Technical Report

UCAM-CL-TR-992
ISSN 1476-2986

Number 992

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Transient execution vulnerabilities in the security context of server hardware

Allison Randal

December 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 2023 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Robinson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-992*

Summary

Transient execution vulnerabilities in the security context of server hardware

Allison Randal

The thesis of this work is that eliminating speculation is a feasible approach to mitigating the transient execution vulnerabilities on large-scale server hardware. Many mitigations have been proposed and implemented for many variants of the transient execution vulnerabilities, and while the Meltdown-type exception-based transient execution vulnerabilities have proven to be tractable, Spectre-type vulnerabilities and other speculation-based transient execution vulnerabilities have been far more resistant to countermeasures. After years of research and development by academia and industry, eliminating speculation is still the only reliable countermeasure against Spectre.

For smaller-scale embedded systems or security-focused hardware such as a cryptographic system or a root-of-trust (RoT), eliminating speculation is widely accepted as a reasonable approach to improving security. But, for larger-scale and general-purpose hardware, eliminating speculation is often rapidly dismissed as inconceivable, though the claim that speculation is required for adequate performance is rarely supported by concrete performance results. The performance results we do have from several independent strands of research over the past few decades have shown that speculation features on large-scale server hardware do not offer the same performance advantages as on smaller-scale hardware, so eliminating speculation on large-scale server hardware does not harm performance as much as we might suspect. And selective speculation techniques have shown that speculation-based transient execution vulnerabilities can be mitigated by a partial elimination of speculation, so we can preserve some of the performance of speculation while subduing the security risk. In order to demonstrate the feasibility of eliminating speculation from modern server hardware microarchitectures, I consider three alternative approaches that partially or completely eliminate speculative execution.

Heterogeneous multicore systems that combine speculative and non-speculative cores make it possible to entirely disable speculation for security-critical or untrusted sections of code, by running that code on a non-speculative core. Code running on a speculative core performs as well as it would on a fully speculative hardware architecture. The systems software developer has the power to choose which code runs with the performance advantage of speculation, and which code runs with the security advantage of no speculation. However, heterogeneous multicores only offer the ability to disable speculation at the process or thread level. A finer-grained approach is desirable, to limit the performance penalty of disabled speculation to the smallest possible region of code.

Non-speculative cores keep the performance advantages of most common features

in modern hardware architectures—such as dynamic multiple issue, dynamic pipeline scheduling, out-of-order execution, and register renaming—while avoiding the risk of speculative execution. Such processors do not perform as well as equivalent speculative processors, but the results of this work indicate that they can perform as well or better than equivalent speculative processors with all relevant mitigations for the transient execution vulnerabilities applied. The performance penalty of eliminating speculation can also be partially offset by increasing the size of fetch and issue stage components in the pipeline. Non-speculative cores do not give systems software developers the option to choose between performance and security. However, these cores may be desirable for large-scale server deployments that exclusively serve privacy-centered workloads, such as processing hospital patient data.

Selective speculation combines speculative and non-speculative features on a single core. The performance of selective speculation cores is proportional to the use of speculative and non-speculative features, so only regions of code that disable speculation pay a performance penalty. Out of the three approaches considered in this dissertation, selective speculation cores are best for large-scale general-purpose server deployments, because they simplify resource allocation by keeping all cores identical, have no performance penalty for code run as entirely speculative, and give systems software developers the most precise control over speculation.

Acknowledgements

Thanks to Ravi Nair for help locating and scanning copies of several pivotal IBM papers on virtual machines from the 1960s that were no longer (or perhaps never) available in libraries or online. Thanks also to the reviewers for various stages of this work (alphabetically): Clint Adams, Matthew Allen, Ross Anderson, Alastair Beresford, Herbert Bos, James Bottomley, Peter Capek, Damian Conway, Kees Cook, Jon Crowcroft, Mike Dodson, Tony Finch, Greg Kroah-Hartman, Ronan Lashermes, Anil Madhavapeddy, Ronald Minnich, Richard Mortier, Mattias Nissler, Sandro Pinto, Ravi Sahita, Vedvyas Shanbhogue, Davanum Srinivas, Tom Sutcliffe, Zahra Tarkhani, Daniel Thomas, and Jonathan Woodruff. Their feedback was greatly appreciated.

Contents

Published work

In the course of this research, I published the following work that contributed directly to the contents of this dissertation:

- An early version of Chapter 2 was published as: *The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers*, in ACM Computing Surveys, February 2020 [351].
- A version of Chapter 3 has also been released as a preprint article: *This is How You Lose the Transient Execution War* [352].

Presentations

In the course of this research, I gave the following presentations related to this work:

- *Secure isolation in Rust: hypervisors, containers, and the future of composable infrastructure*, Code Mesh London, November 2018.
- *The ideal versus the real: A brief history of secure isolation in virtual machines and containers*, University of Cambridge Computer Laboratory Security Seminar, November 2018.
- *Ghosting the Spectre: fine-grained control over speculative execution*, INRIA Security Seminar, June 2022.
- *Ghosting the Spectre*, Open Infrastructure Summit, June 2022.
- *Side-Channel Attacks & Transient Execution Vulnerabilities*, RISC-V Summit, December 2022.

Development work

In the course of this research, the following development work contributed directly or indirectly to the contents of this dissertation:

In 2018, I developed a userspace library interface to the hardware virtualization features in the Linux Kernel provided by KVM. I chose Rust as the implementation language for this work to explore what effects its low-level memory safety features might have on the problem space of secure isolation in virtual machines. I subsequently ported this work to the Bhyve hypervisor on Illumos, and my work became the base for the Propolis [339] hypervisor userspace developed by Oxide Computer Company.

In 2021, I developed three SoC prototypes in RISC-V to explore the impact of several potential approaches to mitigating Spectre on virtualization and containers. Chapter 7 briefly discusses the contributions made by this work, and how subsequent work by others both supported the conclusions I reached, and also superseded my initial prototypes.

In 2022, I worked with a team of RISC-V engineers and with Moein Ghaniyoun, a PhD student at Ohio State University, to improve the implementation of RISC-V on gem5. We found that gem5 has substantial model inaccuracies specifically around branch prediction and memory loads, which is exactly where precision is needed in evaluating mitigations for Spectre. Independent work by Chatzopoulos *et al.* [85] confirmed our findings about gem5's model inaccuracies around branch prediction and memory loads, and explored gem5's limitations in much greater detail. Furthermore, Yang *et al.* [492] criticize the use of gem5 for evaluating Spectre mitigations, because gem5 only models just enough of the microarchitecture to approximate timing for performance evaluation, and intentionally does not model certain microarchitectural details that are critical to the transient execution vulnerabilities. The limitations of gem5 are not specific to any one architecture, they are fundamental to the simulator platform, and equally affect attempts at implementing Spectre mitigations on gem5's x86, ARM, or RISC-V simulations.

Chapter 1

Introduction

A new class of vulnerabilities related to speculative and out-of-order execution, fault-injection, and microarchitectural side channels rose to attention in 2018. The techniques behind the transient execution vulnerabilities were not new, but the combined application of the techniques was more sophisticated, and the security impact more severe, than previously considered possible. The models of secure isolation employed by server workloads such as virtual machines and containers offer little protection from the transient execution vulnerabilities. While the major server hardware vendors have applied some hardware and software mitigations for some known variants of these vulnerabilities, none of the major vendors have bothered to try to eliminate all variants, and the probability of further variants being discovered in the coming years is high. The not-very-well-kept secret of the industry is that public cloud providers generally run with many transient execution mitigations disabled, because it is not cost-effective to enable them.

For smaller-scale or security-focused hardware—such as embedded systems or dedicated cryptographic hardware—the easy and obvious solution to the transient execution vulnerabilities is to simply eliminate speculation entirely. Eliminating speculation ruptures the fundamental DNA of all Spectre-type attacks—blocking the initial fault-injection attack vector that makes these attacks so much more severe than previously known microarchitectural attacks, and blocking the transient microarchitecture states that leak secrets from ever being created in the first place—so hardware without speculation is proof against all currently known variants of Spectre and all variants that may be discovered in the future. Smaller-scale and security-focused hardware microarchitectures rarely implement speculation features anyway, because of resource constraints, because speculation provides too little performance benefit for the particular workloads running on the hardware, or because speculation radically increases the difficulty of verifying the security properties of the hardware. However, for general-purpose, medium-to-large scale hardware such as laptops, desktops, and servers, eliminating speculation has generally been regarded as impractical, on the assumption that these architectures depend on speculation to achieve adequate performance [226, 263, 383, 491, 72, 164, 369].

The thesis of this work is that eliminating speculation is a feasible approach to mitigating the transient execution vulnerabilities on large-scale server hardware. Firstly, for particular kinds of workloads [377] or for large-scale servers [414] it has been demonstrated that the performance benefits of speculation degrade to the extent that the feature either has minimal performance benefits or actively harms performance, indicating that eliminating speculation may be more feasible for the largest-scale servers than for medium-scale laptops or desktops. Secondly, selective speculation techniques make it possible to partially disable

speculation, reducing the risk of speculation while still keeping many of the performance benefits. The research questions we have sought to address are whether eliminating speculation really is as "clearly unacceptable" [491] as other authors have assumed. In concrete terms, this work has involved exploring the performance characteristics of speculation on server hardware, and experimenting with other ways to improve the performance of server hardware either without speculation or with restricted speculation. We hope that this work may help encourage server hardware vendors to consider the kind of fundamental hardware architecture changes the industry needs to effectively control the transient execution vulnerabilities.

The next two chapters provide necessary background for the work of the dissertation, with Chapter 2 providing historical perspective on the security context of server hardware and Chapter 3 providing a critical analysis of the transient execution vulnerabilities. Chapters 4 through 6 explore the feasibility of three alternative approaches to mitigating all variants of Spectre through fundamental microarchitecture design choices that block the initial fault-injection phase of the attack. Chapter 7 describes prototype implementation work completed during the course of this research.

Chapter 2

Background

Many modern computing workloads run in multitenant environments, where each physical machine is split into hundreds or thousands of smaller units of computing, generically called *guests*. Cloud and containers are currently the leading approaches to implementing multitenant infrastructures, but other related technologies, such as unikernels or serverless, are also variations on multitenant infrastructures. The guests in a cloud deployment are commonly called virtual machines or cloud instances, while the guests in a container deployment are commonly called containers. Typically, a single *tenant* (a user or group of users) is granted access to deploy guests in an orchestrated fashion across a cloud or cluster made up of thousands or hundreds of thousands of physical machines located in the same data center or across multiple data centers, to facilitate operational flexibility in areas such as capacity planning, resiliency, and reliable performance under variable load. Each guest runs its own (often minimal) operating system and application workloads, and maintains the illusion of being a physical machine, both to the end users who interact with the services running in the guests, and to developers who are able to build those services using familiar abstractions, such as programming languages, libraries, and operating system features. The illusion, however, is not perfect, because ultimately the guests do share the hardware resources (CPU, memory, cache, devices) of the underlying physical host machine, and consequently also have greater access to the host's privileged software (kernel, operating system) than a physically distinct machine would have.

Ideally, multitenant environments would offer strong isolation of the guest from the host, and between guests on the same host, but reality falls short of the ideal. The approaches that various implementations have taken to isolating guests have different strengths and weaknesses. For example, containers share a kernel with the host, while virtual machines may run as a process in the host operating system or a module in the host kernel, so they expose different attack surfaces through different code paths in the host operating system. Fundamentally, however, all existing implementations of virtual machines and containers are leaky abstractions, exposing more of the underlying software and hardware than is necessary, useful, or desirable. New security research starting in 2018 delivered a further blow to the ideal of isolation in multitenant environments, demonstrating that certain hardware vulnerabilities related to speculative execution—including Spectre, Meltdown, Foreshadow, L1TF, and variants—can easily bypass the software isolation of guests.

Because multitenancy has proven to be useful and profitable for a large sector of the computing industry, it is likely that a significant percentage of computing workloads will continue to run on multitenant infrastructure for the foreseeable future. Randal [351] examined the co-evolution of software and hardware for multitenant infrastructures over

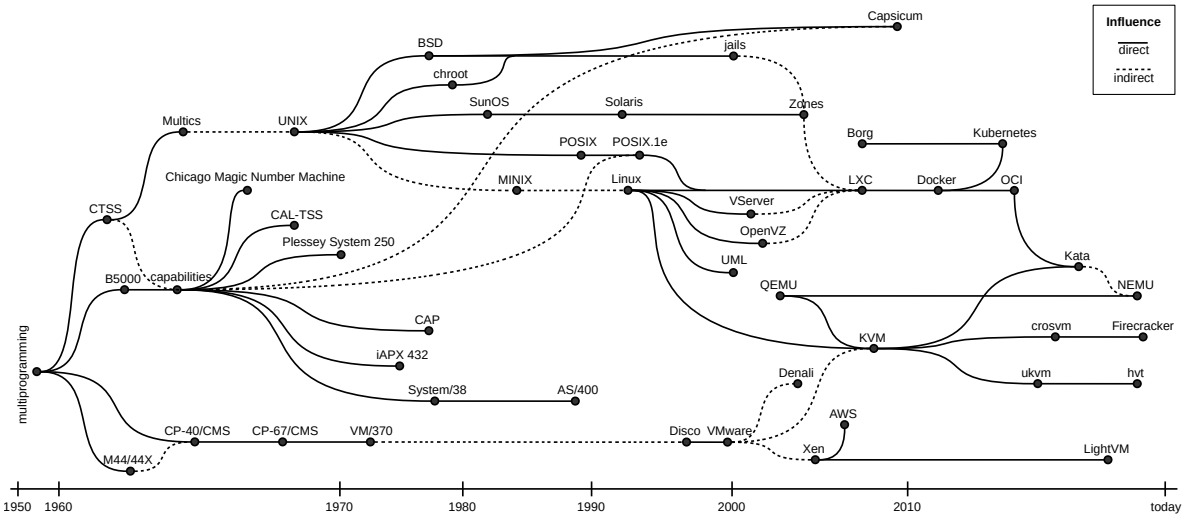

Figure 2.1: The evolution of multitenant infrastructures. Reprinted from Randal [351].

sixty years of history, and how the trade-offs made along the way led to the current tension between the lofty ideals of security versus the flawed reality. This dissertation focuses on the hardware dimension of multitenant infrastructures, and particularly on the impact of transient execution vulnerabilities.

This chapter is divided into sections following the evolutionary paths of the technologies behind virtual machines and containers, generally in chronological order, as illustrated in Figure 2.1. Section 2.2 explores the common origins of virtual machines and containers in the late 1950s and early 1960s, driven by the architectural shift toward multitasking and multiprocessing, and motivated by a desire to securely isolate processes, efficiently utilize shared resources, improve portability, and minimize complexity. Section 2.3 examines the first virtual machines in the mid-1960s to 1970s, which primarily aimed to improve resource utilization in time-sharing systems. Section 2.4 delves into the capability systems of the early 1960s to 1970s—the precursors of modern containers—which evolved along a parallel track to virtual machines, with similar motivations but different implementations. Section 2.6 outlines the resurgence of virtual machines in the late 1990s and 2000s. Section 2.7 traces the emergence of containers in the 2000s and 2010s. Section 3 starts to explore the impact of recent security research into transient execution vulnerabilities on both virtual machines and containers.

2.1 Terminology

For the sake of clarity, this dissertation consistently uses certain modern or common terms, even when discussing literature that used various other terms for the same concepts.

- **cloud**: Implementation approaches that adopt the label "cloud" are typically virtual machines with added orchestration features to enhance portability. Cloud implementations also tend to favor lighter-weight guest images, which enhances performance and reduces complexity, though cloud images are generally not quite as minimal as container images.

- **container**: The term "container" does not have a single origin, but some early relevant examples of use are Banga *et al.* [34] in 1999, Lottiaux and Morin [268] in

2001, Morin *et al.* [299] in 2002, and Price and Tucker [337] in 2004. Early literature on containers confusingly referred to them as a kind of virtualization [337, 402, 292, 209, 77, 92], or even called them virtual machines [402]. As containers grew more popular, the confusion shifted to virtual machines being called containers [60, 498]. This dissertation uses the term "container" for multitenant deployment techniques involving process isolation on a shared kernel (in contrast with *virtual machine*, as defined below). However, in practice the distinction between containers and virtual machines is more of a spectrum than a binary divide. Techniques common to one can be effectively applied to the other, such as using system call filtering with containers, or using seccomp sandboxing or user namespaces with virtual machines.

- **guest**: The term "guest" had some early usage in the 1980s for the operating system image running inside a virtual machine [303], but was not common until the early 2000s [448, 40]. This dissertation uses "guest" as a general term for operating system images hosted on multitenant infrastructures, but occasionally distinguishes between virtual machine guests and container guests.

- **kernel**: A variety of different terms appear in the early literature, including "supervisory program" [94], "supervisor program" [17], "control program" [306, 318, 7], "coordinating program" [318], "nucleus" [69, 98], "monitor" [472], and ultimately "kernel" around the mid-1970s [261, 334]. This dissertation uses the modern term "kernel".

- **process**: The early literature tended to use the terms "job" [361] or "program" [94, 318, 17], and "process" only appeared around the mid-1960s [112, 6]. This dissertation uses the modern term "process". The early use of "multiprogramming" meaning "multiprocessing" was derived from the early use of "program" meaning "process".

- **serverless**: Implementation approaches that adopt the label "serverless" tend to emphasize portability and minimizing complexity. They rely on the underlying infrastructure—typically some combination of bare metal, virtual machines, and/or containers—for whatever secure isolation and performance they provide.

- **unikernel**: Implementation approaches that adopt the label "unikernel" take minimalist guest images to an extreme, by replacing the kernel and operating system of the guest with a set of highly-optimized libraries that provide the same functionality. The code for an application workload is compiled together with the small subset of unikernel libraries required by the application, resulting in a very small binary that runs directly as a guest image. Historically, unikernels have sacrificed portability of guest images, by targeting only a limited set of virtual machine implementations as their host, but recent work has explored running unikernels as containers [474]. The unikernel approach also reduces the portability of application code, since unikernel frameworks tend to require the application code to be written in a specific way to integrate with the unikernel libraries.

- **virtual machine**: This dissertation uses the term "virtual machine" for multitenant deployment techniques involving the replication/emulation of real hardware architectures in software (in contrast with *container*, as defined above). The code responsible for managing virtual machine guests on a physical host machine is often

called a "hypervisor" or "virtual machine monitor", both derived from early terms for the kernel, "supervisor" and "monitor". In many early implementations of virtual machines, the host kernel managed both guests and ordinary processes.

2.2 Time-sharing on mainframes

The earliest form of hardware for multitenant infrastructures was time-sharing systems on mainframes. The origins of both virtual machines and containers can be traced to a fundamental shift in hardware and software architectures toward the late 1950s. The hardware of the time introduced the concept of *multiprogramming*, which included both basic multitasking in the form of simple context-switching and basic multiprocessing in the form of dedicated I/O processors and multiple CPUs. Codd [93] attributed the earliest known use of the term multiprogramming to Rochester [361] in 1955, describing the ability of an IBM 705 system to interrupt an I/O process (tape read), run a process (calculation) on the data found, and then return to the I/O process. The concept of multiprogramming evolved over the remainder of the decade through work on the EDSAC [471], UNIVAC LARC [122], STRETCH (IBM 7030) [120, 94], TX-2 [143], and an influential and comprehensive review by Gill [159]. While the hardware of today is radically different than the hardware of the 1950s-1970s, several key concepts continue to be relevant.

The first key concept was simply the ability to run more than one process on a machine at the same time. This concept was originally called *multiprogramming* and evolved through a series of hardware architectures in the 1950s, notably the IBM 705 [361], EDSAC [471], UNIVAC LARC [122], STRETCH (IBM 7030) [120, 94], and TX-2 [143]. Multiprogramming involved both multitasking (as simple context-switching) and multiprocessing (as multiple CPUs and dedicated I/O processors), which introduced a risk of processes disrupting the operation of other processes on the same machine.

So, the first key concept led naturally to the second key concept: isolating processes to prevent them disrupting each other. Initially, this work revolved around the now familiar approach of a small privileged kernel with unrestricted access to all hardware resources and running processes, as well as responsibility for potentially disruptive operations such as memory and storage allocation, process scheduling, and interrupt handling, combined with restrictions on any software outside the kernel to limit access to these risky features. STRETCH [94] in the 1950s and IBM System/360 [17] in the 1960s were significant early examples of hardware architectures designed to provide hardware support for kernel process isolation.

The concept of process isolation led to two major divergent schools of thought on hardware security for the multitenant systems of the time—*capabilities* and *virtual machines*—both initially focused on strengthening process isolation by adding memory isolation features. Capabilities viewed secure isolation as an essential feature of the hardware and operating system, which should be available to every process. Virtual machines approached secure isolation at a different level of granularity, emphasizing the ability to run an entire operating system in an isolated environment by closely replicating the behavior of physical hardware. Both capabilities and virtual machines depended heavily on custom hardware implementations of their security features. The fundamental principles of these two major divergent schools of thought continue today, as a dichotomy between modern containers and modern virtual machines.

2.3 Early virtual machines

The early work on virtual machines grew directly out of the work on multiprogramming, continuing the goal of safely sharing the resources of a physical machine across multiple processes. Initially, the idea was no more than a refinement on memory protection between processes, but it expanded into a much bigger idea: that small isolated bundles of shared resources from the host machine could present the illusion of being a physical machine running a full operating system.

2.3.1 M44/44X

In 1964, Nelson [306] published an internal research report at IBM outlining plans for an experimental machine based on the IBM 7044, called the M44. The project built on earlier work in multiprogramming, improving process isolation and scheduling in the privileged kernel with an early form of virtual memory. They called the memory mapped for a particular process a "virtual machine" [306, p. 14]. The 44X part of the name stood for the virtual machines (also based on the IBM 7044) running on top of the M44 host machine.

Nelson [306, p. 4-6] identified the performance advantages of dynamically allocated shared resources (especially memory and CPU) as one of the primary motivators for the M44/44X experiments. Portability was another central consideration, allowing software to run unmodified across single process, multiprocess, and debugging contexts [306, pp. 9-10].

The M44/44X lacked almost all of the features we would associate with virtual machines today, but it played an important, though largely forgotten, part in the history of virtual machines. Denning [111] reflected that the M44/44X was central to significant theoretical and experimental advances in memory research around paging, segmentation, and virtual memory in the 1960s.

2.3.2 Cambridge Monitor System

The IBM System/360 was explicitly designed for portability of software across different models and different hardware configurations [17]. In the mid-1960s, IBM's *Control Program-40 Cambridge Monitor System* (CP-40/CMS) project running on a modified IBM System/360 (model 40) took the idea a few steps further—initially calling the work a "pseudo-machine", but later adopting the term "virtual machine" [108, p. 485]. The CP-40/CMS and later CP-67/CMS[1] projects improved on earlier approaches to portability, making it possible for software written for a bare metal machine to run unmodified in a virtual machine, which could simulate the appearance of various different hardware configurations [7, pp. 1-2]. It also improved isolation by introducing privilege separation for interrupts [7, pp. 6-7], paged memory within virtual machine guests [69, 321], and simulated devices [69, 98]. IBM's work on the CP-40/CMS focused on improving performance through efficient utilization of shared memory [7, pp. 3-5], and explictly did not target efficient utilization of CPU through sharing [7, p. 1]. Kogut [229] developed a variant of CP-67/CMS to improve performance through dynamic allocation of storage (physical disk) to virtual machines.

[1] For the IBM System/360 model 67.

2.3.3 VM/370

IBM's VM/370 running on the System/370 hardware followed in the early 1970s, and included virtual memory hardware [108, p. 485]. Madnick and Donovan [273, p. 214] estimated the overhead of the VM/370 at 10-15%, but deemed the performance trade-off to be worthwhile from a security perspective. Goldberg [163, pp. 39-40] identified the source of overhead as primarily: maintaining state for virtual processors, trapping and emulating privileged instructions, and memory address translation for virtual machine guests (especially when paging was supported in the guests). In retrospect, Creasy noted that efficient execution was never a primary goal of IBM's work on the CP-40, CP-67, or VM/370 [108, p. 487], and the focus was instead on efficient utilization of available resources [108, p. 484].

2.3.4 Trade-offs

In their formal requirements for virtual machines in the mid-1970s, Popek and Goldberg [335, p. 413] stated that ideally virtual machines should "show at worst only minor decreases in speed" compared to running on bare metal. In 2017, Bugnion *et al.* [63] explained Popek and Goldberg's requirements in modern terms, exploring the performance impact for hardware architectures that do not fully meet the requirements.

Buzen and Gagliardi [69, p. 291], Madnick and Donovan [273, p. 212], Goldberg [162, p. 75], and Creasy [108, p. 486] all observed that the portability offered by virtual machines was also an advantage for development purposes, since it allowed development and testing of multiple different versions of the kernel/operating systems—and programs targeting those kernels/operating systems—in multiple different virtual hardware configurations, on the same physical machine at the same time.

Buzen and Gagliardi [69] considered one of the key advantages of the virtual machine approach to be that "virtual machine monitors typically do not require a large amount of code or a high degree of logical complexity". Popek and Kline [334, p. 294] discussed the advantage of virtual machines being smaller and less complex than a kernel and complete operating system, improving their potential to be secure. Goldberg [163, p. 39] suggested minimizing complexity as a way to improve performance: selectively disabling more expensive features (such as memory paging in guests) for virtual machines that would not use the features. Creasy [108, p. 488] discussed the advantages of minimizing interdependencies between virtual machines, giving preference to standard interfaces on the host machine.

A frequently-cited group of papers in the early 1970s, by Lauer and Snow [243], Lauer and Wyeth [244], and Srodawa and Bates [408], suggested that virtual machines offered a sufficient level of isolation that it was no longer necessary to maintain a privilege-separated kernel in the host operating system. However, by that point in time the concept of a privileged kernel was well enough established that the idea of eliminating it was unlikely to be widely accepted. Buzen and Gagliardi [69, p. 297] observed that the proposal depended heavily on the ability of the virtual machine implementation to handle all virtual memory mapping directly, but since the papers failed to take memory segmentation into account, the approach could not be implemented as initially proposed.

2.3.5 Decline

As companies like DEC, Honeywell, HP, Intel, and Xerox introduced smaller hardware to the market in the 1970s, they did not include hardware support for features such as virtual memory and the ability to trap all sensitive instructions, which made it challenging to implement strong isolation using virtual machine techniques on such hardware [117, 147]. Creasy [108, p. 484] observed in the early 1980s that the advent of the personal computer decreased interest in the early forms of virtual machines—which were largely developed for the purpose of isolating users in time-sharing systems on mainframes—but he recognized potential for virtual machines to serve "the future's network of personal computers".[2]

2.4 Early capabilities

The origin of containers is often attributed [97, 234, 258, 348, 46] to the addition of the `chroot` system call in the Seventh Edition of UNIX released by Bell Labs in 1979 [218]. The simple form of filesystem namespace isolation that `chroot` provides was certainly one influence on the development of containers, though it lacked any concept of isolation for process namespaces [210, 338]. However, containers are not a single technology, they are a collection of technologies combined to provide secure isolation, including namespaces, cgroups, seccomp, and capabilities. Combe *et al.* [97], Jian and Chen [204], Kovács [234], Priedhorsky and Randles [338], and Raho *et al.* [348] describe how these different technologies combine to provide secure isolation for containers. It is more accurate to attribute the origin of containers to the earliest of these technologies, capabilities, which began decades before `chroot` and several years before the first work on virtual machines. Like containers, capabilities took the approach of building secure isolation into the hardware and the operating system, without virtualization.

2.4.1 Descriptors

In the early 1960s, inspired by the need to isolate processes, the Burroughs B5000 hardware architecture introduced an improvement to memory protection called *descriptors*, which flagged whether a particular memory segment held code or data, and protected the system by ensuring it could only execute code (and not data), and could only access data appropriately (a single element scalar, or bounds-checked array) [283, 251]. A process on the B5000 could only access its own code and data segments through a private Program Reference Table, which held the descriptors for the process [251, p. 23]. A descriptor also flagged whether a segment was actively in main memory or needed to be loaded from drum [251, p. 24].

