISVs (*ISV-S*), Perspective reduces the number of functions that can be speculatively executed to only an average of 9% to that of

Table 8.1: Attack surface reduction with Perspective.

Config	LEBench	httpd	nginx	memcached	redis
ISV-S	92%	91%	90%	91%	90%
ISV	96%	94%	94%	96%	95%

Table 8.2: Perspective's MDS/Port/Cache gadget reduction.

Benchmark	ISV-S	ISV	ISV++
LEBench	87% / 87% / 83%	93% / 93% / 93%	100% / 100% / 100%
httpd	84% / 84% / 78%	91% / 92% / 92%	100% / 100% / 100%
nginx	84% / 83% / 78%	91% / 91% / 91%	100% / 100% / 100%
memcached	84% / 84% / 78%	92% / 92% / 92%	100% / 100% / 100%
redis	83% / 83% / 78%	91% / 91% / 91%	100% / 100% / 100%

Linux. With dynamic ISVs (*ISV*), this number is further reduced to an average of just 4.9% of Linux. This is because dynamic ISVs eliminate functions that static analysis unnecessarily includes. As a result, passive transient execution attacks of any variant are blocked within 95.1% of the kernel functions. In summary, these results highlight the significant reduction in the attack surface achieved by Perspective through personalized ISVs with system call interposition.

Search Space and Transient Execution Gadgets.[1] Perspective's ISVs reduce the number of speculative accessible functions to 5% of the entire kernel, hence, the gadget search space is reduced from 28K functions in Linux down to only 1.4K. To better understand the potential security gains of this reduction, we leverage Kasper [78], a state-of-the-art transient execution gadget scanner that uses taint analysis and fuzzing via Syzkaller [71]. Specifically, we (a) quantify the potential transient execution gadgets that Perspective's ISVs eliminate, (b) leverage the reduced search space of ISVs to speed up the gadget search process, and (c) utilize the results of the analysis to produce even stricter ISVs.

First, based on the results obtained from the authors of Kasper, we calculate the number of gadgets that are blocked from transient execution attacks with Perspective's ISVs. Kasper [78] identifies 1533 potential speculative execution gadgets in the whole Linux kernel. It identifies 805 microarchitectural buffers (MDS), 509 port contention (Port), and 219 cache-based covert channels (Cache) potential gadgets. Perspective blocks all transiently accessible gadgets excluded from the ISVs which account for 78% with static ISVs (*ISV-S*) and more than 91% with dynamic ISVs (*ISV*). Although our threat model (Chapter 3) does not include MDS attacks, as they are fixed in current processors, Perspective's ISVs still provide significant reduction. Overall, the results demonstrate the ability of Perspective to drastically reduce the attack surface of the Linux kernel by reducing the accessible gadgets.

Next, we leverage the security guarantee that kernel functions outside ISVs are blocked from speculative execution to limit Kasper's gadget scanning to only the kernel functions that are

[1]The search space and transient execution gadgets section presented in this thesis was developed in collaboration with David Rudo, as part of our joint research work published in Perspective [82].

within ISVs. Figure 9.1 shows the results. Perspective speeds up Kasper's gadget discovery rate (gadgets/hour) by 1.57× on average. We see that the speedups are even more profound in applications such as nginx reaching 2.23×. In summary, we have shown that Perspective is able to speed up existing kernel analysis techniques and potentially open up new possibilities for other solutions that are currently not considered possible or tractable for large OS code bases.

We further leverage the analysis of Kasper augmented with Perspective to produce even stricter dynamic ISVs as described in Section 5.3 and Section 6.1. Specifically, we use the analysis of Kasper to remove all identified vulnerable functions from the ISVs. The resulting ISVs (*ISV++*) block all identified gadgets as shown in Table 8.2.

Finally, we evaluate ISV protection with practical attacks by launching Proof-of-Concept (PoC) passive attacks using transient execution gadgets in CVEs of Table 4.1. Because the transient execution gadgets in those CVEs are excluded from the ISVs, the attacker cannot coerce the victim to speculatively execute them, blocking passive attacks.

Chapter 9

Performance Evaluation

This chapter presents the performance evaluation of Perspective. It describes the performance results across different benchmarks and applications (Section 9.1), and detailed sensitivity analysis of Perspective that sheds more light into Perspective's design (Section 9.2).