2.4.2 Dennis and Van Horn

In the mid-1960s, Dennis and Van Horn [112] introduced the term *capability* in theoretical work directly inspired by both the Burroughs B5000 and MIT's Compatible Time-Sharing System (CTSS) [112, p. 154]. Like the B5000 descriptors, capabilities defined the set of

[2] It was a reasonable prediction for the time: HTTP was introduced much later in the 1980s, but the RFC for the Internet Protocol (IP) [336] was published in the same month as Creasy's article, and TCP had already been around since the mid-1970s.

memory segments a process was permitted to read, write, or execute [251, p. 42]. These early capabilities introduced several important refinements: a process executed within a protected *domain* with an associated capability list; multiple processes could share the same capability list; and a process could `FORK` a parallel process with the same capabilities (but no greater), or create a subprocess with a subset of its own capabilities (but no greater) [251, pp. 42-44]. These theoretical capabilities also had a concept of ownership (by a process or a user) [251, p. 42], and of persistent data "directories" (but not files) which survived beyond the execution of a process and could be private to a user or accessible to any user [251, pp. 44-45].

Soon after Dennis and Van Horn published their theoretical capabilities, Ackerman and Plummer [6] implemented some aspects of capabilities relating to resource control on a modified PDP-1 at MIT, and added a file capability in addition to the directory capability—a precursor to filesystem namespaces.

2.4.3 Chicago Magic Number Machine

In 1967, the University of Chicago launched the first attempt at designing and building a general-purpose hardware and software capability system, which they later called the Chicago Magic Number Machine[3] [131, 132]. The Chicago machine pushed the concept of separation between capabilities and data further, to protect against users altering the capabilities that limited their access to memory on the system [251, pp. 49-50]. The machine had a set of physical registers for capabilities, which were distinct from the usual set of registers for data. It also flagged whether each memory segment stored capabilities or data, and prevented processes from performing data operations like reading or writing on capability segments or capability registers. Inter-process communication also sent both a capability segment and a data segment [251, p. 51].

The University of Chicago project ran out of funding and was never completed, but it inspired subsequent work on CAL-TSS [251, p. 49].

2.4.4 CAL-TSS

In 1968, the University of California at Berkeley launched the CAL-TSS project [251, pp. 52-57], which aimed to produce a general-purpose capability-based operating system, to run on a Control Data Corporation 6400 model (RISC architecture) mainframe machine, without any special customization to the hardware. Like previous implementations, CAL-TSS confined a process to a domain, restricting access to hardware registers, memory, executable code, system calls to the kernel, and inter-process communication. The project introduced a concept of unique and non-reusable identifiers for objects, to protect against reuse of dangling pointers to access and modify memory that has been reallocated after being freed.

The CAL-TSS project encountered difficulties implementing the operating system as designed, and was terminated in 1971. Levy [251, p. 57] identified the memory management features of the CDC 6400 as a particularly troublesome obstacle to the implementation. In postmortem analysis, Sturgis [411] and Lampson and Sturgis [241] reflected that CAL-TSS ended up being large, overly complex, and slow, and attributed this primarily to a poor

[3]The unusual name was emblematic of the decade, from Ken Kesey's "Magic Bus" to the Beatles' "Magical Mystery Tour". At the level of physical memory, capabilities are effectively a "magic" number.

match between the hardware they selected and the design of mapped address spaces, and also to their design choice of distributing privileged code for manipulating global system data across individual processes, rather than consolidating it in a privileged kernel.

2.4.5 Plessey System 250

In the early 1970s, the Plessey System 250 [124] was a commercially successful real-time multiprocessing telephone-switch controller. It implemented capabilities for memory protection and process isolation [251, p. 65], and expanded capabilities into the I/O system [251, p. 77].

2.4.6 Provably Secure Operating System

Also in the early 1970s, the Stanford Research Institute began a project to explore the potential of formal proofs applied to a capability-based operating system design, which they called the Provably Secure Operating System (PSOS) [308]. The design was completed in 1980, but never fully formally proven, and never implemented [309].

2.4.7 CAP

In the late 1970s, the University of Cambridge's CAP machine [305, 470] successfully implemented capabilities as general-purpose hardware combined with a complementary operating system. The CAP introduced a refinement replacing the privileged kernel with an ordinary process, so the special control the "root" process had over the entire system was really just the normal ability of any process to create subprocesses and grant a subset of its own capabilities to those subprocesses [251, pp. 80-81].

2.4.8 Object systems

Several software offshoots of the early capability systems generalized the idea by treating processes and shared resources as typed objects with associated capabilities, including Carnegie-Mellon's Hydra [486, 487], StarOS [208], and Gnosis later renamed to KeyKOS [181].

2.4.9 IBM System/38

In 1978, IBM announced plans for a capability-based hardware architecture, the System/38, which they shipped in 1980 [251, p. 137]. Berstis [48] characterized the primary goal of the System/38 as improving memory protection without sacrificing performance. Houdek [188] described the implementation of capabilities as protected pointers in detail. The System/38 introduced a concept of user profiles associated with protected process domains [48, pp. 249-250], which were vaguely reminiscent of modern user namespaces, though implemented differently. User profiles allowed for revocation of capabilities, but at the cost of significantly increased complexity in the implementation [251, pp. 155-156].

The System/38 was succeeded by the AS/400 in the late 1980s, which removed capability-based addressing [403, p. 119]. The AS/400 later adopted the concept of logical

partitioning from the IBM System/370 [378, pp. 1-2], to divide the physical resources of the host machine between multiple guests at the hardware level[4] [403, pp. 240, 328].

2.4.10 Intel iAPX 432

In 1975, Intel began designing the iAPX 432 [198] capability-based hardware architecture, which they originally intended to be their next-generation, market-leading CPU, replacing the 8080 [284, p. 79]. The project finally shipped in 1981, but it was significantly delayed and significantly over budget [284, p. 79].

Mazor [284, p. 75] recorded that performance was not considered as a goal in the design of the iAPX 432. Hansen *et al.* [180] measured the performance of the iAPX 432 against the Intel 8086, Motorola 68000, and the VAX-11/780 in 1982, with results as poor as 95 times slower on some benchmarks. Norton [310, p. 27] assessed the poor performance and unoptimized compiler offered by the iAPX 432 as the leading cause of its commercial failure. Levy [251, p. 186] blamed the commercial failure on both poor performance and over-hyped marketing.

In a move that Mazor described as "a crash program...to save Intel's market share" [284, p. 75], Intel launched a parallel project to develop the 8086 architecture (the first in a long line of x86 CPUs), which became Intel's leading product line by default, rather than by design [284, p. 79].[5]

2.4.11 Trade-offs

The early capability systems in the 1960s and 1970s sacrificed performance for the sake of security, though Levy speculated in the mid-1980s that this was partly due to "hardware poorly matched to the task" [251, p. 205]. Wilkes [472, pp. 49-59] contrasted the memory protection features of capabilities with other systems of the time, including detailed descriptions of hardware implementations.

Levy [251, p. 205] also observed that the early capability systems significantly increased complexity for the sake of security. Patterson and Séquin [326] and Patterson and Ditzel [324] judged this sacrifice as a major reason the capability machines were surpassed by simpler architectures, such as RISC.

Kirk McKusick recalled that the primary reason Bill Joy ported `chroot` from UNIX into BSD in 1982 was for portability, so he could build different versions of the system in an isolated build directory [210, p. 11].

2.4.12 Decline

As with virtual machines, interest in the early capability systems sharply declined in the 1980s, influenced by several independent factors. Several early attempts to implement capabilities were terminated uncompleted—notably the Chicago Magic Number Machine, CAL-TSS, and the Provably Secure Operating System—contributing to a reputation that capability systems were difficult to implement and perhaps overly ambitious, despite the

[4] Unlike virtual machines, capabilities, or containers, which divide physical resources at the software level.

[5] In hindsight, the commercial failure of the iAPX 432 probably influenced Intel's single-minded focus on performance and disinterest in memory protection techniques in the decades that followed, which ultimately contributed to the vulnerabilities discussed in Section 3.

successful implementations that followed. The commercial failure of Intel's iAPX 432 raised further doubts on the feasibility of capability-based architectures. In 2003, Neumann and Feiertag [309, p. 6] looked back on the early capability systems, expressing disappointment that "the demand for meaningfully secure systems has remained surprisingly small until recently".

Perhaps the most significant factor in the decline of capabilities was the rise of general-purpose operating systems. Saltzer and Schroeder [371, p. 1294] contrasted capabilities with the access control list models adopted by Multics and its descendants, calling out revocation of access as one major area where capabilities fell short.

While none of the early capability systems remain in use today, they have not been entirely forgotten. In 2003, Miller *et al.* [293] reviewed capability systems from a historical perspective, addressing common misconceptions about capabilities related to revocation, confinement, and equivalence to access control lists. Section 2.7 traces the evolution of a feature called capabilities in the modern Linux Kernel. FreeBSD took a different approach for the feature it calls capabilities, and integrated the Capsicum framework [289, p. 30], which was more directly derived from the classic capability systems [458, 21]. In 2012, the CHERI project [459, 461, 481, 457] expanded on the ideas of the Capsicum framework, pushing its capability model down into a RISC-based hardware architecture. Since 2016, Google has been exploring a revival of capability systems with the Fuchsia operating system and Zircon microkernel [166]. In a 2018 plenary session about Spectre/Meltdown, Hennessy [185] pointed to future potential for capabilities, reflecting that the early capability systems "probably weren't the right match for what software designers thought they needed and they were too inefficient at the time", but suggested "those are all things we know how to fix now...so it's time, I think, to begin re-examining some of those more sophisticated [protection] mechanisms and see if they'll work".

2.5 General-purpose hardware and general-purpose operating systems

As early as the 1960s, hardware vendors recognized that designing complete custom hardware, custom operating systems, and custom application software for each generation of their products was an expensive way to approach systems development. In 1964, Amdahl *et al.* [17] discussed the philosophy of "general-purpose CPU design for communications-oriented systems" as a driving design principle for the IBM System/360. The idea led to a third key concept that survives in modern multitenant infrastructures—portability made possible by architectural stratification and standardization.

The 1970s and 1980s saw the rise of general-purpose hardware capable of running multiple different operating systems, general-purpose operating systems capable of running on multiple different hardware architectures, and application/workload software capable of running on multiple different operating systems and hardware architectures. On the hardware side, companies like DEC, Honeywell, HP, Intel, and Xerox shifted their product lines toward simpler general-purpose hardware architectures that no longer supported the security features of capabilities [309, 284, 403] or virtual machines [117, 147]. On the operating system side, MIT's Compatible Time-Sharing System (CTSS) [101, 472] laid the foundation for Multics [100], which later inspired UNIX [359] and its robust mutation, the Berkeley Software Distribution (BSD) [288, 287], followed by Solaris, Linux, and their

many variants.

On one hand, stratification and standardization were a substantial benefit to the hardware and software industries, as both hardware and software architectures grew so much more complex over the decades, that the only sustainable approach to ongoing development of the full hardware and software stack was to break it into modular and recombinable hardware components—such as CPUs, memory, and storage—together with modular and recombinable software components—such as kernels, system utilities, operating systems, and applications. On the other hand, stratification and standardization were also a source of risk, as researchers and engineers working at one architectural level tended to have less and less exposure over time to how other levels actually functioned, across the boundaries of microarchitecture, instruction set architecture, peripherals, kernel and user space features, and application/workload software. This fundamental disconnect has played a part in the transient execution vulnerabilities. Modern software security research relies on critical assumptions about the behavior of the hardware that have been false for decades, but software security research is so far removed from modern microarchitecture research that few researchers saw the risk, and even those who did [330] radically underestimated the impact.

2.6 Modern virtual machines

Virtual machines still existed in the 1980s and 1990s, but garnered only a bare minimum of activity and interest. IBM's line of VM products, descended from VM/370, continued to have a small but loyal following [443]. DOS, OS/2, and Windows all offered a limited form of DOS virtual machines during that time, though it might be more fair to categorize those as emulation. The rise of programming languages like Smalltalk and Java re-purposing the term "virtual machine"—to refer to an abstraction layer of a language runtime, rather than a software replication of a real hardware architecture—may be indicative of how dead the original concept of virtual machines was in that period.

After nearly two decades, the late 1990s brought a resurgence of interest in virtual machines, but for a new purpose adapted to the technology of the time.

2.6.1 Disco

In 1997, the Disco research project at Stanford University explored reviving virtual machines as an approach to making efficient use of hardware with multiple CPUs (on the order of "tens to hundreds"), and included a lightweight library operating system for guests (SPLASHOS) as an option, in addition to supporting commodity operating systems as guests. Bugnion *et al.* [64] cited portability (rather than security or performance) as the primary motivation of the Disco project, which proposed virtual machines as a potential way to allow commodity operating systems (Unix, Windows NT, and Linux) to run on NUMA architectures without extensive modifications.

2.6.2 VMware

A year later, the team behind Disco founded VMware to continue their work, and released a workstation product in 1999 [65], quickly followed by two server products (GSX and ESX) in 2001 [448, 10, 373]. VMware faced a challenge in virtualizing the x86 architectures

of the time, because the hardware did not support traditional virtualization techniques—specifically the architecture contained some sensitive instructions which were not also privileged—so a virtual machine monitor could not rely on trapping protection exceptions as the sole means of identifying when to execute emulated instructions as a safe replacement, since some potentially harmful instructions would never be trapped [360, p.131].[6] To work around this limitation, VMware combined the trap-and-execute technique with a dynamic binary translation technique [65, p.12:3], which was faster than full emulation, but still allowed the guest operating system to run unmodified [65, p.12:29-36].

2.6.3 Denali

The Denali project at the University of Washington in 2002 [466] introduced the term "paravirtualization",[7] another work-around for the lack of hardware virtualization support in x86, which involved altering the instruction set in the virtualized hardware architecture, and then porting the guest operating system to run on the altered instruction set [465].

2.6.4 Xen

The Xen project at the University of Cambridge in 2003 [40] also used paravirtualization techniques and modified guest operating systems, but emphasized the importance of preserving the application binary interface (ABI) within the guests so that guest applications could run unmodified. Xen's greatest technical contribution may have been its approach to precise accounting for resource usage, with the explicit intention to individually bill tenants sharing physical machines [40, p.176], which was a relatively radical idea at the time,[8] and directly led to the creation of Amazon's Elastic Compute Cloud (EC2) a couple of years later [41].[9]

Chisnall [89] provided a detailed account of Xen's architecture and design goals. Xen's approach to the problem of untrapped x86 privileged instructions was to substitute a set of *hypercalls* for unsafe system calls [89, pp.10-13]. Smith and Nair [400, p.422] highlighted that Xen was able to run unmodified application binaries within the guest, because it ran the guest in ring 1 of the IA-32 privilege levels and the hypervisor in ring 0, so all privileged instructions were filtered through the hypervisor.

2.6.5 x86 Hardware virtualization extensions

In 2000, Robin and Irvine [360] analyzed the limitations of the x86 architecture as a host for virtual machine implementations, with reference to Goldberg's earlier work [161] on the architectural features required to support virtual machines. In the mid-2000s, in response to the growing success of virtual machines, and the challenges of implementing them on x86 hardware, Intel and AMD both added hardware support for virtualization in the form of a less privileged execution mode to execute code for the virtual machine guest directly, but selectively trap sensitive instructions, eliminating the need for binary translation or

[6] Popek and Goldberg [335] classically defined such machines as unvirtualizable.

[7] The term was new, but the technique had roots stretching back to IBM's VM/370 [108, 163].

[8] Partially inspired by earlier work, involving some of the same authors, on resource management in the Nemesis operating system [39].

[9] The EC2 beta was launched in 2006, but when I presented at the Amazon Developers Conference in 2005, they were already working on it.

paravirtualization. Rosenblum and Garfinkel [365] discussed the motivations behind the added hardware support for virtualization in x86, before the changes were released. Pearce *et al.* [328, p. 7] contrasted binary translation, paravirtualization, and the features x86 added for hardware-assisted virtualization, clarifying the x86 virtualization extensions were not full virtualization. Adams and Agesen [8] recounted the difficulties VMware encountered while integrating the x86 hardware virtualization extensions, and concluded that the new features offered no performance advantage over binary translation.

In 2007, the KVM subsystem for the Linux Kernel provided an API for accessing the x86 hardware virtualization extensions [224]. Since KVM was only a Kernel subsystem, the developers released a fork of QEMU[10] as the userspace counterpart of KVM, so the combination of QEMU+KVM provided a full virtual machine implementation, including virtual devices [456, pp.128-129]. Eventually, KVM support was merged into mainline QEMU [259].

2.6.6 Hyper-V

In 2008, Microsoft released a beta of Hyper-V [211] for Windows Server. It was built on top of the x86 hardware virtualization extensions, and for some virtual devices offered a choice between slower emulation and faster paravirtualization if the guest operating system installed the "Enlightened I/O" extensions. Like Xen's Dom0, Hyper-V granted special privileges to one guest, called the "parent partition", which hosted the virtual devices and handled requests from the other guests.

In 2010, Bolte *et al.* [56] incorporated support for Hyper-V into `libvirt`, so it could be managed through a standardized interface, together with Xen, QEMU+KVM, and VMware ESX.

2.6.7 Trade-offs

Denali and Xen both used paravirtualization techniques, sacrificing portability to gain performance, but their goals for scale were completely different: Denali considered 10,000 virtual machines[11] to be a good result [467]—achieved through a combination of lightweight guests and a minimal host—while Xen argued that 100 virtual machines running full operating systems[12] was a more reasonable target [40, p.165, 175]. To some extent, Denali was more in line with modern container implementations than with the virtual machine implementations of its day. Xen has shifted their estimation of required scale upward over the years, but still exhibits a tolerance for unnecessary performance degradation. For example, Manco *et al.* [277] demonstrated that a few small internal changes to the way Xen stores metadata and creates virtual devices improved virtual machine instantiation time by an order of magnitude—a result 50-200 times faster than Docker's container instantiation—however those patches are unlikely to ever make it into mainline Xen.

Xen and KVM have a reputation for sacrificing performance to gain security, however several independent lines of research have raised questions as to whether those security gains are real or imagined. Perez-Botero *et al.* [331] analyzed security vulnerabilities in Xen and KVM between 2008-2012, categorizing them by source, vector, and target,

[10]Which was previously only an emulator [44].
[11]On a 1.7GHz Pentium 4 with 1GB RAM.
[12]On a 2.4GHz dual-core Xeon with 2GB RAM.

and observed that the most common vector of attack was device emulation (Xen 34%, KVM 40%), the majority were triggered from within the virtual machine guest (Xen 71%, KVM 66%), and the majority successfully targeted the hypervisor's Ring -1 privileges or slightly less privileged control over Dom0 or the host operating system (Xen 80%, KVM 76%). Chandramouli *et al.* [83] built on the work of Perez-Botero *et al.* [331], moving toward a more general framework for forensic analysis of vulnerabilities in virtual machine implementations. Ishiguro and Kono [201] evaluated vulnerabilities in Xen and KVM related to instruction emulation between 2009-2017. They demonstrated that a prototype "instruction firewall" on KVM—which denies emulation of all instructions except the small subset deemed legitimate in the current execution context—could have defended against the known instruction emulation vulnerabilities, however the patches are unlikely to ever make it into mainline KVM.

Szefer *et al.* [418] demonstrated in the NoHype implementation (based on Xen) that eliminating the hypervisor and running virtual machines with more direct access to the hardware improved security by reducing the attack surface and removing virtual machine exit events as potential attack vectors. However, the approach involved a performance trade-off in resource utilization that was not viable for most real deployments: it pre-allocated processor cores, memory, and I/O devices dedicated to specific virtual machines, rather than allowing for oversubscription and dynamic allocation in response to load.

One persistent argument in favor of virtual machines has been that virtual machine implementations have fewer lines of code than a kernel or host operating system, and are therefore easier to code-review and secure [64, 150, 277, 328, 389], which is the classic trade-off of minimizing complexity to gain security. However, less code offers only a vague potential for security, and even that potential becomes questionable as modern virtual machine implementations have grown larger and more complex [96, 328, 475, 60].

Recent work on virtual machines—such as `ukvm` [473], LightVM [277], and Kata Containers (formerly Intel Clear Containers) [213]—has shifted back toward an emphasis on improving performance. However, this work appears to be founded on the assumption that the virtual machine implementations under discussion are adequately secure, and need only improve performance, which is a dubious assumption at best.

Two notable departures from this complacent attitude to security are Google's crosvm [165] and Amazon's Firecracker [16], which aim to improve both performance and security, by replacing QEMU with a radically smaller and simpler userspace component for KVM, and by choosing Rust as the implementation language for memory safety.[13] Firecracker started as a fork of crosvm, but the two projects are collaborating on generalizing the divergence into a set of Rust libraries they can share.

2.6.8 Decline

Toward the end of the 2000s, the enthusiasm for virtual machines gave way to a growing skepticism. Garfinkel *et al.* [149] demonstrated that virtual machine environments could reliably be detected on close inspection, reviving the long-running tension between the ideals of strong isolation in virtual machines, and the reality of actual implementations. Buzen and Gagliardi [69] commented on the ideals in the early 1970s, "Since a privileged

[13]The memory safety features of Rust do not address the security vulnerabilities discussed in Section 3, but can eliminate another common class of memory access vulnerabilities, such as buffer overflows/underflows and use-after-free. Szekeres *et al.* [419] provide a systematic account of such vulnerabilities and their impact in the C/C++ programming languages.

software nucleus has, in principle, no way of determining whether it is running on a virtual or a real machine, it has no way of spying on or altering any other virtual machine that may be coexisting with it in the same system." but in the same paper acknowledged, "In practice no virtual machine is completely equivalent to its real machine counterpart."

In 2010, Bratus *et al.* [60] criticized the disproportionate focus of systems security research on virtual machines and the resulting neglect of other potentially superior approaches to system security. Vasudevan *et al.* [445] outlined a set of requirements for protecting the integrity of virtual machines implemented on x86 with hardware virtualization support, and evaluated all existing implementations as "unsuitable for use with highly sensitive applications" [445, p.141]. Colp *et al.* [96] observed that multitenant environments presented new risks for virtual machine implementations, because they required stronger isolation between guests sharing the same host than was necessary when a single tenant owned the entire physical machine.

Virtual machines such as Xen, QEMU+KVM, Hyper-V, and VMware are still in active use today, but in recent years they have ceded their reputation as the leading technology for cloud deployments to containers.

2.7 Modern containers

The collection of technologies that make up modern container implementations started coming together years before anyone used the term "container". The two decade span surrounding the development of containers corresponded to a major shift in the way information about technological advances was broadcast and consumed. Exploring the socio-economic factors driving this shift is outside the scope of this survey, however, it is worth noting that the academic literature on more recent projects such as Docker and Kubernetes is largely written by outsiders providing external commentary, rather than by the primary developers of the technologies. As a result, recent academic publications on containers tend to lack the depth of perspective and insight that was common to earlier publications on virtual machines, capabilities, and security in the Linux Kernel. The dialog driving innovation and improvements to the technology has not disappeared, but it has moved away from the academic literature and into other communication channels.

2.7.1 POSIX capabilities

In the mid-1990s, the security working group of the POSIX standards project began drafting an extension to the POSIX.1 standard, called POSIX 1003.1e [340, 123, 173], which added a feature called "capabilities". The implementation details of POSIX capabilities were entirely different than the early capability systems [460, p.97], but had similarities on a conceptual level: POSIX capabilities were a set of flags associated with a process or file, which determined whether a process was permitted to perform certain actions; a process could `exec` a subprocess with a subset of its own capabilities; and the specification attempted to support the principle of least privilege [340]. However, the POSIX capabilities did not adopt the concepts of small access domains and no-privilege defaults, which were crucial elements of secure isolation in the early capability systems [110]. The POSIX.1e draft was withdrawn from the process in 1998 and never formally adopted as a standard [173, p.259], but it formed the basis of the capabilities feature added to the Linux Kernel in 1999 (release 2.2) [73, 278].

2.7.2 Namespaces and resource controls

A second important strand in the evolution of modern container implementations was the isolation of processes via namespaces and resource usage controls. In 2000, FreeBSD added Jails [210], which isolated filesystem namespaces (using `chroot`), but also isolated processes and network resources, in such a way that a process might be granted root privileges inside the jail, but blocked from performing operations that would affect anything outside the jail. In 2001, Linux VServer [402] patched the Linux Kernel to add resource usage limits and isolation for filesystems, network addresses, and memory. Around the same time, Virtuozzo (later released as OpenVZ) [193, 282] also patched the Linux Kernel to add resource usage limits and isolation for filesystems, processes, users, devices, and interprocess communication (IPC). In 2003, Nagar *et al.* [302] proposed a framework for resource usage control and metering called Class-based Kernel Resource Management (CKRM), and later released it as a set of patches to the Linux Kernel.

In 2002, the Linux Kernel (release 2.4.19) introduced a filesystem namespaces feature [219].[14] In 2006, Biederman [51] proposed expanding the idea of namespace isolation in the Linux Kernel beyond the filesystem to process IDs, IPC, the network stack, and user IDs. The Kernel developers accepted the idea, and the patches to implement the features landed in the Kernel between 2006 and 2013 (releases 2.6.19 to 3.8) [219]. The last set of patches to be completed was user namespaces, which allow an unprivileged user to create a namespace and grant a process full privileges for operations inside that namespace, while granting it no privileges for operations outside that namespace [438]. The way user namespaces are nested bears a resemblance to Dennis and Van Horn's [112] capabilities, where processes created more restricted subprocesses.

In 2004, Solaris added Zones [337] (sometimes also called Solaris Containers), which isolated processes into groups that could only observe or signal other processes in the same group, associated each zone with an isolated filesystem namespace, and set limits for shared resource consumption (initially only CPU). Between 2006 and 2007, Rohit Seth and Paul Menage worked on a patch for the Linux Kernel for a feature they called "process containers" [105]—later renamed to *cgroups* for "control groups"—which provided resource limiting, prioritization, accounting,[15] and control features for processes.

2.7.3 Access control and system call filtering

A third set of relevant features in the Linux Kernel evolved around secure isolation of processes through restricted access to system calls. In 2000, Cowan *et al.* [107] released SubDomain, a Linux Kernel module which added access control checks to a limited set of system calls related to executing processes. In 2001, Loscocco and Smalley [267] published an architectural description of SELinux, which implemented mandatory access control (MAC) for the Linux Kernel. The access control architecture of SELinux was received positively, but the implementation was rejected for being too tightly coupled with the kernel. So, in 2002, Wright *et al.* [482] proposed the Linux Security Modules (LSM) framework as a more general approach to extensible security in the Linux Kernel, which made it possible for security policies to be loaded as Kernel modules. LSM is not an access control mechanism, but it provides a set of hooks where other security extensions such as SELinux or AppArmor can insert access control checks. LSM and a modified

[14] Partially inspired by the namespaces feature of Plan 9 [333] from Bell Labs.

[15] Similar in idea, though not in implementation, to Xen's resource usage accounting.

version of SELinux based on LSM were both merged into the mainline Linux Kernel in 2003. In 2004-2005, SubDomain was rewritten to use LSM, and rebranded under the name AppArmor.

In 2005, Andrea Arcangeli [26] released a set of patches to the Linux Kernel called *seccomp* for "secure computing", which restricted a process so that it could only run an extremely limited set of system calls to exit/return or interact with already open filehandles, and terminated a process attempting to run any other system calls. The patches were merged into the mainline Kernel later that year. However, the features of the original seccomp were inadequate and rarely used, and over the years multiple proposals to improve seccomp were unsuccessful. Then, in 2012, Will Drewry [119] extended seccomp to allow filters for system calls to be dynamically defined using Berkeley Packet Filter (BPF) rules, which provided enough flexibility to make seccomp useful as an isolation technique. In 2013, Krude and Meyer [235] implemented a framework for isolating untrusted workloads on multitenant infrastructures using seccomp system call filter policies written in BPF.

2.7.4 Cluster management

A fourth relevant strand of technology evolved around resource sharing in large-scale cluster management. In 2001, Lottiaux and Morin [268] used the term "container" for a form of shared, distributed memory which provided the illusion that multiple nodes in an SMP cluster were sharing kernel resources, including memory, disk, and network. In 2002, the Zap project [319] used the term "pod"[16] for a group of processes sharing a private namespace, which had an isolated view of system resources such as process identifiers and network addresses. These pods were self-contained, so they could be migrated as a unit between physical machines. In the mid-2000s, Google deployed a cluster management solution called Borg [446, 68] into production, to orchestrate the deployment of their vast suite of web applications and services. While the code for Borg has never been seen outside Google, it was the direct inspiration for the Kubernetes project a decade later [446, p.18:13-14]—the Borg *alloc* became the Kubernetes *pod*, Borglets became Kubelets, and tasks gave way to containers. Burns *et al.* [68, p.70] explained that improving performance through resource utilization was one of the primary motivations for Borg.

2.7.5 Combined features

The strength of modern containers is not in any one feature, but in the combination of multiple features for resource control and isolation. In 2008, Linux Containers (LXC) [260] combined cgroups, namespaces, and capabilities from the Linux Kernel into a tool for building and launching low-level system containers. Miller and Chen [292] demonstrated that filesystem isolation between LXC containers could be improved by applying SELinux policies. Xavier *et al.* [488] and Raho *et al.* [348] contrasted LXC's approach to isolation and resource control using standard Linux Kernel features such as cgroups and filesystem, process, IPC, and network namespaces, versus the approaches taken by Linux VServer and OpenVZ using custom patches to the Linux Kernel to provide similar features.[17]

[16] Given as an acronym for a **Pr**ocess **D**omain abstraction.

[17] In the 2000s, many VM or container implementations relied on custom patches to the Linux Kernel, including VServer, OpenVZ, Xen, VMware, and MetaCluster (an earlier version of LXC). The practice was contentious, as multiple incompatible patch sets competed to be merged upstream [103], and ultimately none were ever accepted.

Docker [290] launched in 2013 as a container management platform built on LXC. In 2014, Docker replaced LXC with `libcontainer`, its own implementation for creating containers, which also used Linux Kernel namespaces, cgroups, and capabilities [196, 348]. Morabito *et al.* [298] compared the performance of LXC and Docker after the transition to libcontainer, and found them to be roughly equivalent on CPU performance, disk I/O, and network I/O, however LXC performed 30% better on random writes, which may have been related to Docker's use of a union file system. Raho *et al.* [348] contrasted the implementations of Docker, QEMU+KVM, and Xen on the ARM hardware architecture. Mattetti *et al.* [281] experimented with dynamically generating AppArmor rules for Docker containers based on the application workload they contained. Catuogno and Galdi [77] performed a case study of Docker using two different models for security assessment. They built on the work of Reshetova *et al.* [356] in classifying vulnerabilities by the goal of the attack: denial of service, container compromise, or privilege escalation.

In 2015, Docker split the container runtime out into a separate project, `runc`, in support of a vendor-neutral container runtime specification maintained by the Open Container Initiative (OCI). Hykes [197] highlighted that SELinux, AppArmor, and seccomp were all standard supported features in `runc`. Koller and Williams [231] observed that `runc` was more minimal than the Docker runtime, while still using the same isolation mechanisms from the Linux Kernel, such as namespaces and cgroups. In 2016, Docker and CoreOS merged their container image formats into a vendor-neutral container image format specification, also at OCI [57].

2.7.6 Orchestration

In 2014, Docker began working on Swarm, described as a clustering system for Docker, which they ultimately released late in 2015 [271]. Also in 2014, Google began developing Kubernetes, an orchestration tool for deploying and managing the lifecycle of containers, which they released in the middle of 2015 [61]. Also in 2014, Canonical began developing LXD, a container orchestration tool for LXC containers, which they released in 2016 [168].

Verma *et al.* [446] outlined the design goals behind Kubernetes, in the context of lessons learned from Borg. Syed and Fernandez [416, 415] pointed out that the performance advantages of the higher-level container orchestration tools, such as Kubernetes and Docker Swarm, were primarily a matter of improving resource utilization. They also contrasted the portability advantages of managing containers across multiple physical host machines against the increased complexity required for the orchestration tools to advance beyond managing a single machine host. Souppaya *et al.* [404] systematically reviewed increased security risks and mitigation techniques for container orchestration tools. Bila *et al.* [52] extended Kubernetes with a vulnerability scanning service and network quarantine for containers.

2.7.7 Trade-offs

Containers have a reputation for substantially better performance than virtual machines, however that reputation may not be deserved. In 2015, Felter *et al.* [136] measured the performance of Docker against QEMU+KVM and determined that neither had significant overhead on CPU and memory usage, but that KVM had a 40% higher overhead in I/O. They observed that the overhead was primarily due to extra cycles on each I/O operation, so the impact could be mitigated for some applications by batching multiple

small I/O operations into fewer large I/O operations. In 2017, Kovács [234] compared CPU execution time and network throughput between Docker, LXC, Singularity, KVM, and bare metal and determined that there was no significant variation between them, as long as Docker and LXC were running in host networking mode, but in Linux bridge mode Docker and LXC exhibited high retransmission rates that negatively impacted their throughput compared to the others. Manco *et al.* [277] demonstrated that Xen virtual machine instantiation could be 50-200 times faster than Docker container instantiation, with a few low-level modifications to Xen's control stack.

Secure isolation technologies have been the core of modern container implementations from the beginning, so it would be reasonable to expect that containers would provide a strong form of isolation. However, early implementations of containers were prone to preventable security vulnerabilities, which may indicate that security was not a primary design consideration, at least not initially. Combe *et al.* [97] analyzed security vulnerabilities in Docker and `libcontainer` between 2014-2015, and determined that the majority were related to filesystem isolation, which led to privilege escalation when Docker was run as the root user. They also suggested that some of Docker's sane default configurations for the isolation features of the Linux Kernel could be easily switched to less secure configurations through standard options to the `docker` command-line tool or the Docker daemon, and so might be prone to user error. Martin *et al.* [279] surveyed vulnerabilities in Docker images, `libcontainer`, the Docker daemon, and orchestration tools, as well as the unique security challenges of containers in multitenant infrastructures. In addition to security patches for specific privilege escalation vulnerabilities, there has been ongoing work to integrate support for user namespaces into Docker and Kubernetes,[18] so they can run as a non-root user and limit the scope of damage from privilege escalation. However, the user namespaces feature itself has had a series of vulnerabilities[19] related to interfaces in the Kernel that were written with the expectation of being restricted to the root user, but are now exposed to unprivileged users.

One significant difference between virtual machine implementations and container implementations is that containers share a kernel with the host operating system, so efforts to secure the kernel greatly impact the security of containers. Reshetova *et al.* [356] considered the set of secure isolation features offered by the Linux Kernel as of 2014 (in the context of LXC), and judged them to have caught up with the features of FreeBSD Jails and Solaris Zones, but highlighted some areas for improvement in support of containers. These improvements included integrating Mandatory Access Control (MAC) into the Kernel as "security namespaces"; providing a way to lock down device hotplug features for containers; and extending cgroups to support all resource management features supported by rlimits. Gao *et al.* [148] discussed the risks of certain types of information that containers can currently access from the Linux Kernel via *procfs* and *sysfs*—which can be exploited to detect co-resident containers and precisely target power consumption spikes to overload servers—and prototyped a power-based namespace to partition the information for containers.

Some more recent approaches to secure isolation for containers have been inspired by virtual machine implementations. Kata Containers (formerly Intel Clear Containers) [213] wraps each Docker container or Kubernetes pod in a QEMU+KVM virtual machine [214]. They realized that QEMU was not ideal for the purpose—since it introduces a substantial

[18]Such as Suda and Scrivano [413] and Suda [412].

[19]Such as CVE-2018-6559, CVE-2018-18955, CVE-2014-9717, and CVE-2014-4014.

performance hit compared to running bare containers, and the majority of the code relates to emulation which is not useful for wrapping containers—so a group at Intel started working on a stripped-down version of QEMU called NEMU [307]. X-Containers [391] used Xen's paravirtualization features to improve isolation between containers and the host, but made an unfortunate trade-off of removing isolation between containers running on the same host. Nabla Containers [301] and gVisor [167] have both taken an approach of improving isolation by heavily filtering system calls from containers to the host kernel, which is a common technique for modern virtual machines.

Bratus *et al.* [60] noted that the "self-protection" techniques employed by container implementations are a necessary path for future research, since even virtual machines depend on those techniques to protect themselves. Hosseinzadeh *et al.* [187] explored the possibility that container implementations might directly adapt earlier work (primarily Berger *et al.* [45]) for virtual machine implementations to integrate a Trusted Platform Module (TPM) as a virtual device.

Container implementations have a potential advantage over virtual machine implementations in addressing the problem of secure isolation over the long-term, not because any existing implementations are inherently superior, but because containers take a modular approach to implementation that permits them to be more flexible over time and across different underlying software[20] and hardware architectures, as new ideas for secure isolation evolve.

[20]Such as `pledge` and `unveil` on OpenBSD versus capabilities and namespaces on Linux.

Chapter 3

Transient execution vulnerabilities

Early in 2018, two papers by Kocher *et al.* [226] and Lipp *et al.* [263] drew attention to a new class of security vulnerabilities related to both speculative execution and out-of-order execution, collectively described as *transient execution*. The specific vulnerabilities they described—Spectre and Meltdown—use transient execution effects to amplify the severity and ease of exploiting previously known microarchitectural side-channel attacks, however subsequent work has demonstrated that transient execution effects can also be used to amplify the effects of other attacks, such as microarchitectural fault-injection attacks like Rowhammer. The broad class of transient execution vulnerabilities upend traditional notions of secure isolation, and radically expand the potential scope and severity of software-induced hardware vulnerabilities.

The features that the transient execution vulnerabilities exploit are common to modern major hardware architectures, such as x86 and ARM, and had already begun to be replicated in RISC-V implementations before the vulnerabilities were reported. While transient execution vulnerabilities may be found on desktop, mobile, embedded, or server hardware, the deployment contexts of virtual machines and containers on server hardware have unique implications on the ease of exploit, scope of impact, and choice of mitigations for the vulnerabilities, as we will discuss in Chapters 4 through 6. It has been argued that these vulnerabilities are not bugs in the traditional sense, because the transient execution features are functioning as they were designed, however they are flaws in the microarchitecture implementations of both speculative execution and out-of-order pipelines as optimizations to improve instruction-level parallelism. Today, it is possible to mitigate Meltdown-type vulnerabilities in the microarchitecture design with reasonably low performance penalties. Among the major server hardware vendors, AMD was never vulnerable to the initial variants of Meltdown [405, 401], and so far it appears that only ARM has made the effort to formally prove that certain generations of their hardware are not vulnerable to Meltdown [272]. Spectre-type vulnerabilities have proven to be more difficult to mitigate, and the products currently shipped by server hardware vendors and actively deployed around the world offer no more than meager protections—limiting some of the damage caused by some variants, while introducing prohibitive performance penalties—and do not resolve the inherent logic flaws of the microarchitecture implementations, which are the true root cause of the entire class of vulnerabilities.

We can never know what might have happened if the security trade-offs of transient execution had been fully considered at the same time the performance advantages were discovered—whether the transient execution vulnerabilities might have been exposed and resolved earlier, or whether modern computer microarchitectures might have evolved down

a slightly different path. If the history of virtual machines and containers has taught us anything, it is that we have the ability and responsibility to re-consider security trade-offs over time, and make better choices for the future. While it may not be fair to judge past work by lessons we learned later, it will be fair to judge future work on whether it applies those lessons or ignores them.

3.1 Precursors to Spectre and Meltdown

While Spectre, Meltdown, and more broadly the entire concept of software-induced transient execution vulnerabilities are relatively new in the field of security research, in essence they are no more than a small step of evolution beyond 70 years of hardware security research on covert channels, side-channel attacks, and fault-injection attacks.

3.1.1 Covert channels and side channels

In 1973, Lampson published "A note on the confinement problem" [240], an early but influential work on the challenges of preventing information leakage between isolated processes running on the same kernel. In that work he defined a *covert channel* as a hardware resource used to bypass isolation mechanisms by transferring information, where the attack succeeds because the hardware resource was never intended or recognized as a communication channel by the system's designers, so they never bothered to protect it against undesirable information leaks.

Later work uses the term *side channel* in combination with covert channel, but it is important to recognize that although the two terms sometimes appear to be used interchangeably in the literature—and the two kinds of attacks use some of the same hardware resources as channels—covert channels and side channels are not the same thing. In a covert-channel attack, the communication of leaked information is intentional, and the sender and receiver are both malicious (sometimes called "trojan" and "spy"). In a side-channel attack, the communication of leaked information is unintentional, and the sender is a victim, while the receiver is a malicious attacker [153, 417, 341].

The hardware resources that Lampson [240] envisioned being used as covert channels were no more complex than shared memory, a file on the file system, interprocess communication, or request/response metadata, but subsequent work over the decades has explored increasingly exotic channels for leaking information. Conceptually, modern side-channel attacks can trace their roots back to acoustic attacks in the mid-1950s, when recordings of the clicking sounds made by mechanical cryptographic machines captured enough information for attackers to break the cipher used in the encryption [50, 146].[1] However, there is a world of difference between the 1950s and today in the sophistication of the machines being attacked, the sources of information targeted, the quality and quantity of information being gathered from those sources, and the elaborate nature of analysis techniques applied to extract secrets from that information.

[1] Peter Wright of MI5 [483, pp. 81-86] described the attack—later codenamed ENGULF—in Chapter 7 of his autobiography. In 1956, with the help of the London Post Office, he bugged a telephone at the Egyptian Embassy in London installed next to their Hagelin cipher machine, with a hard line to GCHQ so they could listen in each morning as the cipher clerk entered the mechanical encryption settings for the day. Analyzing the recorded sounds with an oscilloscope yielded enough information about how the machine was configured each day that they were able to crack the cipher.

3.1.2 Physical side-channel attacks

The first rounds of research into side channels focused on physical side-channel attacks, exploiting indirect physical information to extract secrets. Because physical side-channel attacks require physical access or proximity to the machine, they are more difficult to perform, and have historically been regarded as less risky and only worth mitigating on security-critical components such as cryptographic hardware. The most common kinds of physical information gathered in these attacks, which still remain relevant today, are:

- Timing Analysis: measures execution time of operations (such as encryption/decryption) for different inputs, and infers secret information from variations in timing. This technique is often combined with other physical side-channel attacks. In the mid-1990s, Kocher [227] advanced this technique—and the entire research field of physical side-channel attacks—to a point of being able to extract entire secret keys from a decryption process.

- Power Analysis: measures power usage related to operations (such as encryption/decryption) for different inputs/outputs, and infers secret information from variations in power consumption. In the late 1990s, Kocher *et al.* [225] made similar advances in physical side-channel attack techniques making use of power analysis.

- Electromagnetic Analysis: measures electromagnetic waves produced by current flow over the device, and infers secret information from variations in electromagnetic signals. In the early 2000s, Quisquater and Samyde [344] built on Kocher's earlier work on timing analysis and power analysis to extract secret keys from smart cards using only electromagnetic analysis.

- Fault Analysis: physically tampers with voltage levels, clock signal, or other hardware components to trigger a fault in the device (e.g. disturb a few memory or register bits), and infers secret information based on variations in the output of faulty operations. This is actually a combination of two techniques, it starts with a physical fault-injection attack (violating integrity), then uses the successful results of the fault-injection attack as a source of information for a physical side-channel attack (violating confidentiality). In the mid-1990s, Anderson and Kuhn [22] made a first brief mention of clock and power glitching techniques in the context of smart card attacks, which Skorobogatov and Anderson [397] later explicitly connected with Kocher's work on physical side-channel attacks.

3.1.3 Microarchitectural side-channel attacks

More recent rounds of research into side-channel attacks have expanded the range of information sources considered. In contrast to physical side-channel attacks, microarchitectural side-channel attacks exploit indirect microarchitectural sources of information to extract secrets, do not require physical access to the machine, and may even be software-induced, so they are easier to perform and of greater concern for general-purpose hardware. The analysis techniques and objectives of microarchitectural side-channel attacks are similar to earlier work on physical side-channel attacks, though the sources of information used are more varied and also inspired by that earlier work.

As a common example of microarchitectural side-channel attack techniques, cache-timing analysis measures the time required to load a data value from cache, and infers secret

information from variations in timing. An attacker establishes a pre-defined cache state, allows the victim to perform an operation, then observes cache state changes. Although Kocher [227] briefly mentioned the influence cache-timing effects have on physical timing analysis in the mid-1990s, the idea of microarchitectural cache-timing side-channel attacks was not fully developed until the mid-2000s by Bernstein [47] and Percival [330], who extracted entire secret keys using only cache-timing information. Cache-timing side-channel attacks have been a prolific area of security research for nearly two decades, with variants differentiated by characteristics like the specific cache targeted (for an L1 attack to succeed, the attacker and victim have to share a core, while an LLC attack can succeed across cores), or the specific attacker actions to prepare or observe the cache, such as Prime+Probe [320, 266], Evict+Time [320], Flush+Reload [493], Flush+Flush [175], Stream+Reload [462], or Write+Write [424].

Caches are not the only targets for microarchitectural side-channel attacks, many other microarchitectural sources of information have been successfully exploited to extract secrets, such as:

- Translation Lookaside Buffer (TLB): Wang *et al.* [453], TLBleed [170] successfully bypasses cache isolation
- Page tables: Van Bulck [440], Wang *et al.* [453]
- DRAM: Pessl [332], Wang *et al.* [453]
- Prefetchers: Szefer [417], Shin *et al.* [393], Vicarte *et al.* [372], AfterImage [88]
- Branch Target Buffer (BTB): Branch Prediction Analysis (BPA) [4] and Simple Branch Prediction Analysis (SBPA) [3], Evtyushkin *et al.* [128], Lee *et al.* [249], Yu *et al.* [495]
- Conditional branch predictor, Pattern History Table (PHT): BranchScope [129], Bluethunder [194]
- Return Stack Buffer (RSB): Hyper-Channel [67]
- FPU timing: Andrysco *et al.* [23]
- SMT port contention: Wang and Lee [455], Aciicmez and Seifert [5], Aldaya *et al.* [15]
- GPU timing: Xu *et al.* [490]
- CPU frequency: CLKscrew [420]
- Power analysis: Hertzbleed [454], Platypus [265], Barenghi and Pelosi [38], Collide+Power [228]
- Memory controller scheduler: Semal *et al.* [388]
- Cache way predictor: Take A Way [262]
- Instruction cache: Aciiçmez [1], Aciiçmez *et al.* [2]
- Micro-op cache: Ren *et al.* [355]

- Performance counters: PMU-Leaker [342]
- MCU bus interconnect: BUSted [362]

Spectre and Meltdown build on this history of research into side-channel attacks. They make use of microarchitectural side-channel attack techniques, but are often falsely categorized simply as timing analysis techniques, specifically as cache-timing side-channel attacks [71]. It is more accurate to recognize that Spectre and Meltdown are both primarily fault analysis techniques, because they both begin with a fault-injection attack (violating integrity)—Meltdown by triggering an exception, and Spectre by inserting false entries into branch prediction and other prediction-related microarchitectural state—and then go on to use the successful results of the microarchitectural fault-injection attack as a source of information for a microarchitectural side-channel attack (violating confidentiality).

3.2 Transient execution

The concepts of transient execution, transient instructions, and transient microarchitectural state are modern terminology to describe some curious side-effects of the way general-purpose high-performance processors have been designed since the 1960s. The main features of concern for transient execution are speculative and out-of-order execution, but transient execution effects are compounded by the interactions between several features, including multilevel memory caches, simultaneous multithreading, multiple instruction issue, and prefetching.

In the late 1960s, Tomasulo [431] discussed an approach to dynamically scheduling the execution of instructions across multiple execution units, as implemented for floating-point operations in the IBM 360/91. The key insight of the approach was that instructions could be reordered from the original program sequence, as long as dependencies between instructions were preserved. One usability problem with this early implementation of out-of-order execution was that it delivered interrupts chaotically out of order too, because it had no concept of a separate in-order commit stage, and simply committed instructions as soon as they finished executing [325]. So, later implementations of out-of-order execution delayed interrupts and exceptions until an in-order commit stage, so they would be delivered in program order.

A collection of papers in the early 1970s, including Tjaden and Flynn [427], Flynn [140], Flynn and Podvin [141], and Riseman and Foster [358], explored the logical limits of instruction-level parallelism for the hardware of the time, identifying branches and memory loads as significant obstacles. Within a decade, the tone of publications shifted from assessing these obstacles as insurmountable, to assessing them as straightforwardly solved by combining several techniques that remain in common use today, particularly the speculative techniques of branch prediction and memory load prediction. Other techniques for instruction-level parallelism developed around the same time are commonly combined with speculation today, but are not inherently speculative—such as multiple instruction issue, register renaming, dynamic pipeline scheduling, and out-of-order execution.

Lee and Smith [247] and McFarling and Hennessy [285] captured historical perspectives on branch prediction from the point of view of the mid-1980s. Both surveyed the state of the art in branch prediction techniques at the time—such as dynamic prediction and branch target buffers—and critically reviewed previous techniques to speed up conditional branches without speculation—such as delayed branches, look-ahead resolution, branch

target prefetching, and multiple instruction streams. Smith [399, 398] captured similar early perspectives on hardware prefetching in the late 1970s, surveying the impact of memory access latency for both instruction fetching and memory load instructions, the limitations of existing implementations of instruction and data prefetching at the time, and the potential for future performance improvements.

One noteworthy characteristic shared by these papers—and by much of the substantial work on speculative and out-of-order pipeline techniques in the decades that followed—was a focus on metrics of performance with little or no consideration given to metrics of security. In all fairness to the hardware designers of the time, the groundbreaking work on speculation and out-of-order execution was completed decades before microarchitectural side-channel attacks were considered as a possibility. So, their failure was not a matter of willfully ignoring known threats, it was a naive complacency and unsophisticated design methodology that embraced new features without adequate consideration of the system-wide implications. Modern hardware designers have no such excuse. Some of the earliest work on microarchitectural side-channel attacks in the mid-2000s by Percival [330] explored the risks inherent in combining speculative execution with simultaneous multithreading, dynamic pipeline scheduling, multilevel memory caches, and hardware prefetching—identifying the essential constituents of the transient execution vulnerabilities over a decade before the full extent of their security impact was revealed. Fogh [142] also identified the potential risk that speculative and out-of-order execution could be used to amplify microarchitectural side-channel attacks in 2017, but did not formulate a successful attack.

3.2.1 Speculative branch instructions

Consider a superscalar microarchitecture that is roughly analogous to a modern x86 processor.[2] Like a modern x86, the pipelining approach in this hypothetical microarchitecture uses both dynamic multiple issue and dynamic pipeline scheduling, with out-of-order execution. Figure 3.1 illustrates the essential stages of such a microarchitecture: instructions are fetched, placed in a reorder buffer, queued to reservation stations, dispatched to functional units[3] for execution, and then the commit unit retires instructions, either writing or discarding their results. This illustration abstracts away some details of specific microarchitectures to focus on common elements. For example, microarchitectures for the complex x86 instruction set typically decode the fetched instructions into micro-operations, to simplify the issue, execution, and commit stages, but this extra decoding step is rarely added to microarchitectures for the RISC-V ISA,[4] which is fundamentally designed to be more like the simple micro-operations of x86.

First, consider how speculative branch instructions flow through these essential components of a dynamically scheduled pipeline.

[2]For a specific example, chapter 4, section 11 of Patterson and Hennessy [325] offers a straightforward overview of an x86 microarchitecture, based on an Intel Core i7 920.

[3]Superscalar processors have multiples of each kind of functional unit, including arithmetic logic units (ALU), floating-point units (FPU), load-store units (LSU), and more.

[4]The BOOM [501] RISC-V core does decode some instructions into micro-operations, as an optimization for a special case of data-dependent branches.

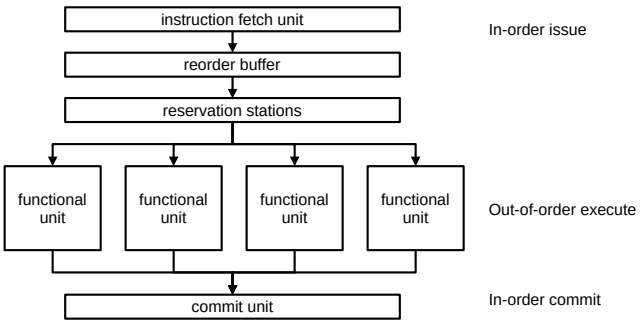

Figure 3.1: Essential components of a dynamically scheduled pipeline. Adapted from Patterson and Hennessy [325], Figure 4.69, p. 330.

3.2.1.1 Fetch

Based on a recent history of executed branches, instruction fetching uses a branch target buffer (BTB) to predict the destination that is most likely for each direct or indirect branch, or a pattern history table (PHT) to predict the whether a conditional branch is likely to be taken or not. Various microarchitectures may also use a branch history buffer (BHB), branch history table (BHT), or a return stack buffer (RSB) in predicting return addresses. Rather than waiting for the result of calculating the branch or return destination or evaluating the branch instruction condition, the instruction fetch unit will continue to fetch instructions along the predicted branch path.

3.2.1.2 Issue

Instruction issue places an entry for each instruction into a reorder buffer, in the order the instructions were fetched. This stage also performs register renaming, mapping the architectural registers (the ones visible in the ISA) onto a larger set of physical registers. Register renaming makes it possible for a sequence of speculatively executed instructions (or unrolled loop instructions) to effectively operate on a temporary virtual register set, which may be discarded if the speculatively predicted branch is later determined to be incorrect. Finally, instruction issue sends instructions to the reservation stations, either copying the operands immediately (if they are available) or copying them later (if the operands depend on the results of other instructions).

3.2.1.3 Execute

The reservation stations queue up instructions and their operands for multiple functional units. The reservation stations buffer each instruction until all its operands are ready and the necessary functional unit is available. The reservation stations dispatch multiple instructions in parallel to multiple functional units in each clock cycle (known as "multiple issue"). The functional unit calculates the result of the operation and sends it to the reorder buffer, as well as to any other reservation stations whose operands depend on the result. Buffering in the reservation stations means that instructions may not be executed in the order they were fetched (known as "out-of-order execution"), because the pipeline tries to avoid hazards and stalls by reordering the instructions (known as "dynamic pipeline scheduling") while maintaining the data flow structure of the program.

3.2.1.4 Commit

The commit unit uses the instruction entry in the reorder buffer to hold the results of the instruction execution until it determines that any speculated results were speculated correctly, and then marks the instruction entry as complete. The commit unit processes the reorder buffer in the order the instructions were fetched, so when the instruction at the head of the reorder buffer is marked as complete, it will perform any pending register writes or memory stores, and then remove the instruction entry from the reorder buffer. Alternatively, if the commit unit determines that the speculation was incorrect, it will discard the result in the reorder buffer, and remove the instruction entry. This approach of preserving the instruction fetch order in the commit process, called "in-order commit", allows the out-of-order pipeline to preserve the appearance of operating like a simple in-order pipeline. The commit unit only stores results to memory after any speculatively predicted branches that the store instruction depends on have been determined as correct.