9.1 Performance Results

LEBench. Figure 9.2 shows the latency of LEBench tests on all schemes, including FENCE, PERSPECTIVE-STATIC, PERSPECTIVE, PERSPECTIVE++, normalized to UNSAFE. Among all the evaluated schemes, FENCE has the highest average execution overhead of 47.5%. In system calls such as `select` and `poll`, the overhead of FENCE can be as high as 228.3%. This is because the execution spins in the kernel repeatedly, paying the cost of mitigations. On the other hand, PERSPECTIVE-STATIC, PERSPECTIVE, and PERSPECTIVE++ exhibit very little execution overhead in most cases, with an average overhead of 4.1%, 3.6%, and 3.5% respectively. Perspective schemes only experience moderate overhead in `big-fork` and `page-fault`, where most of the overhead originates from DSVs as new allocations are created. Overall, Perspective shows little overhead in LEBench.

Datacenter Applications. Figure 9.3 shows the throughput of datacenter applications in requests per second normalized to UNSAFE. The UNSAFE baseline throughput is 11.5K, 18K, 55K, and 40.7K RPS for httpd, nginx, memcached, and redis respectively. The results show that FENCE can induce significant slowdowns, with an average overhead on throughput of 5.7%. The key-value stores, `memcached` and `Redis`, suffer the most due to the high cost of frequent blocking of memory accesses. PERSPECTIVE-STATIC, PERSPECTIVE, and PERSPECTIVE++ achieve nearly the same performance as UNSAFE, with an average overhead on throughput of 1.3%, 1.2%, and 1.2% respectively. In the case of `httpd`, PERSPECTIVE-STATIC performs a bit worse than PERSPECTIVE with dynamic ISVs, due to an increased number of indirect jumps whose targets cannot be statically analyzed and included in static ISVs, as we discuss in Section 9.2.

Comparing to Spot Software Mitigations. We further compare to software mitigations (KPTI [73] and Retpoline [142]) in Linux, which are spot mitigations only for Meltdown and Spectre-v2.

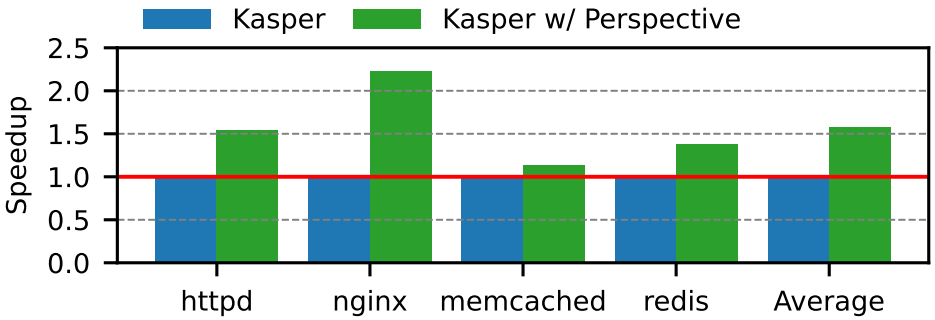

Figure 9.1: Speedup of Kasper's gadget discovery rate (gadgets/hour).

They show an average overhead of 14.5% in LEBench and 5% on datacenter applications. Without KPTI, the performance overhead of spot mitigations is 6.6% and 1.2% for micro and macro benchmarks respectively. Perspective provides better security than spot software mitigations and only shows an overhead of 3.5% and 1.2% respectively.

Comparing to Hardware Mitigations. To provide a better understanding of Perspective, we further compare it to two state-of-the-art hardware-only schemes: Delay-on-Miss (DOM) [127] and Speculative Taint Tracking (STT) [158]. They represent two design points in the spectrum between hardware complexity and performance, with DOM being simpler and slower, and STT more complex and faster.

Compared to the unsafe baseline, DOM shows an average overhead of 23.1%, while STT shows an average overhead of 3.7% in the microbenchmarks. The overhead profile of both DOM and STT is similar to FENCE, where several benchmarks like those based on select, poll, and epoll show overheads of 204% for DOM and 26.4% for STT. On the other hand, Perspective shows an average overhead of 3.5% on microbenchmarks. Overall, in microbenchmarks, Perspective is significantly faster than DOM and slightly faster than STT. In terms of macrobenchmarks, all three configurations show good performance, achieving normalized throughput of 98.3%, 99.6%, and 98.8% for DOM, STT, and Perspective respectively. Overall, all three configurations show marginal overhead in practice.

In terms of complexity, STT requires complex taint tracking additions to the processor pipeline, making its adoption difficult in practice. On the other hand, DOM requires simpler modifications, however its performance overhead can be significant. Perspective represents a balanced design with minimal hardware changes compared to DOM and STT while achieving low overhead. We believe that Perspective achieves its goal of attaining high performance while enabling software to take control of transient execution in the kernel.