3.2.1.5 Discussion

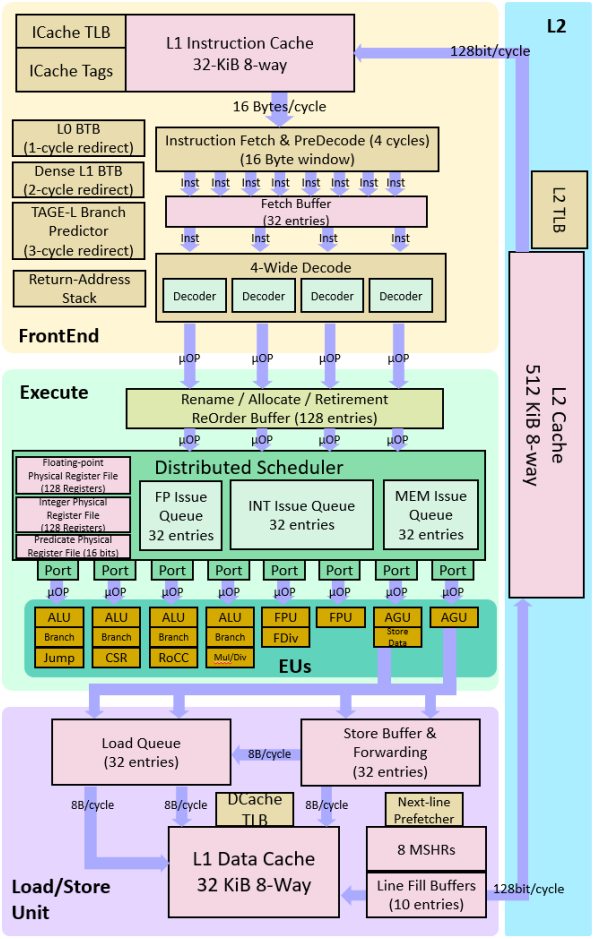

Figure 3.2: RISC-V BOOM microarchitecture. Reprinted from Zhao et al. [501], Figure 1.

A number of different real-world microarchitectures follow this general model of speculative pipelining. For example, Figures 3.2, 3.3, and 3.4 are modern examples of superscalar, speculative x86, ARM, and RISC-V microarchitectures, and though implementation details

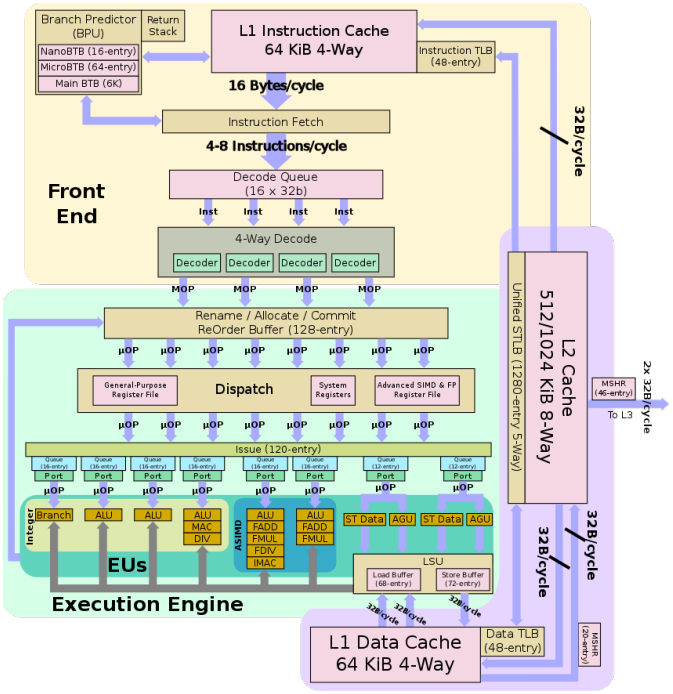

Figure 3.3: ARM Neoverse N1 microarchitecture. Reprinted from Schor [379].

differ, even a cursory examination reveals that they are all designed using the same fundamental pipeline components from Figure 3.1. The third generation of the Berkeley RISC-V Out-of-Order Machine (BOOM) in Figure 3.2, shows instruction fetch near the top center of the figure, branch predictor near the top left, reorder buffer near the center, feeding into reservation stations labeled "Distributed Scheduler", which feed into functional units labeled "EUs", and finally retirement/commit is annotated on the reorder buffer near the center. The ARM Neoverse N1 microarchitecture in Figure 3.3,[5] shows instruction fetch near the top center of the figure, branch predictor at the top left, reorder buffer near the center, feeding into reservation stations labled "Issue" and "Queue", which feed into functional units labeled "EUs", and finally retirement/commit is annotated on the reorder buffer near the center. The Intel Sunny Cove microarchitecture in Figure 3.4 is a single core within Intel's Ice Lake server processors,[6] and shows instruction fetch near the top center of the figure, branch predictor near the top left, reorder buffer near the center, with register alias tables near the center left, feeding into reservation stations labeled "Scheduler", which feed into the functional units labeled "EUs", and finally retirement/commit operates on the reorder buffer near the center right.

The critical security risk in this speculative implementation of branch instructions—and in any microarchitectures that follow a similar pattern—lies in the first component listed above, when instruction fetching predicts a particular result for a conditional branch, and then proceeds to speculatively fetch, issue, and execute instructions on that branch. Spectre-type attacks use techniques to mistrain the branch prediction by feeding it false history, thus "poisoning" the branch predictor state of the microarchitecture [72, pp. 4-6].

[5]The AWS Graviton processor custom-built for Amazon EC2 is based on the ARM Neoverse microarchitecture.

[6]The Ice Lake server processors are also branded as 10th generation Xeon processors, while the Ice Lake client processors are also branded as 10th Generation Core i3, i5, and i7 processors.

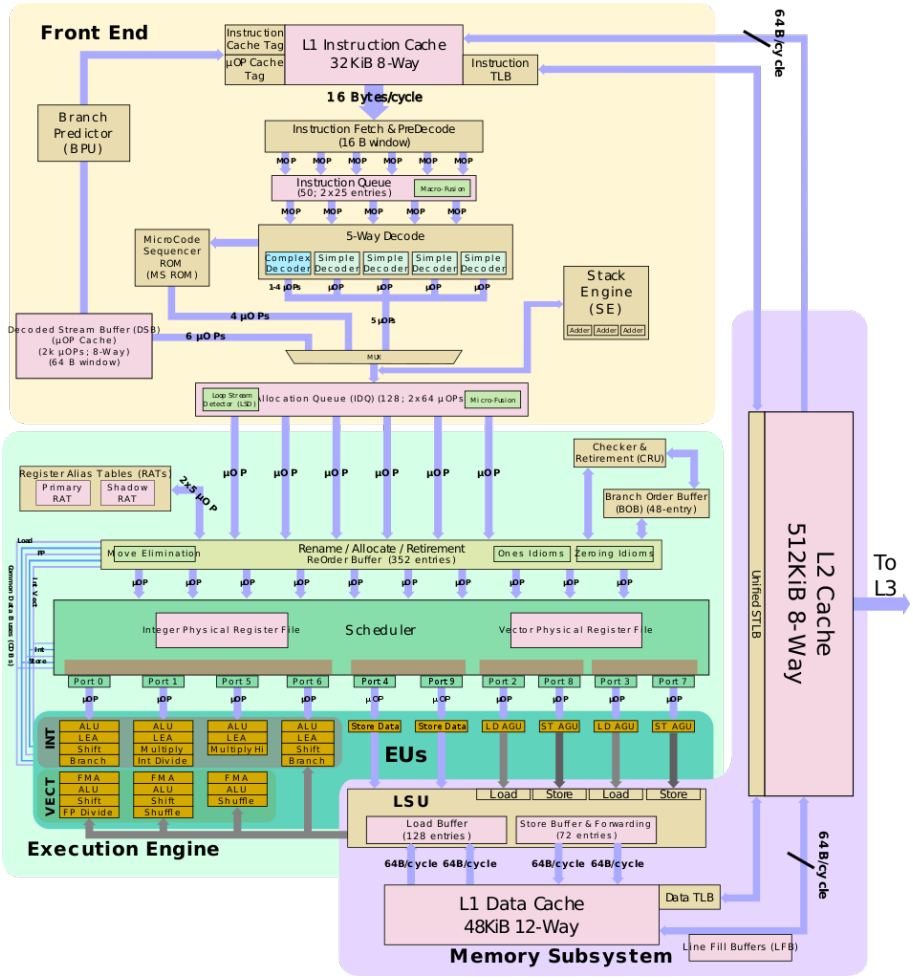

Figure 3.4: Intel Sunny Cove microarchitecture. Reprinted from Schor [380].

3.2.2 Speculative memory load instructions

As we observed in Section 3.2.1, speculative branch instructions are one microarchitectural root cause of Spectre-type vulnerabilities, but another root cause is speculative memory load instructions. Continuing to consider a superscalar microarchitecture that is roughly analogous to a modern x86 processor, speculative memory load instructions flow through the same essential components of a dynamically scheduled pipeline as speculative branch instructions.

3.2.2.1 Fetch

Instruction fetching may encounter an ordinary memory load instruction inside a predicted branch, in which case it will fetch (as well as issue and execute) that memory load instruction in exactly the same way it speculatively fetches all other instructions on the predicted branch.

3.2.2.2 Issue

Instruction issue places an entry for each memory load instruction into the reorder buffer, performs register renaming, and dispatches the instruction to the reservation stations, the

same as in Section 3.2.1.2.

3.2.2.3 Execute

Outside of any branch code path, another way that memory load instructions may be speculatively executed is through a memory dependence predictor that predicts which memory loads do not have a Store To Load (STL) dependency on any prior store instructions, so a memory load may be executed speculatively, before prior stores to the same address have been completed.

The reservation stations buffer memory load instructions until their operands are ready and the necessary functional unit (a load-store unit) is available. The functional unit executes the memory load operation, and sends the result (the value loaded from memory) to the instruction entry in the reorder buffer, and also to any other reservation stations whose operands depend on the result. If the memory load was executed speculatively, other instructions with a data dependency on the memory load may also be executed speculatively.

3.2.2.4 Commit

The commit unit uses the instruction entry in the reorder buffer to hold the results of the memory load instruction (the value loaded from memory), until it determines that the speculated memory load was speculated correctly. If the commit unit determines that the speculation was correct, it marks the instruction entry in the reorder buffer as complete, performs any pending register writes or memory stores, and removes the instruction entry from the reorder buffer.

If the memory load instruction was speculated because of branch prediction, the determination of correctness is based on whether the branch path was predicted correctly, and a misprediction will remove the instruction entry from the reorder buffer. If the memory load instruction was speculated because of the memory dependence predictor, the determination of correctness is based on whether the prediction that all prior stores were complete was correct, and a misprediction will discard the result in the reorder buffer, but will have to execute the memory load operation all over again to get the correct result, and also re-execute any other instructions that had a data dependency on the result of the memory load.

3.2.2.5 Discussion

The critical security risk in this speculative implementation of memory load instructions, and in any microarchitectures that use similar techniques, lies in the third component listed above, when the load-store unit executes the memory load operation. In modern architectures, a memory load operation is not a passive read directly from physical memory, it is a complex operation with cascading side-effects through multiple levels of memory cache, DRAM buffers, and translation lookaside buffers (TLB), and also has the potential to trigger exceptions. Some Spectre-type attacks—notably, the speculative store bypass variant reported by Horn [186]—use techniques to mistrain the memory dependence predictor, so it incorrectly speculates a memory load operation, leaving behind microarchitectural traces of a stale value that should have been overwritten by a prior store, and then access the stale value through cache-based side-channels [72, p. 6].

3.3 Spectre

Spectre is a hardware security vulnerability first discovered in 2017, but not reported publicly until January 2018 by Kocher *et al.* [226]. Together with Meltdown, Spectre is the first of a new class of vulnerabilities—known as transient execution vulnerabilities—that exploit weaknesses in certain low-level microarchitectural effects of out-of-order and speculative pipelines. While any out-of-order pipeline could be vulnerable to Meltdown, only speculative pipelines can be vulnerable to Spectre. Spectre is a fault analysis side-channel attack—it combines both fault-injection techniques to manipulate the victim into a vulnerable state and side-channel techniques to convey the exposed secrets to the attacker. The combination of the two techniques is what makes this class of vulnerabilities so powerful. The fault-injection phase of a Spectre-type attack mistrains a speculative predictor so it starts making false predictions. The victim blindly accepts the false predictions and proceeds to execute with either wrong values or wrong instructions, leaving a trail of microarchitectural state changes as it executes. In theory, those microarchitectural state changes are architecturally invisible if the speculated prediction proves to be false—the architectural changes are all cleaned away and the pipeline is flushed leaving no visible effects[7]—so they are "transient" in the sense that they only exist briefly before they disappear [72, 71]. But, during transient execution, the microarchitectural state changes that the victim made as a result of false predictions are microarchitecturally visible, so the attacker can access them through side channels. All the side channels used as the transmission phase of Spectre-type attacks can be used as stand-alone microarchitectural side-channel attacks, and as we discussed above in Section 3.1.3, some of those side channels have been known for decades. The uniquely interesting thing about Spectre is the initial fault-injection preparation phase, which tricks the victim into exposing its own secrets—the attacker manipulates the victim into executing instructions or values it never would have done non-speculatively, so the victim creates shared microarchitectural state it never would have created non-speculatively, specifically so the attacker can access that shared microarchitectural state through microarchitectural side-channels.

3.3.1 Characterizing the variants

The first few variants of Spectre published early in 2018 were novel, but also relatively simple. After 6 years and hundreds of published papers, the landscape today is a combinatorial explosion of variants and mitigations. Understanding the first few variants published is not enough to make sense of the entire class of Spectre-type vulnerabilities, but far too many hardware researchers and engineers make the mistake of stopping there.

The discouraging truth of Spectre is that potentially any speculative predictor could be used for the fault-injection preparation phase, and potentially any microarchitectural state could be used for the side-channel transmission phase. To compound the complexity, in the access phase any instructions executed transiently by the victim as a result of the fault-injection misprediction could take any action to leave transient traces in shared microarchitectural state, serving as a *gadget* that exposes secrets so they become vulnerable to side-channel transmission. All of those variations in the preparation, access, or transmission phases are still called "Spectre", because they all satisfy the fundamental definition of the technique—as Kocher *et al.* [226] described it in the very first paper, "Spectre attacks

[7]Some architectures are sloppier than others about cleaning up the side-effects speculation.

involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary."

The primary way of categorizing Spectre-type variants, shown in Table 3.1, is by the fault-injection attack vector used to trigger speculative execution in the preparation phase. While the initial Spectre variants reported in 2018 used a branch, return, or memory dependence predictor in the preparation phase, subsequent work on Spectre and other transient execution vulnerabilities has explored a more diverse collection of ways to trigger speculative execution in the pipelines of modern processors, as shown in Table 3.1 and further discussed in Section 3.5. The choice of predictor in the preparation phase has a significant impact on later phases of a Spectre attack. For example, there is a fundamental difference between Spectre variants with an attack vector of conditional branch prediction and Spectre variants with an attack vector of direct or indirect branch prediction, or return prediction. In some ways conditional branch predictors are less powerful attack vectors, because their control flow destinations are limited to two alternatives—either redirecting control flow to one specific label or continuing to the next instruction—rather than being able to redirect control flow to an arbitrary mispredicted address. But, conditional branch predictors also make predictions about the value evaluated by the condition, and that predicted value can be used in later phases of the attack. Some Spectre variants depend on a wrong value prediction, while other variants work equally well with any control flow predictor.

As we discussed in Section 3.1.3, many different microarchitectural states have been exploited in microarchitectural side-channel attacks. So, it should come as no surprise that the side-channel attack vectors used in the transmission phase of Spectre-type vulnerabilities have been equally diverse, some of the highlights are listed in Table 3.2. Not every microarchitectural side channel listed in Section 3.1.3 has a corresponding paper demonstrating that it can be exploited in a Spectre-type variant, and new microarchitectural side channels are still being discovered, so the list in Table 3.2 continues to grow. Over time, while new research publications continue to explore individual side channels to discover new Spectre-type variants, there is also a growing body of research into developing tools to find side channels that can be exploited by Spectre-type variants and other transient execution vulnerabilities, as discussed in Section 3.6.

3.3.2 Characterizing the countermeasures

Many countermeasures for Spectre-type vulnerabilities have been proposed, but overall the results have been disappointing [71, 139]. As Figure 3.5[10] illustrates, the performance penalties of proposed mitigations have been improving over time, and the proposals are trending toward mitigating more than one variant by considering root causes. Unfortunately, it is relatively common to see papers—such as Behrens *et al.* [43] or Guan *et al.* [176]—which claim to evaluate the overall performance of mitigating Spectre, but actually only evaluate a small subset of mitigations that are inadequate to mitigate all variants. So far, the only approach that eliminates all variants of Spectre is to eliminate speculation

[10]The data sources for Figures 3.5 and 3.6 are [11, 12, 20, 30, 35, 43, 58, 72, 75, 76, 90, 91, 114, 118, 127, 145, 151, 164, 171, 205, 220, 221, 222, 226, 233, 238, 239, 242, 246, 248, 255, 254, 257, 270, 286, 304, 312, 316, 323, 350, 354, 366, 367, 369, 368, 382, 385, 392, 422, 421, 423, 428, 432, 444, 447, 451, 452, 463, 476, 478, 484, 485, 491, 497, 496, 500, 502, 503]. Some performance results are self-reported, while others are reported by subsequent papers evaluating earlier papers.

Table 3.1: Spectre variants by preparation phase fault-injection attack vector

Predictor	Mechanisms	Examples
Pattern History Table (PHT) or Conditional Branch Predictor (CBP)	PHT/CBP poisoning: mistrains conditional branch prediction, to redirect control flow to the attacker's chosen branch path, so the victim transiently executes either wrong instructions or with wrong values.	Spectre-PHT (Spectre variant 1, "Input Validation Bypass") [226], Kiriansky and Waldspurger (Spectre variants 1.1 and 1.2) [223], NetSpectre [383], SGXSpectre [313], SiSCloak [66], HammerScope [95], SpecHammer [429], Schwarzl *et al.* [386]
Branch Target Buffer (BTB)	BTB poisoning: mistrains direct or indirect branch prediction, to redirect control flow to the attacker's chosen branch destination, so the victim transiently executes wrong instructions.	Spectre-BTB (Spectre variant 2, "Branch Target Injection") [226, 357, 447], SgxPectre [87], Spectre-BTB-SA-IP [72], SMoTherSpectre [49], Mambretti *et al.* [276], Straight-Line Speculation (BTB variants) [410], Retbleed [469]
Branch History Buffer (BHB)	BHB poisoning: mistrains indirect branch prediction, to redirect control flow to the attacker's chosen branch destination, so the victim transiently executes wrong instructions.	Spectre-BHB ("Branch History Injection") [37]
Return Stack Buffer (RSB) or Return Address Stack (RAS)	RSB poisoning: mistrains the RSB by executing call instructions to add invalid entries to the RSB, or explicitly overwrites return addresses, to redirect return control flow to the attacker's chosen destination, so the victim transiently executes wrong instructions.	Spectre-RSB (Spectre variant 5, "Return Address Injection") [275, 232], SgxPectre (RSB falls back on BTB) [87], Straight-Line Speculation (RSB variants) [410], Spring [468], Inception [434]
Memory dependence predictor	STL poisoning: mistrains store-to-load predictor, so the victim transiently loads stale values that should have been overwritten by intervening stores, and transiently executes with wrong values. If the stale value is a code pointer, it can redirect control flow to a gadget, so the victim transiently executes the wrong instructions.	Spectre-STL (Spectre variant 4, "Speculative Store Bypass") [186]
String Comparison Overrun (SCO)	Does not require mistraining or a leakage gadget, because a single instruction contains both the speculation trigger and the leaking memory access	Oleksenko *et al.* [315]
Zero Dividend Injection (ZDI)	Speculation induced by division instructions	Oleksenko *et al.* [315]

entirely, and while the approach is often dismissed for performance reasons without any actual performance measurements [226, 263, 383, 491, 72, 164, 369, 182, 463, 363], the few papers that do measure the performance of eliminating speculation [423, 350] reveal performance penalties comparable to other mitigations for Spectre.[11]

3.3.2.1 Software-only mitigation approaches

Some of the earliest mitigations proposed for Spectre were software workarounds for the vulnerabilities. These mitigations were inspired by earlier work on mitigating side-channel attacks for cryptographic software, where it was understood that the mitigations only needed to be applied to small but critical sections of code [99, 153, 417, 269, 295, 71, 79, 55]. Software-only mitigations have the advantage that they require no changes to the

[11]The two green "all variants" data points in Figure 3.5 are both non-speculative, as we discuss further in Section 7.1.

Table 3.2: Spectre variants by transmission phase side-channel attack vector

Channel	Mechanisms	Examples
L1 data cache	Leaks information using a cache-timing side-channel on the L1D cache	Take A Way [262][8], PMU-Leaker [342], most attack variants that succeed with L3 as a side channel also work on L1D
L1 instruction cache	Leaks information using a cache-timing side channel on the L1I cache	Mambretti *et al.* [276]
L2 cache	Leaks information using a cache-timing side-channel on the L2 cache	Most attack variants that succeed with L3 as a side channel also work on L2
L3/Last-level cache	Leaks information using a cache-timing side channel on the L3 cache or LLC, for example, Flush+Reload [493] or Prime+Probe [266]	SgxPectre [87]
Translation Lookaside Buffer (TLB)	Leaks information using a TLB-based side channel	Yan *et al.* [491], Khasawneh *et al.* [220], Kiriansky *et al.* [222], Loughlin *et al.* [270], Schwarz *et al.* [381], Seddigh *et al.* [387], PACMAN [353]
Vector instructions	Leaks information using a side channel based on differences in AVX2 instruction timing	NetSpectre [383], Weber *et al.* [462]
SMT and single-threaded port contention	Leaks information using a side channel based on execution timing differences between instructions on different execution ports	SMoTherSpectre [49], Fustos *et al.* [144], Spectre-STC [135]
Branch Target Buffer (BTB)	Leaks information using a side-channel based on timing differences between correct and false BTB predictions[9]	Weisse *et al.* [463], Mambretti *et al.* [276]
Micro-op cache	Leaks information using a micro-op cache-timing side channel	Ren *et al.* [355]
Instruction timing	Leaks information using a side channel based on variable-time arithmetic instructions	Zhang *et al.* [500], Rajapksha *et al.* [349]
Store and load buffers	Leaks information using a side channel based on execution timing analysis of load-store buffers	Timed Speculative Attacks (TSA) [82]
Rowhammer	Leaks information using a side channel based on measuring the power consumed by transient memory accesses	HammerScope [95]
Performance Monitor Unit (PMU)	Leaks information using a side-channel based on performance counters	PMU-Spill [343]

hardware, however, they have prohibitive performance penalties, and have proven to be inconsistently effective.

The very first paper on Spectre by Kocher *et al.* [226] suggested the insertion of speculation barrier instructions—for example, `lfence` on x86 or `sb` on ARM (added in v8.0)—which temporarily block speculative execution for instructions after the barrier, until speculation has resolved for all instructions before the barrier. The major vendors quickly adopted this approach and still actively recommended it today [401, 407]. Oleksenko *et al.* [316] demonstrated performance penalties as high as 440% for comprehensive use of `lfence`, which is worse than simply eliminating speculation [71]. The focus of much subsequent work on speculation barriers has been on limiting their use to improve performance [451, 422, 444, 206]. However, anything less than comprehensive use of speculation barriers means there is no guarantee that Spectre is fully mitigated [394, 422, 444]. Manual placement of speculation barriers is prone to developer mistakes, automatic placement often misses

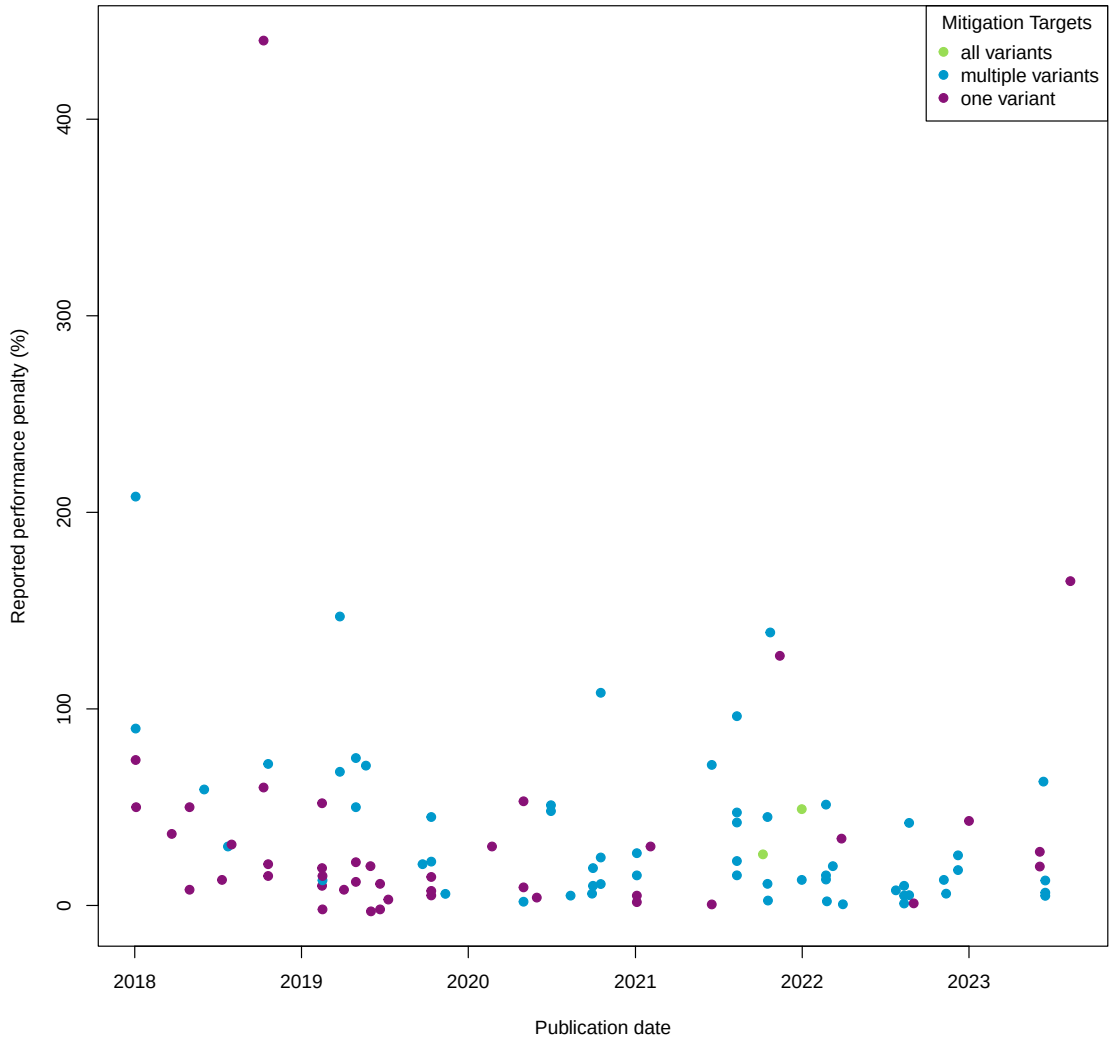

Figure 3.5: Performance penalty trends for Spectre countermeasures (2018-2023)

vulnerable code patterns, and even when the speculation barriers are correctly placed, race conditions in the specific microarchitecture implementation may allow secrets to be leaked past the barrier anyway [291, 312]. As with many Spectre mitigations, speculation barriers are often targeted only at the most well-known variants, and fail to provide protection beyond that narrow scope. For example, `lfence` stops some speculative accesses, but may or may not have any effect on the transmission phase. It is not effective against Spectre variants that use alternative side-channels as the transmission phase of the attack, such as side-channels based on AVX functional units, the TLB, the instruction cache [383], or micro-op cache [355], or against Spectre variants that use a speculative write to modify the gadget code [223]. Intel added a new Indirect Branch Predictor Barrier (IBPB) [406] instruction in 2018 to manually flush indirect branch predictor state so branch predictions after the barrier are not trained by branches before the barrier at a performance penalty of 24% to 53% [233], but Wikner and Ravazi [469] demonstrated that this mitigation was incomplete.

Another early software-only mitigation for Spectre-BTB was retpoline [436, 357], which

replaces an indirect branch instruction with a return sequence in the instruction stream. McIlroy *et al.* [286] reported a performance penalty of 152% for comprehensive use of retpoline, and subsequent work has focused on limiting the use of retpoline [233, 207]. Initially, retpoline was constructed on the assumption that the Return Stack Buffer could not be mistrained by attackers, but the Spectre-RSB variant [275, 232] later proved that assumption to be false and bypassed retpoline as a mitigation for Spectre-BTB. Maisuradze and Russow [275] suggested an alternative form of retpoline as a mitigation for the Spectre-RSB variant. The Retbleed [469] variant of Spectre demonstrated that retpoline is not an effective mitigation on architectures such as Intel and AMD that fallback to the Branch Target Buffer (BTB) to predict returns.

Speculative Load Hardening (SLH) is another software mitigation technique, which only mitigates the Spectre-PHT variant, proposed by Carruth [76] in 2018, adopted by both LLVM and GCC, with a reported performance penalty of 36% [72]. In 2021, Patrignani and Guarnieri [323] demonstrated that the original implementation of Speculative Load Hardening still allowed some data leaks, and proposed a stronger form of the mitigation, with a reported performance penalty of 127% [500]. In 2023, Zhang *et al.* [500] demonstrated that the original SLH mitigation is not effective against alternative side-channels in the transmission phase based on variable-time arithmetic instructions, and proposed an improved "ultimate" SLH mitigation, with a reported performance penalty of 165%.

Swivel [304] applied compiler transformations to sandboxed WASM code to limit some of the effects of Spectre vulnerabilities, however the approach relies on techniques like fences, ASLR, BTB flushing, and Intel's MPK which have been demonstrated not to be effective [383, 223, 72, 71, 355, 469, 387]. Several authors pointed out that Swivel and other compiler-based mitigations such as Jenkins *et al.* [203] and Venkman [392], have never been verified to work [90, 78, 500].

McIlroy *et al.* [286] noted that in their analysis, it was not possible to address the Spectre-STL variant using software-only mitigations.

While the initial mitigations proposed for Spectre were mostly implemented as software patches, over the years the trend has shifted toward mitigations implemented entirely in hardware or with an element of hardware acceleration, as shown in Figure 3.6. One factor in the decline of software-only mitigations is that hardware mitigations have tended to perform better than software mitigations. Another factor is that historically, hardware architectures were rarely designed with the intention of giving software control over speculation features,[12] so the range of options for mitigating Spectre entirely in software have been limited. The software mitigations proposed in recent years have often been refinements of software mitigations from previous years, such as successive attempts to improve the security of Speculative Load Hardening (SLH) [323, 500] or to improve the performance of fences [451, 503, 444, 502, 478].

3.3.2.2 Mitigation approaches that only consider cache-based side-channels

It is unfortunately common for papers about Spectre to focus on variants of the vulnerability that use cache-based side-channels, and then propose cache-based mitigations as if they could be solutions for Spectre. Some early papers went so far as to classify Spectre simply as a cache-timing side-channel attack without any mention of the transient execution effects involved [84, 345]. Such an oversimplification of Spectre-type vulnerabilities indicates

[12]The Intel i860 [230] was one noteworthy exception.

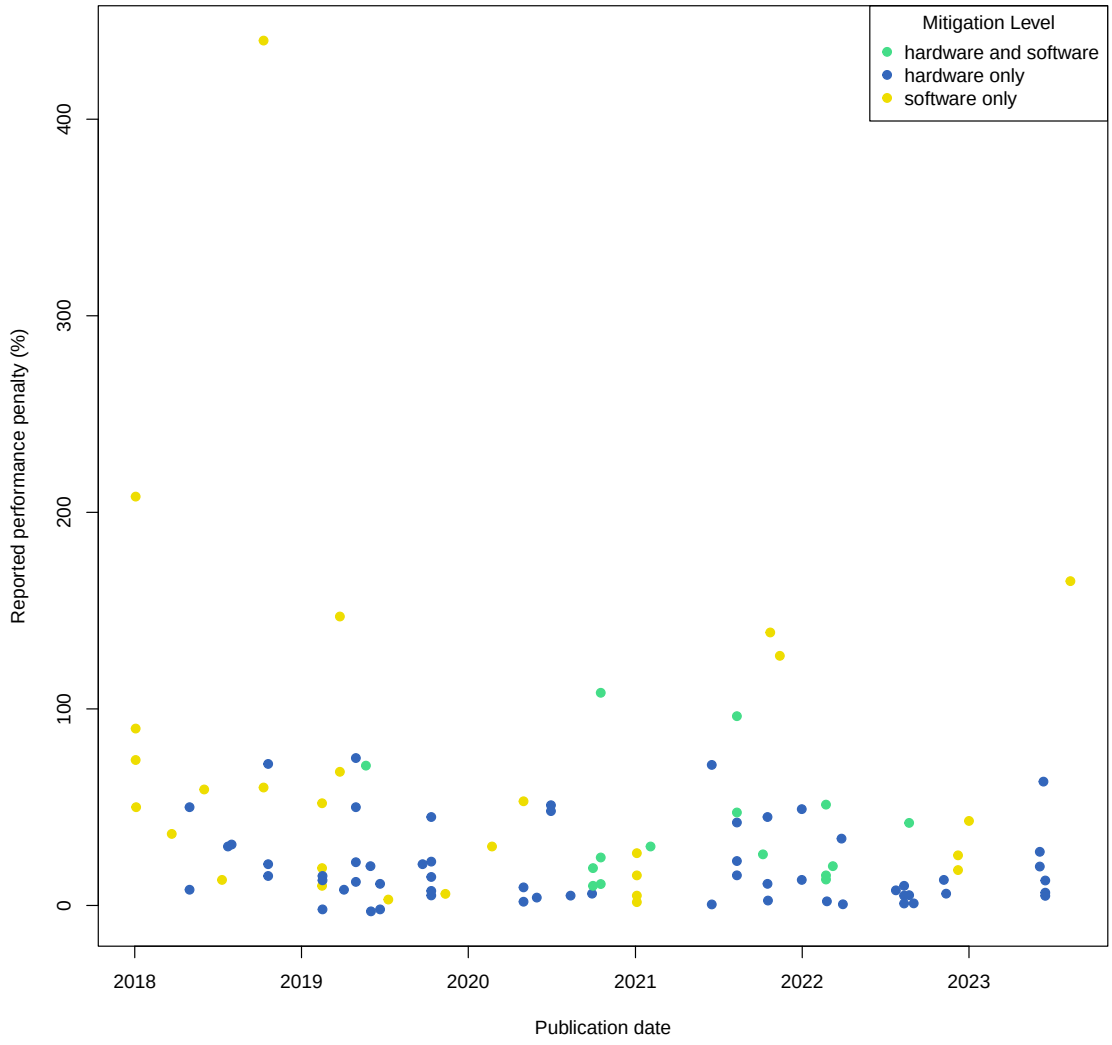

Figure 3.6: Performance penalty trends by implementation level for Spectre countermeasures (2018-2023)

a lack of understanding of past work and the available literature on Spectre. Even the very first paper on Spectre by Kocher *et al.* [226] explicitly discussed the fact that many different microarchitectural side-channels could be used for the transmission phase of Spectre, though the specific examples they chose to implement for the paper used cache-timing side-channels. While it is worthwhile to review these mitigation proposals as part of a comprehensive survey on Spectre, it is also important to recognize that exclusively cache-based mitigations can never be anything more than partial solutions [463, 82].

One group of papers in this category are really no more than general mitigations for cache-timing side-channel attacks. Although they mention Spectre (and sometimes also Meltdown) vulnerabilities as prominent examples, they do not make specific claims that their approach is a viable one for transient execution vulnerabilities. For example, CEASER [345] randomizes the location of lines in the last-level cache (LLC), and only claims to mitigate conflict-based cache attacks. DAWG [222] partitions caches into protection domains. On the more extreme side, Tsai *et al.* [435] redesign the memory hierarchy to replace caches with a memory-safe alternative they call Hotpads. While eliminating caches

would eliminate cache-based side-channels, it does not protect against other side-channels, and the authors did not verify whether Hotpads might be used as side-channels.

One group of mitigations, which came to be known as *invisible speculation*, focused on hiding changes to the cache. Yan *et al.* [491], Khasawneh *et al.* [220], Sakalis [368], Gonzalez *et al.* [164], Ainsworth and Jones [12], and Wu and Qian [484] added a small separate cache to store speculative loads. Sakalis *et al.* [369] proposed delaying updates to the cache hierarchy until after a load is no longer speculative, so L1 data cache hits would execute speculatively, but L1 data cache misses would delay until they could execute non-speculatively. Behnia *et al.* [42] and Fustos *et al.* [144] later demonstrated that invisible speculation approaches are not effective mitigations, because the delayed load introduces timing changes that can be observed in subsequent instructions that depend on the load, so the secret can be inferred even though the cache hierarchy was not immediately updated. Even worse, the subsequent dependent instructions may update the cache, making the secret easily accessible through the cache anyway, despite the delayed load. GhostMinion [11] was proposed to resolve the security problems with previous approaches to invisible speculation, however Yang *et al.* [492] uncovered a new variant of Spectre that bypasses GhostMinion.

CleanupSpec [367], ReversiSpec [485], ReViCe [221], and CacheRewinder [248] all take an approach of cleaning up the cache after speculation fails. However, all rollback techniques permit speculative execution to change the cache system, so they have the same problems as invisible speculation [42, 144], because the cache changes can still be observed by correct instructions executing at the same time as the misspeculated instructions, and the leakage succeeds before the cleanup finishes [11, 484, 179].

3.3.2.3 Mitigation approaches based on isolation

Another general approach to mitigating Spectre has been to increase isolation between user and kernel modes, threads, processes, or other security domains. One problem with isolation approaches to mitigating Spectre is that flushing or partitioning some microarchitectural state when changing domains is generally not sufficient to eliminate all microarchitectural traces, and so hardware remains vulnerable despite the mitigations [71]. A more fundamental problem with all mitigation approaches that rely on isolating one security domain from another is that not all Spectre variants are cross-domain attacks. Even the very first paper on Spectre [226] highlighted the fact that Spectre attack code could be in the same process and the same privilege level as the victim code, with a target of leaking memory that the attacker should not have access to because of a sandboxed interpreter, JIT compiler, or memory-safe language. Canella *et al.* [72] demonstrated that Spectre-PHT, Spectre-BHB, and Spectre-RSB variants still succeeded on Intel processors no matter whether the mistraining was done in the same-process or cross-process, and using the victim branch or a congruent branch. The same-process and cross-proces variants mostly succeeded on AMD and ARM too, though they both had some protections against cross-process congruent branch mistraining for Spectre-BTB, and ARM had some protection against cross-process mistraining for Spectre-RSB. Mitigations that merely isolate predictors across user/kernel mode or between threads are not effective against same-domain Spectre attacks [232, 37].

Intel and AMD added Indirect Branch Restricted Speculation (IBRS) [406, 401] as a hardware defense against Spectre-BTB, to flush branch predictor state when switching between user and kernel mode. Mambretti *et al.* [276] observed that IBRS was not

effective against their icache and Double BTI variants of Spectre-BTB. Intel and AMD later added enhanced IBRS (eIBRS) as an improvement to IBRS, however Barberis *et al.* [37] demonstrated that eIBRS was not effective because it only protected the Branch Target Buffer (BTB), so the mitigation could be bypassed by mistraining the Branch History Buffer (BHB) instead. ARM added similar features in the form of "IbrsSameMode" and CSV2 features, which were also vulnerable to the BHB variant of Spectre [37]. Furthermore, Barberis *et al.* [37] discovered variants of Spectre-BTB using same-mode indirect branch mispredictions (kernel-to-kernel), so that eIBRS and other isolation-based mitigations in general are not sufficient protection.

Intel and AMD added Single Thread Indirect Branch Predictors (STIBP) [406, 401] to isolate branch predictors for different hardware threads on the same core, preventing branch predictors in one thread from influencing branch predictions in other threads. STIBP has been reported to be effective as a partial mitigation only for cross-thread mistrainings [276], with performance penalties in the range of 50% [74], and both Intel and AMD have recommended against enabling STIBP by default [106]. STIBP has no effect on co-resident processes [449] or other same-domain mistrainings.

Wistoff *et al.* [476, 478] proposed a `fence.t` instruction to provide temporal partitioning, and Escouteloup *et al.* [127] proposed thread-level security domains called "domes", to introduce additional levels of isolation on RISC-V processors, but neither approach has any effect on same-domain attacks.

3.3.2.4 Mitigation approaches based on selective speculation

The most successful mitigation approaches to Spectre, in terms of both security and performance, have turned out to be the ones that restrict speculation. These approaches are based on a growing understanding that truly mitigating Spectre at the transmission phase (the side-channel leakage attack vector) would require blocking all known microarchitectural side-channels as well as any that might be discovered in the future [171]. Identifying the initial speculative access of the secret is a more tractable problem than chasing down every possible secondary transmission channel. And, once you have identified which instructions are risky to speculate, it is easier to prevent speculative execution than to chase down all the side effects after speculative execution has already happened.

As a mitigation for Spectre-STL, Intel introduced a processor mode Speculative Store Bypass Disable (SSBD) [406], which prevents loads from executing if they bypass any stores, so attackers cannot read stale values. This mitigation only targets Spectre-STL variants and has no effect on other Spectre variants. Initially measured at an 8% performance penalty in 2018 [463], Behrens *et al.* [43] observed that grew to a 34% performance penalty by 2022, and theorized one cause of the performance loss could be newer processors shipping more complete implementations of SSBD than was possible with the original microcode patches.

Some approaches to selective speculation simply delay the execution of all instructions that may have speculative sources of data as operands. NDA [463] restricts data propagation after an unresolved branch or unresolved store address. It begins with the assumption that instructions can execute speculatively as long as their operands are the results of "safe" instructions. They regard any instruction following a branch instruction as unsafe until the branch target and direction is resolved, and any load instruction as unsafe if it follows a store with an unresolved address. NDA then delays execution of any instruction with unsafe operands until those operands can be marked as safe, because the original

speculation trigger for that operand has resolved and is no longer speculative. SpecShield [35] is similar to NDA, but focuses more on load instructions as sources of speculative data forwarding, and makes some minimal attempt at identifying which instructions are a lower risk for leaking forwarded speculative data. NDA has performance penalties as high as 45%, and SpecShield 21%. Jin *et al* [205] attributed the poor performance of both approaches to the way they delay execution of a large number of instructions that never could have caused changes to the microarchitectural state anyway, and so would have been safe to execute speculatively.

Some approaches to selective speculation are based on hardware taint tracking techniques, inspired by previous work on information flow tracking techniques [426, 425]. Speculative Taint Tracking (STT) [497] begins with the assumption that it is safe to speculatively execute instructions and speculatively forward their results to other instructions, as long as: 1) the forwarded results are marked as "tainted"; 2) the taint propagates as the forwarded results are used as operands for subsequent instructions; and 3) any tainted operands delay the execution of instructions that could serve as a transmission side-channel until the original instruction that tainted the operand is no longer speculative. STT was only proposed as a mitigation for Spectre-PHT variants, at a reported performance penalty of 14.5%, however Loughlin *et al.* [270] later measured the performance penalty of STT as high as 44.5% for protecting data in memory, and as high as 63.4% when extended to protect data in registers. One key challenge of the STT approach is identifying all the instructions that could be used as transmission side-channels, and Jin *et al* [205] and Loughlin *et al.* [270] later identified that STT does not catch Spectre variants using speculative store instructions in the transmission phase with side-channels based on the TLB, store buffer, or load-store aliasing. Choudhary *et al.* [91] observed that STT only prevented speculative transmission of data that was accessed speculatively, but failed to protect data that was originally accessed non-speculatively, so their Speculative Privacy Tracking (SPT) extends the idea of STT, by tainting the results of a much larger set of data access instructions, with a performance penalty as high as 45%. Speculative Data-Oblivious Execution (SDO) [496] extended STT by allowing some transmission side-channel instructions to execute speculatively if they are independent of sensitive data, at a reported performance penalty of 10%. Zhao *et al.* [502] and Kvalsvik *et al.* [239] tried to improve the performance of STT and other similar approaches, by altering the behavior of speculative loads, reporting performance penalties of 13.2% and 4.9% respectively for their implementations of STT.

Dolma [270] is conceptually similar to STT, but instead of taint tracking forwarded results of prior instructions, it tracks speculative control dependencies (on prior branch instructions) and speculative data dependencies (on prior load instructions). Dolma marks micro-ops in the reorder buffer with the speculative control or data dependency, and delays their execution until the dependency is resolved because the original store or load is no longer speculative. Dolma reported a performance penalty of 42.2%, and claimed to protect against all transient execution attacks, but Jin *et al* [205] noted that Dolma does not protect against a load-load reordering side channel, as identified by Yu *et al.* [497]. Conditional Speculation [255] also tracks dependencies on prior branch and load instructions like Dolma, however it only delays execution of potential transmission side-channel instructions if they would change cache contents due to mis-specuation because of a cache miss. This more limited approach lowers the performance penalty to 12.8%, but still leaks information on cache hits [179] and with side-channels other than cache [205].

Ravichandran *et al.* [353] noted that mitigations based on information flow tracking such as STT, NDA, and Dolma only consider load instructions as the source of the taint, so they are not effective against variants of Spectre where the speculative taint has a different source, such as a pointer authentication instruction. The SpecHammer [429] variant of Spectre-PHT defeats some taint tracking mitigations by using Rowhammer to flip bits in the victim code, so code that would not ordinarily work for the access phase of Spectre-PHT becomes a viable attack vector.

SpecTerminator [205] refines earlier selective speculation approaches with performance improvements to taint tracking, and by applying different delayed execution techniques to different kinds of sensitive instructions—TLB request ignoring, extended Delay-on-Miss, delayed squash, and selective issue. SpecTerminator considers side-channels based on the TLB, DRAM, BTB, and port contention in addition to cache-based side channels. Similar to other selective speculation approaches, SpecTerminator uses taint tracking for potential transmission side-channel instructions (loads or stores) that depend on earlier access instructions (loads). But, instead of only delaying execution of transmission instructions, they delay TLB requests, which blocks more potential side-channels at an earlier stage of the pipeline. This approach also delays issue of branch instructions that depend on a prior speculative load, to prevent speculative updates to other microarchitectural states that enable BTB or port contention side channels. And, this approach delays squashes to protect against Spectre-STL variants and load-load reordering. SpecTerminator reported an impressive 6% performance penalty for mitigating the subset of Spectre variants they considered. However, Ghaniyoun [156] independently evaluated the SpecTerminator implementation and measured the performance penalty at 25%—significantly higher than the 6% reported in the original paper—and determined that TLB requests were not being ignored as intended.

SafeBet [171] focuses on the access phase of a Spectre attack, and delays execution of data access instructions until they are non-speculative. To improve performance, the approach uses a Speculative Memory Access Control Table (SMACT) to track prior non-speculative data accesses within the code region of a trust domain, and allows speculative data access instructions to execute if they are accessing the same location in the same region as a prior non-speculative data access. The SafeBet paper claims to mitigate all variants of Spectre, but then goes on to say the approach does not handle side channels based on micro-op caches. The approach only considers load instructions as sources of speculative data, so the limitation that Ravichandran *et al.* [353] identified for STT, NDA, and Dolma would also apply to SafeBet. And, SafeBet is fundamentally an isolation mitigation, so it offers no protection against same-domain Spectre variants, as discussed in Section 3.3.2.3.

The greatest challenges for selective speculation mitigation approaches is determining where speculation is safe or unsafe, and how to disable speculation with the least possible disruption to legacy software stacks while providing strong security guarantees. Manual approaches are possible—leaving the decision of whether speculation is safe or unsafe to the software or compiler developer—but they can never provide strong security guarantees. The approaches described in this section are more automated—the pipeline makes all the decisions about where speculation is safe or unsafe. So far, these automated approaches still have not managed to provide strong security guarantees, because they miss some scenarios where speculation is unsafe or because the implementation fails to disable speculation where the design intended. But, over time selective speculation approaches have been getting

closer to providing a comprehensive solution to Spectre with strong security guarantees. There may be room for a middle-ground selective speculation approach that provides strong security guarantees by disabling speculation for a security domain—such as a container, VM, secure enclave, serverless function, or small region of code—to protect code within the security domain from both cross-domain transient execution attacks launched outside the security domain and same-domain attacks launched within the security domain, and also serve as a sandbox preventing code inside the security domain from launching cross-domain attacks on any other part of the system.

3.4 Meltdown

Like Spectre, Meltdown is a transient execution vulnerability first discovered in 2017 and reported publicly in January 2018, by Lipp *et al.* [263], in a preprint which was republished later that year at the USENIX Security Symposium in June [264]. Also like Spectre, Meltdown is a fault analysis side-channel attack—it combines both fault-injection techniques to manipulate the victim into a vulnerable state and side-channel techniques to convey the exposed secrets to the attacker. Unlike Spectre, Meltdown does not use speculation as an attack vector, so an out-of-order pipeline can be vulnerable to Meltdown, even if it has no speculative features.

Research on Meltdown variants and mitigations has been far less extensive than Spectre, probably partly due to the fact that AMD, ARM, and IBM processors were never vulnerable to some variants of Meltdown [72, 155], so we have always known that hardware mitigations for Meltdown could have reasonable security and performance. Eventually, even Intel figured out that faulty reads could just return zero, preventing the leak of secret information [155].

3.4.1 Characterizing the variants

A number of variants of Meltdown have been reported, primarily focused on unauthorized access to some value protected by a permission check, and the defining characteristics of all variants are two phases: 1) triggering an exception for a failed permission check in the context of transient execution so the exception is delayed; and 2) leaking the unauthorized value through microarchitectural side channels. The permission check will ultimately fail and raise an exception, but in the context of transient execution, the exception is delayed until the transient instruction sequence commits. Some microarchitecture implementations made the design choice to update shared microarchitectural state during transient execution as if the permission checks were successful, and to allow subsequent transient instructions in the sequence to operate using the unauthorized value. In theory, those changes are only temporary and never architecturally visible, but in practice, shared microarchitectural state can be observed by an attacker and leaked over side channels.

The primary way of categorizing Meltdown-type variants, shown in Table 3.3, is by the exception used in the preparation phase. While AMD was not vulnerable to earlier variants of Meltdown, it was vulnerable to the Meltdown-BND variant [72] and to new variants reported by Xiao *et al.* [489].

A secondary way of categorizing Meltdown-type variants is by the microarchitectural states used in the transmission phase of attack—Table 3.4 shows some of the highlights. There have been fewer attempts to replicate Meltdown variants across a diverse collection

Table 3.3: Meltdown variants by exception

Exception	Permission Bit	Mechanisms	Examples
page fault	user/supervisor page-table attribute	Supervisor-only Bypass: bypasses user/supervisor permission checks to read unauthorized kernel memory from user space.	Meltdown (original variant, "Rogue Data Cache Load") [263, 264]
page fault	read/write page-table attribute	Read-only Bypass: bypasses read/write permission checks to transiently write over read-only data within the current privilege level. May be used, for example, to bypass the hardware-enforced isolation of software-based sandboxes.	Meltdown-RW (also inaccurately called "Spectre variant 1.2") [223, 72][13]
page fault	page-table present bit or reserved bit	L1 Terminal Fault (L1TF): bypasses Intel SGX enclave or operating system or hypervisor isolation to read unauthorized memory across isolation boundaries.	Foreshadow (Intel SGX) [441], Foreshadow-NG (OS and hypervisor) [464], Foreshadow-VMM (VM guest to host) [62]
page fault	Intel memory-protection keys for user space (PKU)	Protection Key Bypass: bypasses hardware-enforced read and write isolation, to leak or modify protected memory.	Meltdown-PK [72]
page fault	not present, all access to the page has been revoked	Write Transient Forwarding (WTF): store buffer	Fallout [294]
general protection fault	*N/A*	System Register Bypass: bypasses permission checks on privileged system registers to leak system register contents.	Meltdown-GP (also called variant 3a) [72]
device not available exception	*N/A*	FPU Register Bypass: bypasses isolation of floating point unit or SIMD registers across context switches, to leak register contents.	Lazy FP [409]
bound range exceeded exception	*N/A*	Bounds Check Bypass: bypass hardware-enforced array bounds checking[14] to access out-of-bound array indices.	Meltdown-BR [118, 72] including Meltdown-MPX [200] and Meltdown-BND [72]

of different side-channels, because it quickly became clear that it was feasible to block Meltdown in the preparation and access phases of the attack, so the side channel used in the transmission phase is less interesting.

Table 3.4: Meltdown variants by transmission phase side-channel attack vector

Channel	Mechanisms	Examples
L1 data cache	Leaks information using a cache-timing side channel on the L1D cache	L1TF variants [441, 464, 62] and SMAP and MPK variants [489] only work on L1D
L3 cache or LLC	For example, Flush+Reload [493] or Prime+Probe [266]	Meltdown (original variant) [263, 264], Meltdown-GP (also called variant 3a) [72], Meltdown-PK [72], Lazy FP [409]
uncached memory	Leaks information using a DRAM-based side channel	Meltdown (original variant) [263, 264]
Translation Lookaside Buffer (TLB)	Leaks information using a TLB-based side channel	Schwarz *et al.* [381], Seddigh *et al.* [387]

3.4.2 Characterizing the countermeasures

A number of different countermeasures were proposed for Meltdown-type attacks, but ultimately the right answer was fairly simple: always do permission checks first, and never update shared microarchitectural state or forward the results of data accesses until after the permission checks are successful [491, 155]. It is fine to delay raising the exception until after the transient instructions commit, so out-of-order and speculative pipelines can be safe from Meltdown-type vulnerabilities as long as the microarchitecture design is done correctly. The only reason Meltdown-type attacks ever worked, is that hardware designers assumed that microarchitectural state created in the context of transient execution was safely hidden so deep in the hardware that it could never be accessed, but that assumption was false.

There were some early software-only mitigations for Meltdown, which are still in use on legacy hardware. The KAISER [174] patch to Kernel Address Space Randomization (KASLR) was demonstrated to be an effective mitigation for the first User/Supervisor variant of Meltdown [263], and was later implemented in the Linux Kernel as Kernel Page Table Isolation (KPTI) [102]. However, KAISER and KPTI are only isolation mitigations between kernel and user space memory, and so the mitigation has no effect on other variants of Meltdown or on same-mode attacks. Hua *et al.* [192] measured the KPTI mitigation at a 30% performance penalty, and developed an alternative mitigation, EPTI, that uses extended page tables (EPT) instead of guest page tables for isolation at a 13% performance penalty. While EPTI performed better than KPTI, it was not more effective. Page Table Entry (PTE)-Inversion [104] was implemented as a mitigation for the L1 Terminal Fault (L1TF) variants of Meltdown, by ensuring that addresses used following a translation failure do not point to a valid page frame [155]. He *et al.* [182] observed that software-only mitigations have been far less successful for Meltdown than they were for Spectre, because the microarchitectural causes for Meltdown-type vulnerabilities occur within a single instruction, while the microarchitectural causes for Spectre-type vulnerabilities occur in the interaction between instructions.

Isolation mitigations were also tried, such as flushing the L1 cache on context switches or careful scheduling to prevent processes or VMs from executing on the same core or thread [464, 155]. And, a number of mitigations for Spectre also claimed to mitigate Meltdown, with varying degrees of success [463, 220, 171], even though Meltdown-type attacks really are fundamentally different than Spectre-type attacks [182]. The proliferation of hardware and software mitigations necessary to catch all variants of Meltdown have been deeply unappealing compared to AMD's simple answer of "just don't be vulnerable in the first place" [463, 155, 311].

However, just because it is possible to eliminate Meltdown-type vulnerabilities from out-of-order and speculative cores with careful microarchitecture design, does not mean that every microarchitecture implementation has successfully done so. This is one of many reasons why pre- and post-silicon hardware security verification techniques are critical for modern hardware design, as discussed in section 3.6.

3.5 Transient execution vulnerabilities beyond Spectre and Meltdown

Because Spectre and Meltdown were the first transient execution vulnerabilities discovered, they have received the most attention, but researchers continue to find new transient execution vulnerabilities. The vulnerabilities all share the defining characteristic of using transient execution effects as an attack vector, but otherwise they are a diverse collection. Some are side-channel attacks with a goal of leaking secrets to violate confidentiality like Spectre and Meltdown, but others are straight up fault-injection attacks with a goal of violating integrity.