9.2 Sensitivity Analysis

Breakdown of Speculation Views. Table 10.1 shows the percentage of fenced instructions that were caused by Perspective's DSVs and static, dynamic ISVs, and ISV++. Such fences would represent false positives due Perspective's implementation since we are focusing on be-

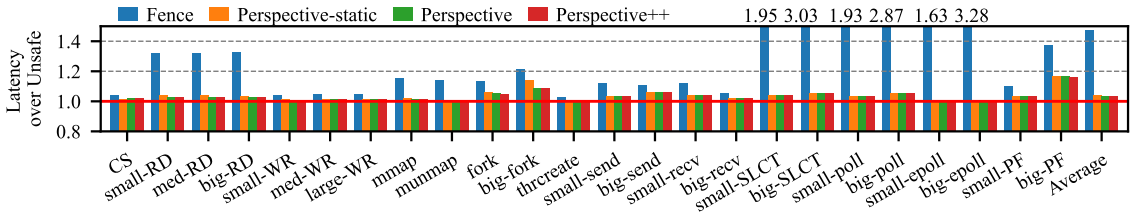

Figure 9.2: LEBench suite's normalized latency to an unsafe baseline under different schemes.

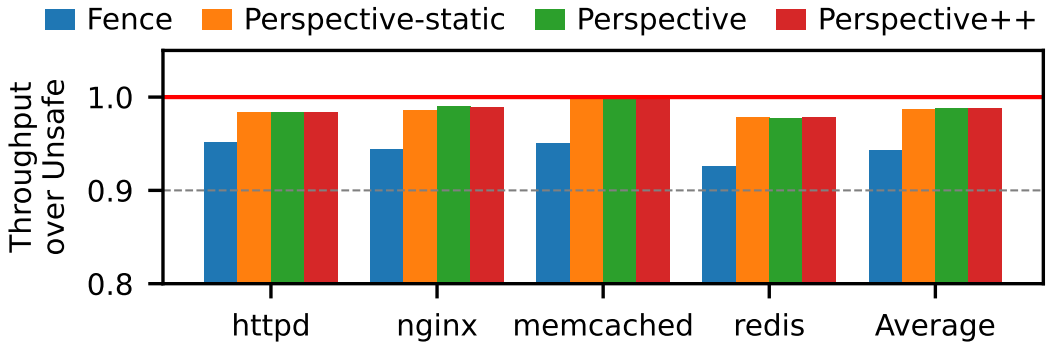

Figure 9.3: Requests per second normalized to an unsafe baseline.

nign workloads. Both LEBench and applications exhibit similar profiles. The results show that on average, the *ISV-Static* accounts for 20% of the total number of fenced instructions, while the DSV accounts for 80%. In *ISV-Dynamic*, the ISV percentage decreases to 18%. This is because dynamic ISVs may capture reachable functions that static ISVs cannot. The fence rate is on average 9 and 37 fences per kilo instructions for ISVs and DSVs, respectively. *ISV++* exhibits similar characteristics.

Unknown Allocations. As discussed in Section 6.1, some allocations beyond those handled by the page and slab allocators may be classified as *unknown*. Accessing memory from unknown allocations can cause stalls in the processor pipeline. We quantify their impact by selectively disabling blocking of unknown accesses. On average, unknown allocations in the kernel's direct map are responsible for 1.5% of Perspective's overhead on LEBench and marginal overhead on applications. We believe that additional engineering efforts can further reduce this overhead.

Hardware Structures. Perspective introduces the ISV and DSV caches. Both caches are organized as 128-entry 4-way set associative caches and provide very high hit rates close to 99% for both microbenchmarks and applications. The reason is that a small cache can hold the relatively small working set of instructions and data in the kernel. Table 9.1 show the characteristics of Perspective's hardware structures: area, access time, energy, and leakage power. The data correspond to 22nm and is obtained with CACTI [54]. Although the access time to Perspective structures is a fraction of a cycle, to be conservative we account for 2 cycles.

Memory Fragmentation. Perspective utilizes separate physical pages to store objects for different `cgroups`. This approach provides strong isolation, but it can lead to increased memory

Table 9.1: Hardware Structure Characterization.

Configuration	Area	Access Time	Dyn. Energy	Leak. Power
DSV Cache	0.0024 mm2	114 ps	1.21 pj	0.78 mW
ISV Cache	0.0025 mm2	115 ps	1.29 pj	0.79 mW

fragmentation. We use `slabtop` to track the memory utilization ratio, defined as the size of active objects divided by the total size of objects. On average, Perspective's secure slab allocator incurs a minimal memory usage overhead of 0.91%.

Domain Reassignment. When a slab object is freed, it is typically returned to its original slab for future allocations. However, if all objects in a slab become free, the entire page is returned to the buddy allocator for general use. In Perspective's secure slab allocator, these page-level operations require domain reassignment to ensure security isolation between different contexts.