3.5.1 Side-channel attacks inspired by Meltdown

Some transient execution vulnerabilities use different ways of inducing transient execution. Rather than exploiting delayed exceptions like Meltdown, Nemesis [439] exploits the fact that interrupts are delayed until instruction retirement. The target of Nemesis-type attacks is to leak instruction timings from secure enclaves. Fallout [294] uses microcode assists as a trigger for transient execution rather than exceptions, leaks information via the store buffer, and is able to bypass the Kernel Page Table Isolation (KPTI) countermeasure for Meltdown.

Possibly inspired by an early mention of line-fill buffers as a potential attack vector for Meltdown [264], microarchitectural data sampling (MDS) attacks exploit the transient effects of line-fill buffers, load ports, and store buffers. MDS attacks are data speculation attacks like Meltdown and some variants of Spectre, but they do not depend on the attack techniques of circumventing failed permission checks (like Meltdown) or mistraining predictors (like Spectre). Similar to Meltdown, MDS attacks exploit the way that out-of-order pipelines defer exceptions until instruction retirement. But, unlike Meltdown, MDS attacks do not access the actual data at the operand address of the faulting load instructions, instead, they access data from various other microarchitectural resources on the CPU. Rogue In-flight Data Load (RIDL) [376] cannot be mitigated in software, and specifically defeats mitigations such as Kernel Page Table Isolation (KPTI), Page Table Entry (PTE) inversion, Speculative Store Bypass Disable (SSBD), and L1 data cache flushing, and works both cross-context and same-context. ZombieLoad [384] amplifies microarchitectural data sampling (MDS) and bypasses mitigations for both Meltdown-type attacks and other MDS-type attacks. CacheOut [375] bypasses mitigations that Intel put in place on the Whiskey Lake architecture to protect against other MDS-type attacks such as Fallout, ZombieLoad, and RIDL. SGAxe [374] adapts CacheOut to target SGX enclaves. Medusa [297] is a more focused MDS-type attack than ZombieLoad or RIDL, which only targets data loads caused by write combining operations, and can only be successfully mitigated if hyperthreading is disabled. Ragab *et al* [346] discovered another variant of an MDS-type attack that leaks information using a global staging buffer shared between all CPU cores and defeats mitigations based on spatial or temporal partitioning or isolating workloads on separate cores. Witharana and Mishra [480] reported another MDS variant that works on AMD architectures, which were not vulnerable to previous variants.

The Gather Data Sampling (GDS) [296] attack exploits the x86 `gather` instruction in the context of transient execution to leak stale data from the shared SIMD register buffers.

3.5.2 Side-channel attacks inspired by Spectre

Rokicki [363] demonstrated that processors based on Dynamic Binary Translation (DBT), such as Nvidia Denver [54] or Hybrid-DBT [364], are vulnerable to variants of Spectre even though the underlying hardware is strictly in-order, because the DBT engine introduces conditional branch prediction and memory dependency prediction as it translates and optimizes the binaries.

3.5.3 Other transient execution vulnerabilities

Not all transient execution vulnerabilities are side-channel attacks, some use transient execution effects for other purposes. Like Meltdown, Load Value Injection (LVI) [442, 121] begins with a preparation phase of triggering an exception, but the target of the attack is fault-injection rather than side-channel leakage, specifically to inject false values into the victim's transient execution (violating integrity). Also, LVI attacks run in the victim domain, so cross-domain isolation is not effective as a mitigation [71]. The Gather Value Injection (GVI) [296] attack extends LVI using the Gather Data Sampling (GDS) technique, with the same target of value injection.

Ragab *et al.* [347] explored transient execution vulnerabilities on Intel and AMD induced by machine clears. Their Speculative Code Store Bypass (SCSB) variant allows attackers to execute stale code, while their Floating Point Value Injection (FPVI) variant is similar to LVI but injects operands into floating point operations. Both are primarily integrity attacks, but they can also be combined with side-channel attack techniques (violating confidentiality).

Like Spectre, ExSpectre [449] has a preparation phase that mistrains branch predictors, but unlike Spectre, it uses transient execution effects to hide malware from static and dynamic analysis techniques, with a primary target of arbitrary code execution (violating integrity). For example, ExSpectre is capable of running system calls to launch a dial-back TCP shell. Isolation techniques such as Intel's Single Thread Indirect Branch Predictors (STIBP) are not effective mitigations against ExSpectre because the attack code and the victim code run in the same context.

GhostKnight [499] has a preparation phase that mistrains branch predictors, but uses speculation execution to amplify the Rowhammer fault-injection attack, extending the reach of the attack to cross privilege boundaries (violating integrity). Spoiler [202] also uses transient execution effects to amplify Rowhammer attacks.

BlindSide [160] is a speculative probing technique that uses speculative execution to amplify a simple memory corruption attack into a speculative control-flow hijacking attack, with targets ranging from leaking sensitive data, to arbitrary code execution, all the way to full-system compromise. Speculative probing attacks are able to bypass mitigations designed to prevent speculative control-flow hijacking such as retpoline, IBPB, IBRS, and STIBP.

Another category of vulnerabilities that can use transient execution effects are microarchitectural replay attacks (MRA) such as MicroScope [395, 396, 370], where the attacker forces pipeline flushes so the victim instructions are repeatedly re-executed. MRA techniques can reduce the noise in side channels used to leak secrets, making transient execution vulnerabilities and other vulnerabilities easier to exploit.

3.6 Hardware security verification for transient execution

Over the years of research into the transient execution vulnerabilities, the emphasis has shifted away from looking for some magic hardware or software countermeasure that will preserve the performance benefits of transient execution while eliminating the security risks. Instead, there is a growing understanding of transient execution as one of those complex multilayered problems, like memory safety, where human errors by the people designing and implementing the systems plays a significant role, and expecting hardware engineers to manually catch all the security flaws is an inadequate answer. In response—and as part of a broader trend of increasing interest in hardware security verification [125, 24, 327, 70, 479, 236, 191, 190, 115, 138, 189]—there has been a rise in academic and commercial tools to inspect, test, fuzz, and scan for transient execution vulnerabilities, at the hardware-level, at the software-level, or with formal models.

Hardware security verification tools are not capable of guaranteeing that a speculative or out-of-order processor is invulnerable to all transient execution vulnerabilties, but they can help improve security by determining whether a specific processor is vulnerable to specific known variants, confirming whether implemented and deployed mitigations actually work, and identifying risky patterns in the design and implementation of hardware. If you are a hardware vendor who produces hardware with transient execution features, you should be using hardware security verification tools to catch flaws in your microarchitecture design and implementation, and to minimize the damage you might cause your customers. If you are a software developer, the post-silicon and software-only tools can help you discover how secure your hardware really is, and what adaptations you might need to make to protect your software and your users.

Among the major server hardware vendors, we know from publicly available information that ARM has used hardware security verification tools for the transient execution vulnerabilities [272]. We have no information about Intel or AMD, however based on available evidence—specifically the way that AMD was not vulnerable to several variants of Meltdown and Spectre before they were even reported—we suspect that AMD does use microarchitecture-level hardware security verification tools.

3.6.1 Formal model verification

Spectector [178] was an early attempt at detecting Spectre vulnerabilities using symbolic execution and comparing the microarchitectural information flows between speculative and non-speculative execution. Loughlin *et al.* [270] argued that Spectector was too restrictive and delayed some transient instructions that would have been safe to execute speculatively. CacheFix [84] and CheckMate [433] both do formal modeling of microarchitectural state to detect vulnerabilities, but only for cache-timing side channel attacks. Guarnieri *et al.* [179] extended Spectector with a concept of speculation contracts. Fabian *et al.* [130] extended Spectector beyond modeling branch instructions to also model store and return instructions, so it could detect variants of Spectre-PHT, Spectre-RSB, and Spectre-STL.

Cauligi *et al.* [78] surveyed formal frameworks for software mitigations. Cheang *et al.* [86] formally defines a class of information flow security properties for reasoning about the security of microarchitectural speculation features, and operational semantics for an intermediate assembly representation which can run small programs and verify if

they conform to the secure speculation property. Griffin and Dongol [172] implemented the secure speculation properties defined by Cheang *et al.* in the Isabel/HOL proof assistant. Unique Program Execution Checking (UPEC) [134, 133, 135] is a structured and systematic formal methodology for hardware security verification that targets transient execution vulnerabilities at the register-transfer level (RTL) of the hardware design and implementation workflow. InSpectre [177] proposes a formal microarchitectural model of out-of-order and speculative features used as attack vectors in a variety of transient execution vulnerabilities, and implements the model as a Machine Independent Language (MIL) as an abstract microcode target language for translating ISA instructions. Pitchfork [79] performed constant-time code analysis on an abstract model, but lacks microarchitectural implementation details. KLEESpectre [450] extends the KLEE symbolic execution engine with modeling of cache and speculative execution.

Pensieve [492] formally modeled early-stage microarchitectural designs, to evaluate the security of proposed mitigations for transient execution vulnerabilities. Ponce-de-león and Kinder [250] use the CAT modeling language for memory consistency to implement an axiomatic framework to detect attacks and validate defenses for transient execution vulnerabilities, including execution models of speculative control flow, store-to-load forwarding, predictive store forwarding, and machine clears. Mathure *et al.* [280] applied refinement-based formal verification methods to detect whether a microarchitecture design is vulnerable to variants of Spectre.

3.6.2 Pre-silicon verification

Hu *et al.* [190] surveyed hardware verification strategies based on information flow tracking, for a variety of hardware security vulnerabilities including the transient execution vulnerabilities.

Barber *et al.* [36] instrumented RTL simulations to produce detailed execution traces of microarchitectural structures, and perform differential analysis on the traces to identify potential attack vectors. TEESec [157] is a pre-silicon framework for discovering microarchitectural vulnerabilities in secure enclaves, by profiling the processor design for microarchitectural structures relevant to enclave data, crafting verification gadgets to exercise all possible access paths to the enclave data, running the verification gadgets through a cycle-accurate RTL simulation of the design-under-test, and analyzing the simulation logs for traces that violate microarchitectural security principles.

SpecDoctor [195] is an automated RTL fuzzer to detect both Spectre-type and Meltdown-type vulnerabilities, which systematically tests a comprehensive set of configuration options while selectively monitoring specific RTL components to discover constraint violations, then chains those violations to construct concrete proof-of-concept transient execution attack instruction sequences. SpecDoctor was implemented by adding monitoring logic for reorder-buffer rollback events to the Chisel source code for two RISC-V core implementations, BOOM and NutShell. IntroSpectre [158] is another RTL fuzzer to detect Meltdown-type leaks.

3.6.3 Post-silicon verification

SpeechMiner [489] is a software framework focused on detecting transient execution vulnerabilities on existing hardware, by generating sequences of instructions as tests. It models Meltdown-type vulnerabilities as a race condition between data fetching and

processor fault handling and models Spectre-type vulnerabilities as a race condition between side-channel transmission and speculative instruction squashing. SpeechMiner was only implemented for 32-bit and 64-bit x86 architectures, not ARM or RISC-V. Revizor [314, 315] is a black-box testing framework that detects microarchitectural leakage on x86 CPUs, using a concept of speculation contracts.

Transynther [297] used fuzzing techniques to systematically identify whether hardware is vulnerable to variants of Meltdown and microarchitectural data sampling (MDS) attacks. Transynther was only implemented for x86 (Intel and AMD) and has not been ported to ARM or RISC-V. Osiris [462] and SIGFuzz [349] are also fuzzing frameworks to detect microarchitectural side channels. Plumber [199] is a framework that generates instruction sequences from templates to identify side-channel behavior, using concepts from instruction fuzzing, operand mutation, and statistical analysis. It was only implemented for ARM and RISC-V, but could be ported to x86. Scam-V [66] generates tests to validate side-channel models, based on validation of information flow properties using relational analysis.

Li and Gaudiot [253, 252], Depoix and Altmeyer [113], Ahmad [9], and Alam *et al.* [14] used a combination of hardware performance counters and machine-learning classifiers to detect Spectre and Meltdown attacks, and more broadly cache side-channel attacks, in live running hardware. CloudShield [183] used similar techniques to detect Spectre, Meltdown, and cache-based side-channel attacks on server hardware deployed in a cloud infrastructure. However, Dhavlle *et al.* [116] demonstrate that these detection mechanisms can be bypassed by variants of Spectre that use same-domain code-injection as part of the attack, and Pashrashid *et al.* [322] demonstrated they can be bypassed by Spectre variants that chain benign gadgets or insert `nop` instructions into the branch mistraining code.

Spectify [322] tracks the attack phases of a Spectre attack using microarchitecture-level information to find and report data leaks before the transmission phase of the attack, to help identify where hardware mitgations for Spectre need to be applied.

ABSynthe [169] takes an automated, black box approach to synthesizing contention-based side-channel attacks for x86 and ARM microarchitectures, which can be used by hardware designers for regression testing.

3.6.4 Software-only mitigation verification

Kasper [206] is a software scanner for the Linux Kernel that looks for code sequences that could be used as gadgets in the access phase of a Spectre-PHT attack, and models not only cache-based side channels, but also port-contention side channels, MDS-based side channels, and LVI. Kasper operates as a fuzzer on the syscall interface, and requires recompiling the kernel with support for the scanner. SpecFuzz [317] enhanced conventional fuzzing techniques with instrumentation to simulate speculative execution. FastSpec [430] used fuzzing and deep learning techniques to automatically generate and detect Spectre gadgets.

Mosier *et al.* [300] developed a static analysis tool for software based on their concept of a microarchitectural leakage containment model (LCM), which is able to identify some Spectre vulnerabilities. RelSE [109] performs static analysis of program binaries for Spectre-PHT and Spectre-STL, based on security property of speculative constant-time.

The CrossTalk [346] framework analyses the microarchitectural behavior of x86 instructions, with special attention to their use of globally shared staging buffers.

Easdon *et al.* [121] developed two open source frameworks—Transient Execution Attack

library (libtea) and SCFirefox—to generate prototype Meltdown, LVI, and MDS attacks on x86 and ARM.

Chapter 4

Discussion of heterogeneous multicores

In Chapter 3, we discussed the fundamental nature of Spectre-type vulnerabilities: 1) first the fault-injection phase mistrains a speculative predictor, 2) as a result of the successful fault-injection attack, the victim transiently stores a secret value in shared microarchitectural state, and 3) the side-channel leakage phase uses the shared microarchitectural state to convey the exposed secret to the attacker. Because a fault-injection attack on the speculative predictors is the first step of all Spectre-type vulnerabilities, only cores that have those predictors are vulnerable to Spectre. On a non-speculative core, speculative predictors cannot be mistrained, because there are no speculative predictors. On a non-speculative core, the secret value will never be transiently stored in shared microarchitectural state, so there is no way for any side-channel to access the secret value and convey it to the attacker.

This chapter explores combining speculative and non-speculative cores in a heterogeneous multicore system, as one possible approach to protecting security-critical workloads running on general-purpose server hardware. A workload running on a non-speculative core is not vulnerable to Spectre, because the core has no speculative predictors for the attacker to mistrain. Workloads on non-speculative cores never create transient shared microarchitectural state, so there is no transient state for the attacker to leak over side-channels, it simply never exists. Likewise, an untrusted workload running on a non-speculative core has a significantly restricted ability to launch Spectre-type attacks against other workloads—even other workloads running on speculative cores—because the non-speculative core never directly trains predictors.

The crucial insight of the approach is that targeting the fault-injection phase of Spectre—rather than the side-channel leakage phase—so thoroughly disrupts Spectre-type attacks that non-speculative cores are protected even though they share hardware resources with speculative cores on the same machine. Adding non-speculative cores to server hardware makes it possible to run limited sections of code non-speculatively—when the code is either critical to security or untrusted—while allowing most code to run speculatively for performance gains. With this approach, it is important to note that the speculative cores are not protected, and may still be mistrained in the fault-injection preparation phase—either by other speculative cores or in-place as a "confused deputy". Once a speculative core has been tricked into transiently updating shared microarchitectural state, any sensitive data in that shared microarchitectural state is exposed to non-speculative cores too. Heterogeneous multicore architectures cannot completely eliminate all risk from

speculation like the non-speculative approaches discussed in Chapter 5, but they do limit the scope and impact of the vulnerabilities more effectively and with more reasonable performance than other common mitigations today. The approach can also be strengthened by other known techniques to improve isolation between cores, such as partitioning shared resources like the last-level cache (LLC).

The interesting research question is not whether heterogeneous multicore architectures are possible, we know they are possible and already shipping in production. Nor is it interesting to ask whether non-speculative cores are an effective way to eliminate Spectre type attacks, we know that they are—as one author expressed it, processors with no speculation are "trivially immune to speculative execution attacks" [463, p. 11]. The interesting research question is whether heterogeneous multicores are a feasible approach for server hardware vendors to adopt, as a practical countermeasure against Spectre-type vulnerabilities and other speculation-based transient execution vulnerabilities. Chapters 5 and 6 explore more experimental approaches to eliminating speculation, but heterogeneous multicore architectures are a little easier to understand, so we will start there and build up to more complex approaches in later chapters.

4.1 Feasibility considerations

Heterogeneous multicore systems have been an area of increasing interest in systems research, as well as increasing plausibility in practical applications, since the early 2000s. While earlier work tended to focus on the performance advantages or energy efficiency advantages of the approach, more recent work has brought attention to the security advantages. As early as 2003, Kumar *et al.* [237] simulated a model combining cores with the same instruction set architecture (ISA) but different microarchitectures—some in-order and some out-of-order—on a single heterogeneous multicore chip, dynamically migrating a thread between cores to improve performance (on a more powerful core) or energy efficiency (on a less powerful core). In 2010, Li *et al.* [256] modified the Linux Kernel to simultaneously support sets of cores with different performance characteristics and slightly different instruction set architectures (ISAs), dynamically migrating threads between different kinds of cores to improve performance and throughput. In 2014, Aminot *et al.* [19] explored combining cores with different ISAs, some with a minimal set of instructions and others with added instruction extensions for less frequently used and more energy hungry features, dynamically migrating code between cores to improve energy efficiency. In 2017, Birhanu *et al.* [53] built on the ARM big.LITTLE architecture—a heterogeneous multicore architecture combining cores with the same ISA but different power and performance characteristics—and demonstrated that replacing the Linux Kernel's Completely Fair Scheduler (CFS) with their Fastest-Thread-Fastest-Core (FTFC) scheduler, which dynamically migrated threads between "big" and "little" cores based on CPU utilization and capacity, improved power efficiency by 2.22% and improved performance by 52.62%. In 2019, the mainline Linux Kernel accepted the Energy Aware Scheduler (EAS) [216], specifically for heterogeneous multicore architectures like ARM big.LITTLE, which similarly takes CPU utilization and capacity into account to improve energy efficiency while minimizing negative impact on performance. In 2020, Ainsworth and Jones [13] proposed adding a set of non-speculative special-purpose cores with different ISAs running specialized kernels, alongside the main speculative general-purpose cores, to improve the security of small regions of code that are either offloaded to the non-speculative cores or

run as independent validation of results on the main cores. Also in 2020, Le *et al.* [245] briefly suggested a heterogeneous multicore system combining non-speculative Rocket cores [27] with speculative BOOM cores [81, 80] as a mitigation approach for transient execution vulnerabilities, without any implementation details. In 2022, Anzai *et al.* [25] proposed a heterogeneous architecture for security sandboxing, where kernel and user space run on separate cores that do not share memory or cache and can only communicate using remote direct memory access (RDMA).

4.1.1 Production hardware

Beyond academic research, several factors make the heterogeneous multicore approach worthwhile to consider as a possibility for the near-term future of mainstream server hardware. In mainstream production hardware, the ARM big.LITTLE architecture has been shipping for several years in smartphones such as Samsung and Apple, and in tablets and laptops such as Apple's M1 family of SoCs with their own heterogeneous ARM architecture. Intel has also been trying out a heterogeneous multicore approach for x86 architectures, with the Lakefield mobile processors and Alder Lake desktop processors. These architectures work well in a mobile devices or laptops—where peak performance is only required sporadically, cores are frequently idle, and switching low priority workloads to lower power cores can significantly extend battery life. It is less clear that heterogeneous multicores are desirable in server hardware especially for cloud or container deployments—which require consistent peak performance for all workloads, and aim to minimise idle cores by sharing the same hardware resources between many workloads for many tenants—as we will discuss further in Section 4.2.

4.1.2 Prototyping

The University of Berkeley's Chipyard framework [18] for design, simulation, and implementation of RISC-V SoCs—originally called the "Rocket Chip Generator" [27]—includes a collection of default configurations to build SoCs that combine varying numbers of heterogeneous cores, including non-speculative Rocket cores and speculative BOOM cores. Balkind *et al.* [31] developed the BYOC ("bring your own core") framework—extending the earlier OpenPiton framework [32]—explicitly for the purpose of supporting implementations of heterogeneous multicore systems.

Both Chipyard and BYOC provide open hardware implementations of common shared components, including memory with coherent caches, accelerators, and standard peripherals such as UART, block devices, and NICs. Both frameworks support software simulation, FPGA emulation, and tapeout to silicon as output targets. Chipyard is implemented as a parameterized hardware generator based on Chisel, with support for integrating Verilog components directly, while BYOC is implemented in Verilog and SystemVerilog. Both frameworks have a strong emphasis on enabling verification and validation of designs. Both frameworks support heterogeneous cores as general-purpose first-class citizens, but Chipyard focuses on cores running versions of the RISC-V ISA plus extensions, while BYOC includes implementations of cores using RISC-V, x86, and SPARC ISAs, and specifically targets combining different ISAs into unified general-purpose heterogeneous multicore systems.

In addition to Chipyard and BYOC, a number of other more specialized frameworks build custom multicore systems with varying degrees of heterogeneity. OpenPiton [32] was

originally focused exclusively on the SPARC ISA, but later added the RISC-V ISA in the form of the 64-bit Ariane core [33]. lowRISC [59] combines RISC-V cores with different ISA extensions, however the small cores only serve as subordinate "minions".

The existence of frameworks like Chipyard and BYOC mean that it is currently relatively straightforward to build custom heterogeneous multicore systems for development and testing, using either the same ISA with different microarchitectures or entirely different ISAs.

4.1.3 Kernel

The Linux Kernel already supports heterogeneous multicore systems like ARM's big.LITTLE architecture with the Energy Aware Scheduler [216] or Capacity Aware Scheduler [215]. A speculation-centric heterogeneous architecture could use those existing schedulers, with the non-speculative cores serving the purpose of the lower energy, less powerful "little" cores, and the speculative cores serving the purpose of the higher energy, more powerful "big" cores. However, the existing schedulers would only take into account CPU utilization, capacity, and energy usage, and would not give any consideration to the security implications of the differences between the big and little cores.

A new scheduler would be necessary to take full advantage of non-speculative cores in heterogeneous multicore systems. Rather than prioritizing energy efficiency or performance, such a scheduler would prioritize consistent allocation of tasks within an appropriate "security domain"—a group of CPUs with the same security characteristics—directly parallel to the "performance domains" that the Energy Aware Scheduler uses to group CPUs by performance characteristics [217]. A region of code that is not safe for speculation must only be scheduled on a non-speculative core. A region of code that is safe for speculation could always be scheduled on either a speculative or non-speculative core, so the processing resources of the non-speculative cores would still be useful for the purpose of energy efficiency, even when they are not being utilized for the purpose of security.

Developing a new scheduler for a heterogeneous multicore system—with a bifurcation of cores for security rather than performance—is not a trivial task, but the desired features are a relatively minor departure from existing schedulers, and not disruptive to the overall stack of systems software.

4.1.4 Workloads

One level up the system stack from the kernel, modern implementations of virtual machines and containers make use of kernel security features to improve their own security. Chapter 2 and Randal [351] reviewed the evolution of standard features in the Linux Kernel used by modern virtual machines and containers, such as filesystem, process, IPC, and network namespaces, resource usage limits, access controls, and system call filtering. These security features are applied at the process-level, where each virtual machine or container is a process on the host kernel, and may contain additional processes on the same host kernel or a guest kernel. To take full advantage of speculation-centric heterogeneous multicore systems, the container runtime or virtual machine manager need the ability to declare the security domain of the processes it launches as virtual machines or containers, so the kernel scheduler can appropriately choose a speculative or non-speculative core for the process. One simple way to integrate such a feature into the Linux Kernel would be to add a speculation capability (perhaps `CAP_SYS_SPEC`) that grants permission to run on a

speculative core. The Linux Kernel scheduler would then take the speculation capability into account when choosing where to schedule tasks.

Another alternative, which would not require any modification to the Linux Kernel scheduler or capabilities, would be to use the Linux Kernel's existing features for processor affinity and CPU pinning, to restrict a process or thread so that it will only run on a specific core or set of cores. The `taskset` command and `sched_setaffinity` system call set the CPU affinity of a process, while the glibc functions `pthread_setaffinity_np` and `pthread_attr_setaffinity_np` set the CPU affinity of a thread. The existing features for CPU affinity are more manual, and would require defining CPU sets for speculative and non-speculative cores. But, using existing features would make it easier to evaluate an unmodified Linux distribution on a heterogeneous multicore system generated by a default configuration of a framework like Chipyard.

4.2 Trade-offs

Heterogeneous multicore systems that combine both speculative and non-speculative cores make it possible to entirely disable speculation for security-critical or untrusted sections of code, by running that code on a non-speculative core. Code running on a speculative core performs as well as it would on an ordinary speculative hardware architecture.[1] The systems software developer has the power to choose which code runs with the performance advantage of speculation, and which code runs with the security advantage of no speculation. However, heterogeneous multicores only offer the ability to disable speculation at the process or thread level. A finer-grained approach is desirable, to limit the performance penalty of disabled speculation to the smallest possible region of code.

The greatest advantage of the heterogeneous multicore approach is that it requires less extensive and less disruptive changes to the hardware and software stack, which makes production hardware realistically achievable in the short-term future. By comparison, the manual approach to selective speculation in Chapter 6 requires substantial changes at the microarchitecture level and throughout the systems software stack. The non-speculative approaches in Chapter 5 and the automatic selective speculation approaches in Section 3.3.2.4 only require changes at the microarchitecture level, but developing those microarchitectures to production-ready silicon chips for large-scale server hardware will be a multi-year effort.

One significant disadvantage of the heterogenous multicore approach is the level of granularity in control over speculation. The manual selective speculation approaches in Chapter 6 have instruction-level granularity—speculation can be enabled and disabled at the level of a single instruction in the instruction stream. The heterogenous multicore approach has process-level or thread-level granularity—speculation can only be enabled and disabled at the process or thread level. So, an entire virtual machine or container—or a process or thread inside a virtual machine or container—may be speculative or non-speculative, but controlling speculation down to the level of a single instruction would be impossible. For some use cases the granularity of control may not be important, but as long as speculative pipelining continues to be substantially faster than non-speculative

[1] In light of the risks posed by the transient execution vulnerabilities, it is unlikely that any serious hardware architecture would ship a speculative core with absolutely no mitigations, but it could ship a speculative core with a moderate level of mitigations similar to what Intel, AMD, or ARM provide today—providing some combination of the most critical software, hardware, and configurable mitigations.

pipelining, there will be a performance advantage to keeping speculation enabled by default, and only disabling it for the smallest possible regions of security-critical code.

The greatest disadvantage of the heterogenous multicore approach in the context of large-scale server infrastructures is inflexible hardware resource allocation. When building a large-scale data center, the heterogenous multicore approach would require deciding in advance exactly how many speculative and non-speculative cores to manufacture in every machine in the data center. Getting the upfront allocation decision right involves an impossible level of precision in predicting the kinds of workloads that customers will want to run, and exactly what percentage of speculative versus non-speculative execution they will want. Getting the upfront allocation decision wrong means a massive waste of capital spent on cores that lie idle in production due to a lack of customer demand. A private cloud might get away with downgrading some workloads from big cores to small cores, but a public cloud provider selling CPU time in performance tiers by the hour would quickly lose customers if they arbitrarily downgraded workloads that the customer paid to deploy on big speculative cores, substituting unused small non-speculative cores.