We analyzed the percentage and frequency of page-level operations caused by slab object frees in Perspective's secure slab allocator. The percentage of page-level operations in the slab allocator is low across all workloads. For redis, 0.23% of slab object frees result in a page being returned to the buddy allocator. Other applications exhibit even lower ratios, with httpd, nginx, and memcached showing ratios of 0.01%, 0.01%, and 0.003% respectively. The absolute frequency of these page-level operations also remains low. For redis, 96 slab object frees result in a domain reassignment per second. Other applications show much lower frequencies, with httpd, nginx, and memcached showing 4, 3, 2 occurrences per second, respectively. The low percentage and frequency of page-level operations suggests that the overhead of domain reassignment during page allocation and freeing is minimal. The majority of slab operations occur at the object level within already-allocated pages, allowing Perspective to maintain its strict isolation guarantees without significant performance impact.

Chapter 10

Related Work

Numerous hardware-only defense mechanisms have been proposed [47, 62, 63, 72, 81, 85, 97, 110, 112, 126, 127, 139, 140, 150, 156, 158, 159, 163] to defend against transient execution attacks. Invisible speculation schemes such as InvisiSpec [156], SafeSpec [81], DAWG [85], DOM [127], conditional speculation [97], MuonTrap [48], CleanupSpec [126], GhostMinion [47] focus primarily on blocking cache-based covert channels through techniques like shadow caches or cache partitioning, but do not comprehensively protect against all speculative channels. For example, DOM (Delay-on-Miss) proposes delaying speculative loads that miss in the L1 cache until they become non-speculative. Restrictive speculation schemes like CSF [139], NDA [150], STT [158], SPT [62], SDO [159] take a more comprehensive approach to block all speculative covert channels by selectively delaying unsafe instructions defined by the scheme. For instance, STT introduces speculative taint tracking based on information flow tracking techniques to delay only transmitter instructions that depend on speculatively accessed data. Recent optimizations like Pinned Loads focus on improving performance by resolving memory consistency violations earlier to accelerate the advancement of visibility points. While hardware defenses offer great software transparency, they lack the necessary information to determine which data and execution paths require protection. Consequently, they either conservatively protect *all* speculative instructions that could leak information [127, 150, 156], or conservatively infer what data is not confidential [62, 158], resulting in high execution overhead or complex hardware.

Prior work on system call interposition [2, 3, 5, 67, 77, 83, 96, 136] has focused solely on application sandboxing to protect the kernel against traditional attacks such as buffer overflows. Similarly, a body of work has focused on architectural support for memory-based protection [64, 128, 134, 148, 153, 154]. These works focus on traditional attacks rather than transient execution attacks.

Kernel mitigations have primarily focused on page table isolation to mitigate unauthorized data accesses. KPTI separates userspace and kernel page tables in order to mitigate Meltdown-style attacks. Other software-only approaches [50, 56, 80, 130, 155] also rely on page tables to block speculation through page faults. However, these approaches require context switches, leading to substantial overhead. Furthermore, when access to unmapped data is required, the kernel needs to switch to the complete mappings, voiding isolation [56].

Control flow integrity (CFI) [45, 65, 111, 116, 117, 118, 132, 146, 160, 161] is a powerful technique that can reduce the attack surface of the kernel. CFI prevents attackers from diverting

Table 10.1: Percentage of fenced instructions due to ISV and DSV.

Config	LEBench	httpd	nginx	memcached	redis
ISV-S/DSV	27% / 73%	20% / 80%	15% / 85%	13% / 87%	23% / 77%
ISV/DSV	22% / 78%	12% / 88%	16% / 84%	15% / 85%	23% / 77%
ISV++/DSV	22% / 78%	13% / 87%	16% / 84%	15% / 85%	23% / 77%

the victim's control flow to attacker-chosen logic. Recent works, such as SpecCFI [89], leverage CFI to restrict speculative execution, increasing the bar for attacks. Such techniques are orthogonal to Perspective. First, Perspective introduces DSVs to protect data accesses, which is not the goal of SpecCFI, CET [132], and CFI in general. Second, SpecCFI and CET would mark all kernel functions as legal, and hence speculation would be able to reach any function irrespective of the userspace process making the system call. This leaves a large attack surface vulnerable. ISVs provide tailored protection to applications, as an attacker cannot hijack the control flow to speculatively execute a function outside the ISVs. We demonstrate this point in Section 8.2 as Perspective drastically reduces the attack surface of the kernel.

Chapter 11

Conclusion

This thesis presents *Perspective*, a principled framework for secure speculation in the operating system. Perspective introduces Data Speculation Views (DSVs) and Instruction Speculation Views (ISVs) to mitigate active and passive attacks, respectively. DSVs define the ownership of kernel data by a given execution context. ISVs define the set of kernel functions that can be speculatively executed by a given execution context. Perspective's ISVs open up the opportunities of (i) swiftly patching gadgets in kernel functions, (ii) reducing the surface of passive attacks, and (iii) speeding up the kernel auditing process. Our evaluation showed that Perspective achieves significant security improvements with minimal performance overhead.