While heterogeneous multicores are not a good choice for large-scale server hardware, they may be a reasonable approach to consider for mobile and laptop devices, which have less extreme requirements for flexible resource allocation. The disruption of shifting to heterogeneous multicores would be particularly minimal in products where the hardware is already running a combination of big and little cores for performance and energy consumption considerations, as in some modern smartphones and laptops.

Chapter 5

Discussion of eliminating speculation

Since the transient execution vulnerabilities were first reported in 2018, it has often been said that we must continue producing superscalar processors based on speculative execution, despite the security risks, because the speculation features are critical for performance [226, 263, 383, 491, 72, 164, 369, 182]. It is true that stripping away all the modern features of superscalar processors and stepping back in time to the exact features of old simple scalar processors would certainly mean returning to the performance levels of those scalar processors. However, the defining characteristics of modern superscalar processors are not speculative execution, they are dynamic multiple issue and dynamic pipeline scheduling [325, p. 328]. In light of the transient execution vulnerabilities discovered in recent years, it is a worthwhile exercise to consider the logical limits of performance for a modern superscalar processor without speculative execution features. In the context of large-scale server hardware, it is also worth considering whether maximizing instruction-level parallelism for a single stream of instructions—which has always been the performance goal of speculative execution—is still the right performance goal for systems that are massively multi-core, multi-threaded, multi-workload, and multi-user.

5.1 Feasibility considerations

There are many possible ways to implement the microarchitecture of any given instruction set architecture (ISA). This chapter explores the feasibility of a small but significant variation on existing microarchitecture implementations, evolving superscalar processor techniques forward while eliminating speculation. Chapters 4 and 6 explore alternative approaches that make it possible to disable speculation for specific regions of code, without eliminating it entirely.

5.1.1 Performance characteristics of speculation on server hardware

In the early 1970s, before speculative execution became the norm, Riseman and Foster [358] published a thought experiment on the theoretical limits of performance for conditional branches. This work posited a machine with an infinite reorder buffer,[1] infinite functional units, infinite registers with register renaming, and the ability to hold an infinite number of

[1]They did not call it a "reorder buffer", or settle on any consistent way to refer to it, but the most common were "instruction stack" or "dispatch stack".

"tentative computational paths" simultaneously for both code paths from each conditional branch (taken or not taken) and discard incorrect paths when the branch conditions were resolved. They observed that the maximum speed on such a machine is attained through the potential to hold an infinite number of conditional branches, which meant that the maximum possible performance improvement was limited by the size of the reorder buffer. They calculated that holding j conditional branches in the pipeline required a reorder buffer large enough to hold least 2^j simultaneous code paths. On the hardware of the time, they considered a reorder buffer with two slots to be a reasonable size, and ten slots as unusually large, so they rejected the approach as impractical.

This is the theoretical context in which speculative execution was embraced. At a fundamental level, speculative execution was not originally a performance optimization so much as it was a space optimization—using less storage space at various stages of the pipeline to achieve roughly the same performance. Speculation allows the pipeline to only hold the instructions for one path of each conditional branch at a time, while ignoring the other path that it predicts as unlikely. But, the space optimization comes with a heavy performance cost when the speculation turns out to be wrong and the instructions for the ignored path have to be fetched, decoded, issued, and executed after the result of the branch condition is known, as if the speculation never happened.

In 2003, Swanson *et al.* [414] observed that the performance benefit of speculation decreases on multithreaded superscalar processors as they increase parallelism through different elements of the pipeline. In their experiments, a non-speculative pipeline with 8 functional units (combined ALU/LSU) performed 12% better than a speculative pipeline with 4 functional units. Increasing the size of the L1 data and instruction caches decreased the performance benefit of speculation from 33% at 16KB, to 24% at 128KB. Increasing the number of threads on the core reduced the performance benefit of speculation from 300% at 1 thread, to 100% at 4 threads, 24% at 8 threads, and 0% at 16 threads. They observed that only heavily loaded servers with many workloads would keep such a machine continuously busy, and so dismissed their own results as not applicable to "the vast majority of cases" [414, p. 335]. However, their results are uniquely applicable to large-scale server hardware, and the performance impacts they observed are also more relevant to modern superscalar processors than they were 20 years ago.

In 2012, Ferdman *et al.* [137] analyzed common cloud workloads running on large-scale server hardware, and reached a similar conclusion to Swanson *et al.* [414] that for these kinds of workloads it is better—for performance, die area, and power costs—to reduce out-of-order/speculative features and instead increase the number of hardware threads.

In 2021, Thoma *et al.* [423] implemented an extension to the RISC-V ISA core, called BasicBlocker, to annotate the instruction stream with basic block information, so the pipeline can make efficient choices about out-of-order branch instruction delays, without speculation. They also implemented a modified version of the LLVM compiler to generate the basic block information for the instruction stream. On a VexRiscv implementation, BasicBlocker achieved a 2.12x performance gain, compared to a 2.88x performance gain for full control-flow speculation. On gem5, BasicBlocker achieved a 2.13x performance gain over no speculation, compared to a 3.69x performance gain for full control-flow speculation.[2]

In 2022, Schall *et al.* [377] and Asheim *et al.* [29] analyzed serverless workloads on

[2]Keep in mind that gem5 inaccurately models branch prediction in a way that skews results in favor of speculation [85, 492].

large-scale server hardware, which tend to have small memory footprints, short execution times, and a high degree of "lukewarm" invocations. These workloads do not get the full performance benefit of microarchitecture features designed to optimize for frequent repetition of closely-related code—such as speculative prediction, prefetching, and cache—because these predictive microarchitectural structures are thrashed by a large number of unrelated workloads and are never fully "warm". They observed an overall performance penalty of 31-114% for lukewarm invocations. Asheim *et al.* [28] also demonstrated that large-scale server hardware suffers from performance problems in branch prediction, because the large instruction footprints of modern workloads exceed the capacity of the server hardware's branch target buffer (BTB) and L1 instruction cache.

Performance measurements for transient execution mitigations are generally taken on idealized workloads like the SPEC CPU benchmarks, and run repeatedly to gain the maximum possible benefit from fully "warm" branch predictors and other predictive microarchitecture features. However, it is important to remember that these idealized performance results do not reflect the real-world performance of speculation, especially not on large-scale server hardware. In a deployment context where speculative cores have a performance penalty as high as 114%, it becomes far more reasonable to consider non-speculative cores in production server hardware.

5.1.2 Non-speculative branch instructions

One possible way to entirely avoid the security risk of speculative branch instructions is to replace them with non-speculative branch instructions, which do not participate in branch prediction. In a traditional, scalar, in-order microarchitecture without branch prediction, non-speculative branch instructions stall the pipeline, so the pipeline does not fetch or issue any instructions from either path of the branch until the branch condition or branch target is resolved. A number of more nuanced approaches to non-speculative branch instructions are possible in the context of dynamic pipeline scheduling. One example, not particularly new or radical, would be to design the system so that it fetches and issues instructions from both paths of a conditional branch instead of trying to predict which path of the branch will be taken or not—beyond the theoretical work by Riseman and Foster [358] in the 1970s, the IBM 370/168 and IBM 3033 used similar techniques [247], though the pipelining techniques of the time were a poor fit for fetching multiple independent instruction streams. Another more recent example is optimizations that replace branch prediction with predicated execution to improve performance [437].

5.1.2.1 Fetch

Instruction fetching for non-speculative branch instructions does not read from or update a branch target buffer, branch history buffer, branch history table, pattern history table, or return stack buffer. This means both that branch instructions in malicious code cannot mistrain a branch predictor, and that branch instructions in secure code are not vulnerable to mistraining attacks.

5.1.2.2 Issue

The non-speculative pipeline places instruction entries in the reorder buffer for branch instructions and other instructions in the usual way, but depending on the non-speculative

approach, it may take additional steps. The predicated execution approach would add a decoding step for a conditional branch instruction and the instructions that follow it, rewriting them as predicated micro-ops. The IBM 3033 approach would add entries in the reorder buffer for instructions on both the taken or non-taken code paths of a conditional branch, tagging the entries with a control dependency on the result of the conditional branch instruction, and renaming registers in each code path so the two alternative sets of instructions do not use the same physical registers.[3]

The pipeline would then send predicated micro-ops or the instructions tagged with control dependencies to the reservation stations.

5.1.2.3 Execute

The reservation stations buffer the instructions in the usual way, however they would treat a control dependency tag or predicated micro-op similarly to a data dependency of waiting for operands. No instruction (or micro-op) from a taken or not-taken branch code path is dispatched to a functional unit until the result of the non-speculative conditional branch instruction is known, so there is no "speculative execution" of any instruction. However, since the instructions have already been fetched and issued, the instructions on the taken branch path can be dispatched to the functional unit on the next clock cycle after the result of the conditional branch has been calculated. This is far more rapid than if the entire fetch and issue process was delayed until after the branch condition is evaluated. The instructions (or micro-ops) on the not-taken branch path are discarded by the reservation stations, and tagged for discard in the reorder buffer (the instruction entries are effectively re-written as completed no-op entries).

5.1.2.4 Commit

The commit unit treats instructions from the not-taken conditional branch path in the same way as incorrectly speculated instructions, freeing up the renamed registers and removing the instruction entry from the reorder buffer. The commit unit still processes the reorder buffer in the order the instructions were fetched. The instructions from either the taken or not-taken code path for the conditional branch might have been fetched first, but it ultimately does not matter branch code path is first in the reorder buffer, since the instructions from the not-taken branch path will all be discarded (in the order they were fetched), and the instructions from the taken branch path will all be committed (in the order they were fetched). No instructions on either branch code path will ever produce a result until after the result of the conditional branch is known.

5.1.2.5 Discussion

If we altered Figures 3.2, 3.3, and 3.4 for a non-speculative branch pipeline, the primary visible change would be removing the branch predictor components. The size of some components of the pipeline—such as the L1 instruction cache, fetch width, fetch buffer, decode width, reorder buffer, and the reservation stations—might to be increased to maintain the same instruction throughput despite discarding instructions on the not-taken code path or predicated micro-ops. On the whole, however, non-speculative superscalar

[3]This step is similar enough to the microarchitecture technique of loop unrolling that it might deserve to be called "branch unrolling".

branch pipelines preserve the microarchitectural state and functionality of the existing superscalar core design, requiring only minimal changes to the internal behavior of instruction fetch, issue, execute, and commit.

The approaches to non-speculative branch pipelines discussed in this section do more work than a speculative branch pipeline—such as decoding predicate micro-ops or fetching and issuing instructions from both branch paths, instead of fetching and issuing instructions from the predicted branch code path and ignoring the other branch code path. However, the extra work of the non-speculative pipeline does not necessarily imply slower throughput of instructions with dynamic pipeline scheduling, since predicated micro-ops or the instructions on the two branch code paths are independent, and can be fetched and issued in parallel. The execution of non-speculative branch instructions will always be faster than mispredicted speculative branch instructions, since the pipeline can immediately proceed with executing predicated micro-ops or the already issued instructions from the correct branch code path, instead of doing all the work of executing the misspeculated branch path instructions, discarding them, and then starting the fetch for the correct branch path instructions after the result of the conditional branch is known. So, it is likely that a non-speculative superscalar branch pipeline will tend to perform no better than the best case of all correctly predicted speculative branch instructions, but will tend to perform better than the worst case of all mispredicted speculative branch instructions. In practical terms, the relative performance of a non-speculative branch pipeline compared to a speculative branch pipeline will also depend on the actual nature of the code being run—specifically on whether the code is dominated by branch conditions and targets that are consistently predictable or change results frequently, on whether branches have long or short sequences of instructions on their code paths, and on how extensive the data dependency of surrounding instructions is on the results of branch instructions and the results of other instructions with a control-flow dependency on those branch instructions.

In Chapter 7, we will explore the performance potential of non-speculative branch pipelines further with RTL and FPGA simulation.

5.1.3 Non-speculative memory load instructions

One possible way to entirely avoid the security risk of speculative memory load instructions is to replace them with non-speculative memory load instructions. However, while the limiting factor for branches is parallelism in the pipeline itself, the limiting factor for memory loads is unavoidable memory latency through the data cache hierarchy and DRAM, which means that the performance impact of eliminating speculation will be greater for memory loads than for branches.

5.1.3.1 Fetch

Instruction fetching for non-speculative memory load instructions does not use speculative predictions from the memory dependence predictor, and does not participate in training the memory dependence predictor for any future speculative predictions. It does, however, use the memory dependence predictor for tracking when all prior stores to the same address have been completed—resolving all Store To Load (STL) dependencies for the memory load instruction—to determine when the memory load is safe to execute non-speculatively.

5.1.3.2 Issue

Non-speculative memory load instructions are fetched, issued with an entry in the reorder buffer, and dispatched to the reservation stations, the same as in Section 5.1.2.2. Memory load instructions are tagged with a control dependency by the memory dependence predictor, so they cannot execute before all prior stores to the same address have been completed. If a memory load instruction is fetched and issued as part of a non-speculative branch code path, it may also be decoded as a predicated micro-op or tagged with a control dependency on the result of the branch instruction.

5.1.3.3 Execute

The reservation stations preserve control dependency tags or predicated micro-ops, treating them similarly to the data dependencies of waiting for operands. They will not dispatch any memory load instruction to the load-store unit until all prior stores to the same address have been completed. When the memory load is on a conditional branch code path, the reservation stations also will not dispatch the memory load instruction to the load-store unit until the result of the branch condition or branch target is known.

5.1.3.4 Commit

Since non-speculative memory load instructions are never speculatively executed, the memory load instruction entry in the reorder buffer is always marked as complete after it receives a result from the load-store unit.

5.1.3.5 Discussion

Non-speculative memory loads will always be slower than the best case where a speculative memory load is correctly predicted through the memory dependence predictor. In the worst case of branch misprediction, non-speculative memory loads avoid the cost of loading a value through multiple layers of cache that will only be discarded, while still giving the pipeline flexibility to dynamically schedule the memory load instruction out-of-order (without executing it speculatively). In the worst case of the memory dependence predictor mispredicting, non-speculative memory loads avoid the cost of re-executing the memory load operation and any other instructions that depended on the value it loaded.

In the case where a non-speculative memory load is an L1 data cache hit, simple out-of-order execution without speculation may be able to hide any performance penalty of the memory load, because it can execute the memory load instruction as soon as any control dependencies on non-speculative branch instructions or prior stores are completed. These control dependencies may be resolved and the memory load executed long before any instructions with a data dependency on the memory load are ready to execute, even without speculation. In the case where the non-speculative memory load is an L1 data cache miss, however, the performance penalty of the memory load may be prohibitive. Zhao *et al.* [501] estimate that on the BOOM RISC-V microarchitecture, the performance penalty for an L1 fetch is only 10 cycles, while the performance penalty for an L3 fetch is on the order of 50 cycles, which would require a lookahead of 200 instructions on a 4-wide BOOM pipeline, exceeding the capacity they designed for the reorder buffer.

5.1.4 Thread-level parallelism

Modern hardware architectures like the x86 do not limit a single hardware core to running a single process, instead they use the abstraction of *threads* to share a core between multiple tasks. Multithreading is important in the context of desktop and mobile hardware—which tend to have a relatively small number of cores—but it is absolutely essential in the context of multitenant server architectures—where the cloud/container business model depends on the ability to overcommit CPU resources, running more (mostly inactive) virtual machines or containers than the machine has cores. Sharing cores yields a substantial performance gain, because idle compute resources from one thread can be used for another thread. Even combining all the most advanced techniques of instruction-level parallelism,[4] a single stream of instructions for a single thread will still regularly block on data hazards or control hazards when there are not enough instructions ready to be dispatched to a functional unit in a clock cycle to fill all the available functional units, leaving compute resources idle.

Unfortunately, the performance gain of multithreading compounds the security risk of speculative execution, because it means that the microarchitectural state exposed by branches and memory loads is shared between multiple threads, which may be running completely unrelated code from unrelated virtual machines or containers for unrelated users. Percival [330] provides an early but comprehensive exploration of the risks inherent in combining simultaneous multithreading with speculative superscalar pipelining, multilevel memory caches, and hardware prefetching. Ge *et al.* [152] more broadly survey the microarchitecural side-channel attacks enabled by multithreading, multilevel memory caches, and co-resident virtual machines. Escouteloup *et al.* [126] propose a collection of fundamental security principles and recommendations for the design of future hardware in light of these microarchitectural risks. Taram *et al.* [421] propose an adaptive partitioning approach to provide more complete isolation for shared hardware resources between threads, but in a temporary way that adapts as the resource needs of the threads change.

Existing techniques to improve the security of multithreading with speculative superscalar pipelines—such as tracking a unique thread ID for every instruction through all stages of the pipeline and segmenting caches and other microarchitectural state per-thread—are equally effective with non-speculative pipelines. On each clock cycle, multiple instructions from multiple different threads may be fetched, entered in the reorder buffer, issued to the reservation stations, dispatched to functional units, and marked as complete by the commit unit. Although they share the same microarchitecture hardware, instructions from one thread must not have access to another thread's microarchitectural state. Non-speculative superscalar pipelines have the potential to provide a stronger guarantee of the required isolation between threads than speculative pipelines, because they never create the lingering traces of mispeculated microarchitechural state that are exploited by the transient execution vulnerabilities.

5.2 Trade-offs

Non-speculative superscalar pipelining makes it possible to eliminate the speculation-based transient execution vulnerabilities, in a way that is invisible outside the microarchitecture. The approach does not require any changes to the ISA or system software, so it is not

[4]Such as dynamic multiple issue, dynamic pipline scheduling, hardware prefetching, speculative execution, etc.

disruptive to existing software stacks, and preserves portability between different processors with the same ISA, even when one processor has a speculative microarchitecture and another has a non-speculative microarchitecture.

From a security perspective, the approach in this chapter—a standard ISA with a non-speculative microarchitecture—has the advantage of entirely eliminating the risk of speculative execution, while the heterogeneous multicore approach in Chapter 4 or the selective speculation approach in Chapter 6 only provide the ability to partially disable speculation.

From a complexity and die-area perspective, the standard ISA approach in this chapter may have an advantage over the selective speculation ISA approach in Chapter 6 because it eliminates the complexity and die-area of speculation features in the pipeline, rather than keeping all the complexity of speculation features and adding additional complexity for features to disable speculation. The approach in this chapter also has a resource allocation advantage over the heterogeneous multicore approach in Chapter 4, because every core on the server is identical, so the host retains full flexibility to control scheduling between cores in response to demand, rather than restricting certain workloads to specialized non-speculative cores.

One potential disadvantage of the standard ISA approach in this chapter is that the microarchitecture is exclusively dedicated to non-speculative pipelining, and has no option to use speculation even in regions of non-critical code where it might be safe to speculate. For the immediate future, despite the security risks, speculation is a solid bet for improving performance, and as long as that continues to be true, it is worthwhile to explore approaches that only partially disable speculation, as in Chapters 4 and 6.

Chapter 6

Discussion of selective speculation

The microarchitectural features that make the transient execution vulnerabilities possible are integral to the performance potential of modern hardware architectures. Entirely eliminating speculative features from an architecture does eliminate the transient execution vulnerabilities, but it also degrades performance. Applying the known mitigations for the transient execution vulnerabilities also significantly degrades performance. This performance trade-off is a blunt instrument, all code running on the system is affected.

We question the fundamental assumption in current research and production hardware—and in decades of hardware architecture design—that speculative features must be always off or always on, and so mitigations must also apply in a universal fashion. Early systems like the Intel i860 [230] combined speculative and non-speculative features, and despite decades of evolution in a different direction, the combination is still possible in modern hardware architectures. The advantage of such a hardware architecture would be the ability for systems software to choose between parts of a host or guest operating system that are so sensitive they must not be speculated, and other parts where performance is crucial but leaking information is harmless.

This chapter explores adding instructions to the RISC-V instruction set architecture (ISA), making it possible to selectively disable speculative execution within a single core. The microarchitecture design options discussed in this chapter are part of a wider body of work on selective speculation techniques discussed in Section 3.3.2.4—called "selective" because they are speculative microarchitectures but also provide the option of running non-speculatively.

If we shift the paradigm slightly, recognizing that some regions of code are more sensitive to leaking transient microarchitecural state than others, it opens the door to different approaches. Systems software developers are familiar with making trade-offs between security and performance, and with the fact that different technical choices may be appropriate for different software contexts such as host kernels and operating systems, guest kernels and operating systems, application or workload software, and cryptographic software. Techniques to disable speculation entirely can improve the security of critical code, because they rupture the fundamental DNA of all Spectre-type attacks—blocking the initial attack vector that makes these attacks so much more severe than all previously known side-channel attacks, and preventing speculative microarchitectural state from ever being created in the first place—so they are proof against all currently known variants and all variants that may be discovered in the future. However, the performance penalty of disabling speculation for an entire machine (or entire data center) is high. A finer-grained approach to disabling techniques that narrows their use to a specific security

domain—such as, a region of code, process, thread, sandbox, trusted execution environment, VM, container, or privilege level—can reduce the system-wide performance penalty for large-scale deployments by limiting the use of the techniques to the smallest possible scope.

Similar to non-speculative cores in the heterogeneous multicore approach in Chapter 4, a region of code running non-speculatively on a selective speculation core does not create transient shared microarchitectural state, so there is no transient state for the attacker to leak over side channels. Likewise, a region of code running non-speculatively on a selective speculation core never directly trains predictors, so it has a significantly restricted ability to launch Spectre-type attacks against other regions of code. Unlike the non-speculative cores discussed in Chapter 5, selective speculation approaches do not completely eliminate all risk from speculation. It is especially important to note that speculative regions of code on a selective speculation core are not protected, and can still be mistrained in the fault-injection preparation phase, and tricked into transiently updating shared microarchitectural state in the access phase, so that any sensitive data in that shared microarchitectural state is exposed to non-speculative regions of code too. But, the ability to disable speculation for a region of code can offer more complete control over the risky effects of speculation than other mitigation approaches like speculation barriers or invisible speculation. When applied appropriately in the compiler and full system software stack, software-controlled selective speculation provides stronger security guarantees for critical regions of code than software-only mitigations can provide. The approach can also be strengthened by other techniques to improve isolation between security domains.

6.1 Feasibility considerations

In the late 1980s, the RISC-based Intel i860[1] included both speculative and non-speculative instructions, so the compiler had the power to choose whether to use a primitive form of speculation for branches and memory loads within a particular region of code [230]. The architecture was not commercially successful, so Intel abandoned it in the 1990s. However, in light of the transient execution vulnerabilities discovered in recent years, it is fascinating to consider where the industry might have ended up today if mainstream superscalar processors had taken the path of the i860 rather than the x86. We can never know what might have been, but we can apply a similar approach to modern hardware architectures and evaluate the results.

Some research into countermeasures for the transient execution vulnerabilities has taken the approach of implementing a traditional superscalar architecture with the standard RISC-V ISA and experimenting with minor variations in the microarchitecture. The Berkeley Out-of-Order Machine (BOOM) [81, 80] is a superscalar out-of-order RISC-V core, designed as a compatible substitute for the 5-stage in-order scalar pipelined RISC-V core Rocket [27], within the Chipyard [18] implementation framework for RISC-V custom SoCs. The third version of the BOOM core [501] offers more advanced speculation features, including an instruction fetch unit based on the TAGE branch predictor algorithm and a load-store unit that supports multiple loads per cycle. Gonzalez *et al.* [164] replicated the Spectre [226] bounds check bypass and branch target injection attacks on an extended RISC-V BOOM processor, and experimented with adding an L0 speculation buffer to mitigate the attacks, with only partial success.

[1] Also known as the 80860.

Some researchers have also proposed minor variations on the RISC-V ISA. Yu *et al.* [494] prototyped a RISC-V extension on the BOOM core, to track confidentiality labels on data against security-guarantees of instructions, as a mitigation for microarchitecural side-channel attacks. Escouteloup *et al.* [126] proposed an extension to the RISC-V ISA (specifically the RV32I base ISA) introducing a concept of confidential registers and hardware security contexts to express security boundaries, as a mitigation for some side-channel attacks. Wistoff *et al.* [477] proposed adding a fence instruction to the RISC-V Ariane core, to flush microarchitectural state when switching between security contexts, based on earlier research [152, 184, 154, 153] into time protection mechanisms against side-channel attacks.

6.1.1 RISC-V ISA extensions

The ideas discussed in this chapter are applicable to other hardware architectures, such as x86 and ARM. But, the modular nature of the RISC-V ISA—consisting of a base set of integer instructions and a collection of standard and non-standard instruction extensions for more complex features—does make it a particularly good target for experimenting with combining speculative and non-speculative instructions in a single core.

Within the RISC-V 32-bit base integer instruction set (RV32I), only a relatively small number of instructions are relevant for speculative execution: load operations access memory, and branch or return instructions create decision points in the control flow. An ISA extension to support both speculative and non-speculative features would add a small number of duplicate instructions, so each speculative instruction would have a non-speculative variant. Following the RISC-V naming convention specified for non-standard extensions, we call this hypothetical extension "Xnospec". Table 6.1 lists the relevant RV32I base instructions and their variants in the extension.[2] Throughout this chapter, we will describe the base instructions as speculative and the Xnospec extension as non-speculative, for example the `beq` (branch if equal) instruction is speculative, while the `xbeq` instruction in the Xnospec extension is non-speculative. The concept could be implemented equally well with non-speculative base instructions, but a speculative base means that unmodified compilers get the performance advantages of the speculative instructions, while modified compilers can access the security benefits of disabling speculation for limited sections of code.

Memory store operations (`sb`, `sh`, `sw`, and `sd`) participate in speculative execution, but only to the extent that they must wait until any speculative results they depend on have been finally committed. Since the Xnospec extension makes it possible to mix speculative and non-speculative instructions, it would not be safe to provide the user with alternative store instructions that ignore speculation, since the user might incorrectly use the non-speculative instruction to store results that actually depend on some speculatively executed code. Instead, the microarchitecture implementation of the store instructions must be modified to recognize that the Xnospec load and branch instructions do not participate in speculative execution, which effectively means they always commit immediately after the execution stage.

In RISC-V, computational instructions—such as math or logic operations—only operate

[2]There is no particular significance in the use of "X" as the first letter in the names of the variant instructions, it was chosen merely because the character is rarely used, relatively distinctive, and akin to the "X" prefix for non-standard extensions.

Table 6.1: Selective speculation extensions for RISC-V 32-bit base integer instruction set

Description	**RV32I Base**	**RV32I_Xnospec**
Load Byte (8-bit)	`LB rd,rs,imm`	`XLB rd,rs,imm`
Load Halfword (16-bit)	`LH rd,rs,imm`	`XLH rd,rs,imm`
Load Word (32-bit)	`LW rd,rs,imm`	`XLW rd,rs,imm`
Load Byte Unsigned	`LBU rd,rs,imm`	`XLBU rd,rs,imm`
Load Halfword Unsigned	`LHU rd,rs,imm`	`XLHU rd,rs,imm`
Branch Equal	`BEQ rs1,rs2,offset`	`XBEQ rs1,rs2,offset`
Branch Not Equal	`BNE rs1,rs2,offset`	`XBNE rs1,rs2,offset`
Branch Less-Than	`BLT rs1,rs2,offset`	`XBLT rs1,rs2,offset`
Branch Greater-Than or Equal	`BGE rs1,rs2,offset`	`XBGE rs1,rs2,offset`
Branch Less-Than Unsigned	`BLTU rs1,rs2,offset`	`XBLTU rs1,rs2,offset`
Branch Greater-Than or Equal Unsigned	`BGEU rs1,rs2,offset`	`XBGEU rs1,rs2,offset`
Direct branch ("jump and link")	`JAL rd,offset`	`XJAL rd,offset`
Direct branch (pseudoinstruction for `JAL`)	`J offset`	`XJ offset`
Indirect branch ("jump and link register")	`JALR rd,offset(rs1)`	`XJALR rd,offset(rs1)`
Indirect branch (pseudoinstruction for `JALR`)	`JR rs1`	`XJR rs1`
Return (pseudoinstruction for `JALR`)	`RET`	`XRET`

on registers and immediates, so while they might benefit from a speculative memory fetch or might run as part of a speculative branch prediction, they are inherently neutral to speculation, and do not require variants in the Xnospec extension.[3]

The RISC-V 64-bit base integer instruction set (RV64I) adds two instructions that are relevant to speculation. Table 6.2 shows variants for these instructions in the Xnospec extension.