Bibliography

[1] Linux kernel stable tree. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=linux-6.1.y. 1

[2] OpenBSD Pledge. https://man.openbsd.org/pledge. 2.3, 10

[3] OpenBSD Tame. https://man.openbsd.org/OpenBSD-5.8/tame.2. 1.1, 2.3, 10

[4] PROCESS MITIGATION SYSTEM CALL DISABLE POLICY structure. https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_system_call_disable_policy. 2.3

[5] Improving host security with system call policies. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association. URL https://www.usenix.org/conference/12th-usenix-security-symposium/improving-host-security-system-call-policies. 1.1, 10

[6] Vulnerability details: Cve-2018-15572. https://www.cvedetails.com/cve/CVE-2018-15572/, Aug 2018. 1

[7] Vulnerability details: Cve-2018-15594. https://www.cvedetails.com/cve/CVE-2018-15594/, Aug 2018. 1

[8] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2018-12126, May 2019. 1

[9] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2018-12127, May 2019. 1

[10] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2018-12130, May 2019. 1

[11] Vulnerability details: Cve-2019-15902. https://www.cvedetails.com/cve/CVE-2019-15902/, Sept 2019. 1, 4.1, 4.2

[12] Vulnerability details: Cve-2019-18660. https://www.cvedetails.com/cve/CVE-2019-18660/, Nov 2019. 1, 4.1

[13] Vulnerability details: Cve-2019-19338. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19338, Nov 2019. 1

[14] Apache HTTP Server Project. https://httpd.apache.org/, 2019. 7

[15] Vulnerability details: Cve-2020-10766. https://www.cvedetails.com/cve/CVE-2020-10766/, Sept 2020. 1

[16] Vulnerability details: Cve-2020-10767. https://www.cvedetails.com/cve/CVE-2020-10767/, Sept 2020. 1, 4.1

[17] Vulnerability details: Cve-2020-10768. https://www.cvedetails.com/cve/CVE-2020-10768/, Sept 2020. 1

[18] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2021-26401, Jun 2020. 1, 4.1

[19] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2021-29154, May 2021. 1

[20] Vulnerability details: Cve-2021-31829. https://www.cvedetails.com/cve/CVE-2021-31829/, May 2021. 1, 4.1

[21] Vulnerability details: Cve-2021-33624. https://www.cvedetails.com/cve/CVE-2021-33624/, Jun 2021. 1, 4.1

[22] Vulnerability details: Cve-2021-34556. https://www.cvedetails.com/cve/CVE-2021-34556/, Aug 2021. 1

[23] Vulnerability details: Cve-2021-35477. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-35477, Jun 2021. 1

[24] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2021-38300, Oct 2021. 1

[25] Intel trusted domain extentions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf, 2021. 6.2

[26] Vulnerability details: Cve-2019-7308. https://www.cvedetails.com/cve/CVE-2019-7308/, Jul 2022. 1, 4.1

[27] Vulnerability details: Cve-2020-27170. https://www.cvedetails.com/cve/CVE-2020-27170/, Jul 2022. 1, 4.1

[28] Vulnerability details: Cve-2020-27171. https://www.cvedetails.com/cve/CVE-2020-27171/, Jul 2022. 1, 4.1

[29] Vulnerability details: Cve-2021-29155. https://www.cvedetails.com/cve/CVE-2021-29155/, Apr 2022. 1, 4.1

[30] Cve-2022-0001 details. https://nvd.nist.gov/vuln/detail/CVE-2022-0001, Mar 2022. 4.1

[31] Cve-2022-0002 details. https://nvd.nist.gov/vuln/detail/CVE-2022-0002, Mar 2022. 4.1

[32] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2022-21123, Jul 2022. 1

[33] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2022-21125, Jul 2022. 1

[34] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2022-21166, Jul 2022. 1

[35] Cve-2022-23824 details. https://nvd.nist.gov/vuln/detail/CVE-2022-23824, Nov 2022. 1, 4.1

[36] Cve-2022-23960 details. https://nvd.nist.gov/vuln/detail/CVE-2022-23960, Mar 2022. 4.1

[37] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2022-26373, Jul 2022. 1

[38] Vulnerability details: Cve-2022-27223. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-27223, Mar 2022. 1, 4.1, 4.2

[39] Cve-2022-29900 details. https://nvd.nist.gov/vuln/detail/CVE-2022-29900, Jul 2022. 4.1

[40] Cve-2022-29901 details. https://nvd.nist.gov/vuln/detail/CVE-2022-29901, Jul 2022. 4.1

[41] Linux kernel cves. https://www.linuxkernelcves.com/cves/CVE-2019-11091, Jan 2023. 1

[42] Cve-2022-2196 details. https://nvd.nist.gov/vuln/detail/CVE-2022-2196, Jan 2023. 4.1

[43] Cve-2022-42331 details. https://nvd.nist.gov/vuln/detail/CVE-2022-42331, Mar 2023. 1

[44] Cve-2023-1998 details. https://nvd.nist.gov/vuln/detail/CVE-2023-1998, Apr 2023. 4.1

[45] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595932267. doi: 10.1145/1102120.1102165. URL https://doi.org/10.1145/1102120.1102165. 10

[46] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, 2020. USENIX Association. 2.3

[47] Sam Ainsworth. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 592–606, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480074. URL https://doi.org/10.1145/3466752.3480074. 1, 10

[48] 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. ISBN 9781728146614. doi: 10.1109/ISCA45697.2020.00022. URL https://doi.org/10.1109/ISCA45697.2020.00022. 10

[49] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. IEEE Symposium on Security and Privacy (S&P), 2019. 2.2

[50] Alexandre Chartre. Kernel Address Space Isolation. https://lkml.iu.edu/hypermail/linux/kernel/1907.1/03688.html. 1, 10

[51] AMD. Software Techniques for Managing Speculation. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/software-techniques-for-managing-speculation.pdf. 1

[52] Apache. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2022. 7

[53] AWS. Firecracker Design. https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md. 2.3

[54] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization*, 14(2):14:1–14:25, June 2017. ISSN 1544-3566. 9.2

[55] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*, August 2022. URL Paper=http://download.vusec.net/papers/bhi-spectre-bhb_sec22.pdfWeb=https://www.vusec.net/projects/bhi-spectre-bhbCode=https://github.com/vusec/bhi-spectre-bhb. Intel Bounty Reward. 1.1, 3, 4.1, 4.1, 4.2

[56] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1139–1154. USENIX Association, 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/behrens. 1, 10

[57] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 2011. 7

[58] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652. 5.2

[59] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic eval-

uation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/canella. 1, 2.2

[60] 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 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 769–784, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3363219. URL https://doi.org/10.1145/3319535.3363219. 1

[61] Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018. 1

[62] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. *Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy*, page 607–622. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450385572. URL https://doi.org/10.1145/3466752.3480068. 1, 10

[63] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure TLBs. In *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2019. 1, 10

[64] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1):487–502, mar 2015. ISSN 0163-5964. doi: 10.1145/2786763.2694383. URL https://doi.org/10.1145/2786763.2694383. 10

[65] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 131–148, USA, 2017. USENIX Association. ISBN 9781931971409. 10

[66] Docker. Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/. 2.3

[67] Jake Edge. A seccomp overview. https://lwn.net/Articles/656307/, September 2015. 1.1, 2.3, 10

[68] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004. 7

[69] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1871–1885, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. 1

[70] Google. gVisor: Container Runtime Sandbox. https://github.com/google/gvisor/blob/master/runsc/boot/filter/config.go, . 2.3

[71] Google. Syzkaller - Kernel Fuzzer. https://github.com/google/syzkaller, . 6.1, 8.2

[72] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proc. of the USENIX Security Symposium (USENIX)*, 2017. 1, 10

[73] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. pages 161–176, 06 2017. ISBN 978-3-319-62104-3. doi: 10.1007/978-3-319-62105-0_11. 1, 9.1

[74] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows, 2019. 5.4, 6.1

[75] Intel. Indirect branch restricted speculation, . https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html. 4.2

[76] Intel. Refined Speculative Execution Terminology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html, . 4.1

[77] K. Jain and R. C. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Network and Distributed System Security Symposium*, 2000. 1.1, 10

[78] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*, April 2022. 5.1, 5.4, 6.1, 8.2, 8.2

[79] Julien Tinnes. Introducing Chrome's next-generation Linux sandbox. https://blog.cr0.org/2012/09/introducing-chromes-next-generation.html, September 2012. 2.3

[80] Junaid Shahid. Address Space Isolation for KVM. https://lore.kernel.org/lkml/20220223052223.1202152-1-junaids@google.com/. 1, 10

[81] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19)*, 2019. 1, 10

[82] Tae Hoon Kim, David Rudo, Kaiyang Zhao, Zirui Neil Zhao, and Dimitrios Skarlatos. Perspective: A principled framework for pliable and secure speculation in operating systems. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 739–755, 2024. doi: 10.1109/ISCA59077.2024.00059. 1.1, 2, 1

[83] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference (USENIX ATC*

13), pages 139–144, San Jose, CA, June 2013. USENIX Association. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim. 1.1, 10

[84] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. doi: 10.1109/ISCA.2014.6853210. 3

[85] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. 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, Los Alamitos, CA, USA, oct 2018. IEEE Computer Society. doi: 10.1109/MICRO.2018.00083. URL https://doi.ieeecomputersociety.org/10.1109/MICRO.2018.00083. 1, 10

[86] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2399–2416. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/kirzner. 1, 4.1

[87] Paul Kocher, 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 *IEEE Symposium on Security and Privacy (S&P)*, 2019. 1, 1.1, 2.2, 5, 2.2, 3, 4.1, 4.1, 4.2

[88] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. USENIX Association, 2018. 1.1, 2.2, 3, 4.1, 4.1, 4.2

[89] 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 *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53, 2020. doi: 10.1109/SP40000.2020.00033. 5.1, 10

[90] Kubernetes Documentation. Configure a Security Context for a Pod or Container. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, July 2019. 2.3

[91] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. In *Proceedings of the 2020 ACM SIGMETRICS Conference (SIGMETRICS'20)*, June 2020. 1.1, 5.1

[92] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *Network and*

Distributed System Security Symposium, 2013. 1.1, 5.1

[93] Amund Bergland Kvalsvik, Pavlos Aimoniotis, Stefanos Kaxiras, and Magnus Själander. Doppelganger loads: A safe, complexity-effective optimization for secure speculation schemes. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589088. URL https://doi.org/10.1145/3579371.3589088. 1

[94] Redis Labs. Redis In-Memory Data Structure. https://redis.io, 2022. 7

[95] Redis Labs. Redis-Benchmark. https://redis.io/topics/benchmarks, 2022. 7

[96] Bo Li, Jianxin Li, Tianyu Wo, Chunming Hu, and Liang Zhong. A vmm-based system call interposition framework for program monitoring. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 706–711, 2010. doi: 10.1109/ICPADS.2010.53. 1.1, 10

[97] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *International Symposium on High Performance Computer Architecture*, 2019. 1, 10

[98] Linux. Kernel Speculation Defenses. https://www.kernel.org/doc/Documentation/speculation.txt, . 1

[99] Linux. The Linux Core Scheduling Documentation. https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/core-scheduling.rst, . 1

[100] Linux. The Linux Kernel Documentation. https://docs.kernel.org/admin-guide/hw-vuln/l1d_flush.html, . 1

[101] Linux. Linux Memory Management - Complete virtual memory map. https://docs.kernel.org/arch/x86/x86_64/mm.html, . 1, 2.3, 4.1

[102] Linux. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v5.1/userspace-api/seccomp_filter.html, . 2.3

[103] Linux. slabinfo(5) - Linux manual page. https://man7.org/linux/man-pages/man5/slabinfo.5.html, . 5.2, 6.1

[104] Linux. slab_common.c - Linux suource code. https://elixir.bootlin.com/linux/v6.7.5/source/mm/slab_common.c#L809, . 5.2, 6.1

[105] Linux. slub.c - Linux suource code. https://elixir.bootlin.com/linux/v6.7.5/source/mm/slub.c, . 5.2

[106] Linux. Control Group v2. https://docs.kernel.org/admin-guide/cgroup-v2.html, 2015. 6.1

[107] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security'18*, 2018. 1, 2.2, 3

[108] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 643–660, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/lipp. 3

[109] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015. doi: 10.1109/SP.2015.43. 1, 2.2

[110] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016. 1, 10

[111] LLVM. Clang Documentation on Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html. 10

[112] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1397–1414. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin. 1, 10

[113] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2109–2122. ACM, 2018. doi: 10.1145/3243734.3243761. 1.1, 2.2, 3, 4.1, 4.1, 4.2

[114] Mesos. Linux Seccomp Support in Mesos Containerizer. http://mesos.apache.org/documentation/latest/isolators/linux-seccomp/. 2.3

[115] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 72–86, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527412. URL https://doi.org/10.1145/3470496.3527412. 6.1

[116] Ben Niu and Gang Tan. Modular control-flow integrity. *SIGPLAN Not.*, 49(6):577–587, jun 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594295. URL https://doi.org/10.1145/2666356.2594295. 10

[117] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 577–587, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594295. URL https://doi.org/10.1145/2594291.2594295. 10

[118] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page

914–926, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813644. URL https://doi.org/10.1145/2810103.2813644. 10

[119] P. Lawrence. Seccomp filter in Android O. https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html, July 2017. 2.3

[120] Aimoniotis Pavlos, Kvalsvik Bergland Amund, Chen Xiaoyue, Själander Magnus, and Kaxiras Stefanos. Recon: Efficient detection, management, and use of non-speculative information leakage. In *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, 2023. 1

[121] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021. 5.4, 6.1

[122] radareorg. Libre Reversing Framework for Unix Geeks. https://github.com/radareorg/radare2, 2023. 6.1

[123] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: Attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 685–698, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527429. URL https://doi.org/10.1145/3470496.3527429. 1

[124] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008. 7

[125] 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 '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359640. URL https://doi.org/10.1145/3341301.3359640. 7

[126] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019. 1, 10

[127] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, 2019. 1, 9.1, 10

[128] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. Practical Byte-Granular Memory Blacklisting Using Califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019. 10

[129] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. 1

[130] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *CoRR*, abs/1905.09100, 2019. URL `http://arxiv.org/abs/1905.09100`. 1, 10

[131] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic, and Milos Prvulovic. A new side-channel vulnerability on modern computers by exploiting electromagnetic emanations from the power management unit. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 123–138, 2020. doi: 10.1109/HPCA47549.2020.00020. 3

[132] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372268. doi: 10.1145/3337167.3337175. URL `https://doi.org/10.1145/3337167.3337175`. 10

[133] Singularity. Security Options in Singularity. `https://sylabs.io/guides/3.0/user-guide/security_options.html`. 2.3

[134] Kanad Sinha and Simha Sethumadhavan. Practical Memory Safety with REST. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018. 10

[135] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, 2019. 1

[136] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. Draco: Architectural and Operating System Support for System Call Security. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, October 2020. 1.1, 10

[137] Dimitrios Skarlatos, Zirui Neil Zhao, Riccardo Paccagnella, Christopher W. Fletcher, and Josep Torrellas. Jamais Vu: Thwarting Microarchitectural Replay Attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021. 6.2

[138] SUSE. Security hardening: Use of ebpf by unprivileged users has been disabled by default. https://www.suse.com/support/kb/doc/?id=000020545. 4.2

[139] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019. 1, 10

[140] Daniel Townley and Dmitry Ponomarev. Smt-cop: Defeating side-channel attacks on execution units in smt processors. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019. 1, 10

[141] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *CoRR*, abs/1802.03802, 2018. URL `http://arxiv.org/abs/1802.03802`. 1

[142] Paul Turner. Retpoline: a Software Construct for Preventing Branch-target-injection. `https://support.google.com/faqs/answer/7625886`, 2018. 1, 4.1, 4.2, 9.1

[143] Ubuntu. LXD. `https://help.ubuntu.com/lts/serverguide/lxd.html#lxd-seccomp`. 2.3

[144] Ubuntu. Microarchitectural Data Sampling (MDS). `https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/MDS`. 3

[145] 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 *USENIX Security'18*, 2008. 1

[146] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953, 2016. doi: 10.1109/SP.2016.60. 10

[147] 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 *S&P*, May 2019. 1

[148] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting Software with Code-Centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014. 10

[149] 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. 1

[150] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358306. URL `https://doi.org/10.1145/3352460.3358306`. 1, 10

[151] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *31th USENIX Security Symposium (USENIX Security 22)*. USENIX Association, August 2022. 1, 1.1, 3, 4.1, 4.2

[152] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security*, August 2022. URL `Paper=https://comsec.ethz.ch/wp-content/files/retbleed_sec22.pdfURL=https://comsec.ethz.ch/research/microarch/retbleed`. Intel Bounty Reward, CSAW Europe finalist. 4.1, 4.2

[153] 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 X, page 304–316, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135742. doi: 10.1145/605397.605429. URL `https://doi.org/10.1145/605397.605429`. 10

[154] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014. 10

[155] Hongyan Xia, David Zhang, Wei Liu, Istvan Haller, Bruce Sherwin, and David Chisnall. A secret-free hypervisor: Rethinking isolation in the age of speculative vulnerabilities. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 370–385, 2022. doi: 10.1109/SP46214.2022.9833726. 1, 10

[156] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018. 1, 3, 6.2, 10

[157] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security'14*, 2014. 1, 2.2, 2.2

[158] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 954–968, 2019. 1, 9.1, 10

[159] 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. doi: 10.1109/ISCA45697.2020.00064. 1, 10

[160] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013. doi: 10.1109/SP.2013.44. 10

[161] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 337–352, USA, 2013. USENIX Association. ISBN 9781931971034. 10

[162] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. Speculation invariance (invarspec): Faster safe execution through program analysis. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1138–1152, 2020. doi: 10.1109/MICRO50266.2020.00094. 6.2

[163] Zirui Neil Zhao, Houxiang Ji, Adam Morrison, Darko Marinov, and Josep Torrellas. Pinned loads: Taming speculative loads in secure processors. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. Association for Computing Machinery, 2022. 1, 10

[164] Mingqiang Zhuang and Brian Aker. memaslap: Load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memaslap.html`, 2012. 7