Table 6.2: Selective speculation extensions for RISC-V 64-bit base integer instruction set

Description	**RV64I Base**	**RV64I_Xnospec**
Load Doubleword (64-bit)	`LD rd,rs,imm`	`XLD rd,rs,imm`
Load Word Unsigned	`LWU rd,rs,imm`	`XLWU rd,rs,imm`

Beyond the base integer instruction sets, RISC-V defines a combination of standard extensions for general-purpose computing, including multiplication and division instructions (extension M), atomic instructions (A), single-precision (F) and double-precision (D) floating point instructions, control and status register instructions (Zicsr), and the instruction-fetch fence instruction (Zifencei). This combination of integer base and standard extensions is abbreviated from "IMAFDZicsr_Zifencei" to simply "G", so the 32-bit and 64-bit general-purpose combined instruction sets can be clearly referred to as RV32G and RV64G.

RV64G is the typical target for RISC-V hardware capable of running a full Linux operating system, so it is reasonable for the Xnospec extension to consider the full set of RV64G instructions. The multiplication and division extension (M) and floating point

[3]It is worth noting that the x86 instruction set defines computational instructions that operate directly on memory locations, so a similar extension for x86 would require many more instruction variants than RISC-V.

extensions (F and D) do not add any instructions relevant to speculation, and so do not require variants in the Xnospec extension. The atomic extension (A) adds load and store operations for memory, however the instructions read, modify, and write memory within a single operation (for syncronization between multiple RISC-V hardware threads), and so will always be non-speculative. The control and status register extension (Zicsr) adds loads and stores of the Control/Status Register (CSR) set, however CSR instructions are also only executed non-speculatively. The instruction-fetch fence extension (Zifencei) may require changes at the microarchitecture level to handle the combination of speculative and non-speculative instructions, but does not require the addition of any variant instructions for the Xnospec extension.

In summary, the Xnospec extension adds a total of only 18 new instructions to the RV64G general-purpose instruction set. This increase in the footprint of the ISA is tolerably small, when weighed against the benefit of providing user-level control over speculative execution.

6.1.2 Microarchitecture

One crucial challenge for implementing selective speculation features at the instruction level lies in the microarchitecture, specifically in implementing a pipeline capable of efficiently executing both speculative and non-speculative instructions. As in Chapter 5, consider a foundation of a superscalar microarchitecture that is roughly analogous to a modern x86 processor, which uses dynamic multiple issue and dynamic pipeline scheduling, with out-of-order execution. There are many possible ways to implement the microarchitecture of both speculative and non-speculative features, but some are more compatible than others. Specifically, combining a modern superscalar microarchitecture with an old scalar microarchitecture on a single core would effectively require including two completely different instruction pipelines each with their own microarchitectural state. On the other hand, the non-speculative superscalar microarchitectures described in Chapter 5 is highly compatible with a speculative superscalar microarchitecture—they use identical instruction pipelines and nearly identical microarchitectural state, with only minor differences in which instructions are fetched, whether instructions are decoded as predicate micro-ops, and when instructions are executed. The next two sections describe a combination of speculative and non-speculative pipelining as one reasonable way to implement a microarchitecture supporting the Xnospec extension.

6.1.2.1 Branch instructions

In the selective speculation ISA, ordinary memory load, branch, and return instructions are speculative, while the non-speculative instructions in the Xnospec extension do not participate in memory dependence prediction, branch prediction, or return prediction.

Fetch: Instruction fetching for the non-speculative branch and return instructions does not read from or update the branch target buffer, branch history buffer, branch history table, pattern history table, or return stack buffer, even though the microarchitecture has these features in the hardware and uses them for the speculative branch instructions. This means both that non-speculative branch and return instructions will never use mistrained predictions from other speculative branch or return instructions (so they might be used in regions of code that are critical to security), and also that non-speculative branch and

return instructions cannot be used to mistrain branch or return predictors (so they might be substituted for speculative branch or return instructions in regions of code that are untrusted).

Issue: As in Section 5.1.2.2, the pipeline would place entries in the reorder buffer instructions would fetch instructions in the non-speculative extension as usual, though it might take some additional steps to tag entries with control dependencies or decode predicate micro-ops. Because the reorder buffer entries for non-speculative branch instructions are tagged with a special control dependency or decoded as predicate micro-ops, they do not interfere with ordinary speculative branch instructions, which proceed through the pipeline in the normal speculative way. The pipeline sends instructions to the reservation stations in the usual way, but non-speculative instruction sequences will either be marked with a control dependency tag or decoded as predicated micro-ops.

Execute: The reservation stations buffer all instructions issued from either speculative or non-speculative instruction sequences in the usual way, however they treat a control dependency tag or predicated micro-ops as similar to a data dependency of waiting for operands. Instructions from a speculated branch path are dispatched to a functional unit to be executed immediately. However, no instructions from non-speculative instruction sequences are dispatched to a functional unit until the result of the non-speculative memory load, branch, or return instruction is known, so there is no speculative execution of any instruction for the non-speculative instruction sequences.

Commit: The commit unit treats the instructions from the non-speculative sequences in the same way speculative sequences. For incorrectly speculated instructions or non-speculative instruction sequences on not-taken branch code paths, it frees up the renamed registers and removes the instruction entries from the reorder buffer. As in Sections 3.2.1.4 and 5.1.2.4, the commit unit processes results from the reorder buffer for all instructions—speculative or non-speculative—in the order the instructions were fetched.

As discussed in Section 5.1.2, the size of several fetch and issue components need to be increased in this selective speculation design to maintain the same instruction throughput despite discarding instructions on non-speculative branch paths.

6.1.2.2 Memory load instructions

Continuing to consider a superscalar microarchitecture that is roughly analogous to a modern x86 processor, the this selective speculation RISC-V microarchitecture may speculatively execute ordinary memory load instructions. Ordinary memory load instructions within a non-speculative branch path behave like any other instruction to the extent that they will be fetched and issued, but will not be speculatively executed as part of the branch. However ordinary memory loads could still be set up for speculative execution by the memory dependence predictor. There is a viable use case for permitting the memory dependence predictor to speculate memory loads within a non-speculative branch path,[4]

[4]As usual, this is primarily a trade-off between security and performance.

so in this hypothetical selective speculation microarchitecture only the non-speculative memory load instructions in the Xnospec extension fully avoid speculative execution.

Fetch: Instruction fetching for speculative memory load instructions uses speculative predictions from the memory dependence predictor, and participates in training the memory dependence predictor for future speculative predictions. Non-speculative memory load instructions do not use the memory dependence predictor for predictions, however, they do use the memory dependence predictor for tracking when all prior stores to the same address have been completed—resolving all Store To Load (STL) dependencies for the memory load instruction—to determine when the memory load is ready to execute non-speculatively.

Issue: Both speculative and non-speculative memory load instructions are fetched, issued with an entry in the reorder buffer, and dispatched to the reservation stations, the same as in Sections 3.2.2.2 and 5.1.3.2. Speculative memory load instructions proceed through the reorder buffer to the reservation stations in the normal speculative way. Non-speculative memory load instructions that are fetched and issued as part of a non-speculative branch path are tagged with a control dependency on the result of the non-speculative conditional branch instruction. Non-speculative memory load instructions outside of a non-speculative branch path are tagged with a control dependency by the memory dependence predictor, so they cannot execute before all prior stores to the same address have been completed.

Execute: The reservation stations preserve the control dependency tag or predicated micro-op for non-speculative memory load instructions, treating it similarly to a data dependency of waiting for operands. They will not dispatch a non-speculative memory load instruction to the load-store unit until all prior stores to the same address have been completed and/or the result of the non-speculative branch or return is known. Speculative memory load instructions proceed through the reservation stations, through the functional units, and on to the commit unit in the normal speculative way.

Commit: Since non-speculative memory load instructions are never speculatively executed, the commit unit always marks the memory load instruction entry in the reorder buffer as complete after it receives a result from the load-store unit. For speculative memory load instructions, the commit unit determines whether the speculated memory load instruction was speculated correctly, and if so, marks the instruction entry in the reorder buffer as complete, performs any pending register writes or memory stores, and removes the instruction entry from the reorder buffer. If the commit unit determines the speculation was incorrect, it will discard the result of the speculated memory load, and either remove the memory load instruction entry from the reorder buffer (if the instruction was on a misspeculated branch path), or else execute the memory load all over again together with any instructions that depended on its result (if the instruction was misspeculated by the memory dependence predictor).

6.1.3 High-level language modifications

If selective speculation features were implemented at the instruction level, the challenge for high-level languages would be how to expose the concept of choosing between speculative and non-speculative instructions, in a way that is meaningful to programmers and relatively easy to use. An extended ISA would require modifying the compiler to output the added instructions. Such a change involves modifying the definition of the source high-level language, adding a way for programmers to indicate which sections of code should be non-speculative, modifying the parser to recognize the new syntax, modifying the semantic analysis phase to retain information about regions of code tagged as non-speculative, and modifying the generator to select different instructions within non-speculative regions of code. There are many different ways such changes could be implemented in a compiler, this section and Section 6.1.4 explore one possible example using Rust as the high-level language and LLVM as the compiler toolchain, to demonstrate the feasibility of the approach.

In Rust, the `unsafe` keyword instructs the Rust compiler to alter a few small but significant low-level behaviors related to memory safety. Without delving into the details of how `unsafe` works in Rust, what is relevant here is that Rust programmers have become familiar with the concept of `unsafe` as a strategically placed keyword that alters the output of the compiler.[5] The `unsafe` keyword is allowed in three positions in the syntax of Rust: as a block, on a function or method definition, and on a trait declaration or implementation. We posit a new keyword named `nospec`, allowed in all the same syntactic positions as the `unsafe` keyword.

When used as a block, the `nospec` keyword indicates that all code within the body of the block should be compiled using non-speculative instructions:

```
nospec {
    // non-speculative code here
}
```

Conditional branches within the `nospec` block will not run speculatively, which means no code that depends on the branch instruction will execute until the condition is evaluated, no trace will be left in the cache of falsely predicted results, and no trace of the branch will be left in the branch predictor, so the code cannot influence any future branch predictions. Memory accesses within the `nospec` block will not perform speculative memory fetching, which means they will leave no trace of misspeculated loads in the cache to be exposed by side-channel attacks.

When used on a function or method definition, the `nospec` keyword indicates that all code within the body of the function should be compiled using non-speculative instructions. A function defined as a `nospec` function can only be called from within a `nospec` block, as a way of requiring the programmer to explicitly take responsibility for the altered behavior. In the code example below, the `leave_no_trace` function is defined as non-speculative, and called from within a `nospec` block[6]:

[5] Other programming languages have keywords to alter compiler behavior, such as `volatile` in C, C++, C#, and Java, but Rust's `unsafe` is a cleaner example of high-level language syntax for clearly delineated blocks of code.

[6] The Rust language only allows `unsafe` functions to be called from within an `unsafe` block, as a way of ensuring that the programmer is always explicitly aware when low-level behavior has been changed. The restriction is not absolutely necessary, but from a programming language design perspective it fits with Rust's strong emphasis on compile-time enforcement of memory safety.

```
nospec fn leave_no_trace() {}

nospec {
    leave_no_trace();
}
```

When used on a trait definition or implementation, the `nospec` keyword indicates that all code within the body of the trait should be compiled using non-speculative instructions. In the code example below, the `LeaveNoTrace` trait is defined as non-speculative, and later implemented for the `Password` type:

```
nospec trait LeaveNoTrace {
    // method signatures
}

nospec impl LeaveNoTrace for Password {
    // method implementations
}
```

6.1.4 Compiler toolchain modifications

Taking a step back from the high-level language to the compiler toolchain that supports it, the challenge at this layer is how to capture and preserve information about the `nospec` feature through all compilation phases, in order to finally output non-speculative instructions.

LLVM is a collection of libraries and tools, used for both static and dynamic compilation of programming languages. The compiler for the Rust programming language uses LLVM for code generation: one of the later stages of Rust compilation produces output in LLVM Intermediate Representation (IR)—a kind of heavily annotated assembly language—which LLVM takes as input to run optimization passes and generate machine code for the target architecture as the final output. LLVM is designed to be extensible and allow for custom behavior at every stage of the compilation process, including code generation for a wide variety of hardware architectures and even non-traditional code generation targets such as WebAssembly.

At the highest level of LLVM IR produced by the Rust compiler, LLVM has a built-in system for attaching metadata to annotate the IR instruction stream with additional information. The most common use of the metadata feature in LLVM is the `!dbg` identifier to capture source-level debug information in a standard form, and preserve that information through any optimization or tranformation passes, all the way through to the final generated machine code. It would be possible to add a `!nospec` metadata identifier to LLVM, and then modify the Rust compiler to annotate IR instructions, functions, and modules with the metadata identifier, corresponding to the high-level language code blocks, functions, and traits defined with the `nospec` keyword. The following code example in LLVM IR is a conditional branch, which branches to the destination label `true` when the condition evaluates as true, and to the label `false` when the condition evaluates as false. Normally, LLVM would compile this conditional branch as a speculative branch instruction in the standard ISA, but adding the metadata identifier `!nospec` to this conditional branch would tell LLVM to compile it as a non-speculative branch instruction in the extended ISA.

```
br i1 %cond, label %true, label %false, !nospec !0
```

At the lowest level of LLVM's code generation, the extended selective speculation RISC-V ISA would need to be defined as a separate target, though it would inherit almost all features from the existing standard RISC-V target. Each instruction added in the selective speculation RISC-V extension in Section 6.1.1 would require adding an entry in the `TargetInstrInfo` class for the target to describe the instruction.

Optimization or transformation passes in LLVM discard any metadata annotation they are unable to recognize, so each pass to be used in the compilation of the LLVM IR produced by the Rust compiler would need to be modified to preserve the custom `!nospec` metadata identifier. The instruction selection pass would also need to be modified, to recognize the metadata identifier, and use it to select the extended RISC-V non-speculative instructions instead of the standard RISC-V instructions.

Overall, the modifications to LLVM required to support an extended RISC-V ISA are not trivial, but they do lie within the realm of custom compiler features that LLVM was designed to support.

6.2 Trade-offs

While the ability to disable speculation for small regions of code is a security advantage over speculating all code, implementing selective speculative features at the ISA level is not radically more secure than the heterogeneous muilticore alternative outlined in Chapter 4 and is less secure than entirely eliminating speculation with the standard ISA approach in Chapter 5. Taking full advantage of the extended RISC-V ISA requires a web of changes through multiple layers of systems software. Such changes are feasible, but also disruptive, in a way that may initially make it more difficult to validate the security of the overall system.

From a performance perspective, the selective speculative ISA approach has an advantage over the heterogeneous multicore approach in Chapter 4, in that it avoids the overhead of inter-process communication between speculative and non-speculative regions of code. However, this advantage is balanced against the potential performance loss of a more complex microarchitecture pipeline combining speculative and non-speculative features.

From a portability perspective for large-scale server infrastructures such as cloud and containers, the disadvantage of the selective speculation ISA approach is that host and guest operating systems and workloads would be highly dependent on the low-level details of the extended ISA. A host operating system or guest image compiled for a selective speculation RISC-V ISA on one server is not portable to a standard RISC-V ISA on another server. On the other hand, the selective speculation ISA approach also has an advantage in resource allocation over the heterogeneous multicore approach—because every core is identical and able to run in a non-speculative mode, the host is free to allocate workloads to any core. With the heterogeneous multicore approach in Chapter 4, the host must allocate speculative workloads to speculative cores, and non-speculative workloads to non-speculative cores. Since the hardware for a heterogeneous multicore system is manufactured with a fixed number of speculative and non-speculative cores, there is no flexibility to change that allocation in response to demand. Across a data center with thousands or hundreds of thousands of servers, an imbalance in the utilization

of the speculative and non-speculative cores could result in a substantial performance loss through unused resources and wasted capacity.

Ultimately, the heterogenous multicore approach is easier to deliver in the short-term future, but selective speculation and entirely non-speculative cores are worth further research in the longer-term.

Chapter 7

RISC-V prototypes

The performance penalties of the Spectre mitigations available at the time we submitted this dissertation were prohibitive—in the range of 50% to 200% to fully mitigate the vulnerabilities—so a minimal amount of microarchitecture design and prototyping work was necessary to demonstrate that other approaches were possible. Research on the transient execution vulnerabilities continues to move at a rapid pace, and subsequent work by other researchers discussed in Section 3.3.2.4 both confirmed our hypothesis that selective speculation is the best approach for both security and performance, and also superseded the lightweight prototypes we developed with far more advanced microarchitecture designs built on the same concepts. We do not claim any credit for the subsequent work, it was a case of multiple independent researchers reaching similar conclusions through separate streams of experimental work.

In 2021 we implemented several lightweight SoC prototypes with RISC-V cores, in support of the feasibility discussions in Chapters 4-6. These prototypes were not intended as proposals of complete microarchitecture implementations, they were only intended as design space exploration of the potential performance impact of heterogenous multicores, non-speculative cores, and selective speculation cores. The purpose of the prototype work was to answer several key research questions:

- What is the performance penalty of eliminating speculation entirely, and how does that compare with the performance of mitigating Spectre?
- Are there alternative ways to improve performance, either with no speculation, or with restricted speculation?
- Can restricted speculation provide adequate protection against Spectre?

We simulated the prototypes and reference Rocket and BOOM cores with FireSim[212] and FireMarshal[329] on AWS F1 FPGAs, using the SPEC CPU2017 intspeed benchmarks.[1] The FireSim simulations ran at a host frequency of 65MHz on the FPGAs, and modeled the system running at 1GHz, configured with 512KB L2, 4MB simulated L3, and 16GB DRAM, and with 32KB L1I, 32KB L1D on the BOOM core and prototypes based on BOOM, but 16KB L1I, 16KB L1D on the Rocket core. The SPEC benchmarks were compiled with `gcc`, with `-O3` optimizations.

Figure 7.1 shows the combined results of the prototypes. The Rocket [27] core is an extensible RISC-V in-order scalar core, which uses branch prediction in the fetch

[1] So far, only the intspeed benchmarks have been successfully ported to run as a FireMarshal workload.

stage, but does not execute instructions speculatively. The BOOM [501] core is a RISC-V out-of-order speculative core, based on the Rocket core, with a high-performance TAGE [390] branch predictor. The first prototype we implemented was a heterogeneous multicore configuration, as discussed in Chapter 4, combining a non-speculative Rocket core with a speculative BOOM core. The performance of the Rocket and BOOM cores was identical in either single core or heterogeneous multicore configurations, so the results only show one entry each for Rocket and BOOM. The second prototype explored non-speculative cores, as discussed in Chapter 5, using a modified version of the BOOM core which eliminates the microarchitectural implementation of branch prediction from the Chisel source code for the core. The performance results for the non-speculative prototype appear in Figure 7.1 with the "Non-spec" label. And, the third prototype explored selective speculation, as discussed in Chapter 6, again using a modified version of the BOOM core. Instead of entirely removing branch prediction from the implementation, the selective speculation prototype work was a series of small variations, ranging from almost entirely speculative to almost entirely non-speculative. Figure 7.1 shows two of those variations with the "Mostly non-spec" and "Mostly spec" labels.

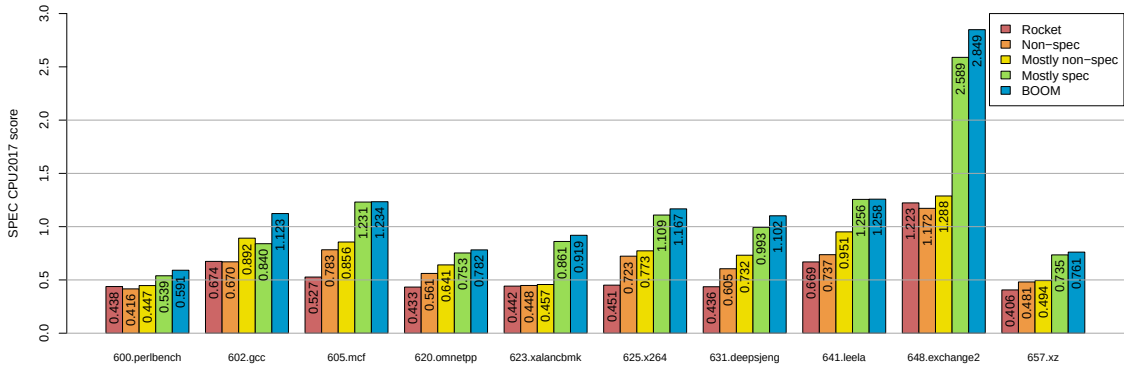

Figure 7.1: Comparing RISC-V prototypes on SPEC CPU 2017 benchmarks.

7.1 Baseline comparison of reference cores

As a baseline on the research question about the performance penalty of eliminating speculation entirely, the first performance comparisons we made were between two reference cores that already existed: Rocket (non-speculative) and BOOM (speculative). For this measurement, we made no modifications to either Rocket or BOOM, we only compared them in their default single-core configurations as provided by the original developers. Figure 7.1 shows the performance results for the Rocket and BOOM cores. On average, the Rocket core performed with a 49% performance penalty compared to the BOOM core.

Independent work by Thoma *et al.* [423], also in 2021, demonstrated a performance penalty of only 26% for a non-speculative superscalar core designed as a countermeasure for the speculation-based transient execution vulnerabilities.

These two non-speculative reference cores were shown in Chapter 3, Figure 3.5 as the two data points with mitigation targets of "all variants".

7.2 Heterogeneous multicores

We also implemented a heterogeneous multicore prototype as a baseline for comparison to other approaches. Prior work by other researchers was not adequate for the comparisons we wanted to make, either because the small non-speculative cores they implemented were too specialized [13], or because they only discussed the possibility of speculative/non-speculative heterogeneous multicores without actually implementing them [245]. The performance results for heterogeneous multicores were what one would reasonably expect, they demonstrated that workloads allocated to a big core in a heterogeneous configuration perform the same as on a big core in a single-core configuration (the "BOOM" label in Figure 7.1), and workloads allocated to a small core in a heterogeneous configuration perform the same as on a small core in a single-core configuration (the "Rocket" label in Figure 7.1). So, the heterogeneous multicore prototype served a useful purpose in confirming our expectations, but was not particularly interesting otherwise.

7.3 Non-speculative

The non-speculative prototype was also originally intended to serve only as a baseline for comparison to other approaches. However, the performance results were better than we anticipated, which led us into an exploration of the performance characteristics of speculation in large-scale processors, as discussed in Chapter 5. The non-speculative prototype removed BOOM's speculative features from the RTL, while keeping all other superscalar and out-of-order features of the processor, and did not execute any instructions speculatively.[2]

The results in Figure 7.1 show that the non-speculative prototype tended to perform nearly as well as or better than the Rocket core. On average, the non-speculative prototype performed 20% better than the Rocket core—slightly worse than the Rocket core on the `perlbench`, `gcc`, and `exchange2` benchmarks, and better than the Rocket core on the `mcf`, `omnetpp`, `xalancbmk`, `x264`, `deepsjeng`, `leela`, and `xz` benchmarks. Compared to the speculative BOOM core, the non-speculative prototype performed on average 40% worse—about a 30% performance penalty on the `perlbench` and `omnetpp` benchmarks, about a 40% performance penalty on the `gcc`, `mcf`, `x264`, `leela`, and `xz` benchmarks, about a 50% performance penalty on the `xalancbmk`, `deepsjeng` benchmarks, and about a 60% performance penalty on the `exchange2` benchmark. These performance results may sound dismal, but at the time of submission the performance penalties for this non-speculative core were nearly as good as or slightly better than speculative cores with all relevant mitigations for the transient execution vulnerabilities applied.[3] The performance results for the non-speculative prototype demonstrated that implementing superscalar and out-of-order features on a core without speculation can still significantly improve performance.

Further experimentation with the non-speculative prototype confirmed the results of earlier work by Swanson *et al.* [414], demonstrating that increasing the fetch width and

[2]The BOOM core also implements a short forward branch optimization, which is a non-speculative technique for improving the performance of branches [501, p. 4].

[3]Large public cloud providers have privately mentioned that the effect of deploying all known and relevant current mitigations for the speculative execution vulnerablities in production is in the range of a 50% performance penalty, and still cannot fully protect against the vulnerabilities.

decode width, the size of the fetch buffer and reorder buffer, and the number of ALUs, FPUs, and LSUs all improved performance of the non-speculative core to the point that the largest configuration of the non-speculative core we measured performed with only a 13% performance penalty compared to the default configuration of the speculative BOOM core on some benchmarks. Figure 7.2 shows a performance comparison of the non-speculative prototype and BOOM configurations listed in Table 7.1. The largest configuration we tested was a fetch width of 16, decode width of 8, fetch buffer of 64, reorder buffer of 256, with 8 ALUs, 4 FPUs, and 2 LSUs, which is in the realm of a reasonable scale for a modern server processor.[4]

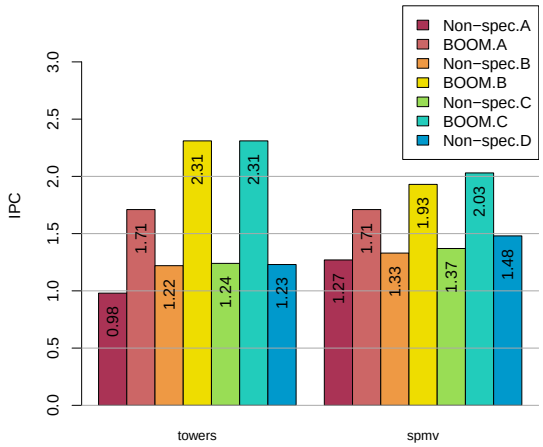

Figure 7.2: Equivalent variations on the non-speculative prototype and BOOM.

Table 7.1: Configurations of non-speculative prototype and BOOM

Variant	Fetch Width	Decode Width	Fetch Buffer	ROB	ALUs	FPUs	LSUs
A	8	3	24	96	3	1	1
B	8	4	32	128	4	2	2
C	8	5	40	130	5	2	2
D	16	8	64	256	8	4	2

The performance potential of non-speculative cores, combined with the ability to completely eliminate the transient execution vulnerabilities, places them in the running as a viable solution. Non-speculative cores may be suitable for special-purpose cloud or container infrastructure deployments which run extremely large-scale hardware, or primarily run smaller and short-lived workloads (also known as "serverless"), or exclusively serve privacy-centric workloads, or in legal jurisdictions with strong data privacy laws. However, selective speculation cores are a better fit from a performance perspective for most general-purpose server hardware, even though non-speculative cores are more secure.

[4]The B, C, and D configurations were too large and complex for FireSim to build them for the AWS F1 FPGAs, so we were not able run these comparisons on the FPGA simulations, and instead built and ran them within the Chipyard framework using the Verilator RTL simulator for a cycle-accurate behavioral model.

7.4 Selective speculation

The third and final prototype implemented selective speculation—mixing speculative and non-speculative execution of instructions in the same pipeline on a single core. This prototype explored the performance potential of keeping speculation where it is safe, and disabling speculation only where necessary for security.

The results in Figure 7.1 show that the performance of the selective speculation prototype depended heavily on the percentage of instructions in the workload that run speculatively rather than non-speculatively.[5] A benchmark run on the selective speculation prototype with speculation fully enabled has the same performance as on an unmodified BOOM core, and a benchmark run with speculation fully disabled has the same performance as on an entirely non-speculative core. The more interesting question is what happens when a workload mixes speculative and non-speculative execution of instructions. To model the effects of mixed workloads, we implemented a series of variations on the selective speculation prototype, shown in Table 7.2, where some branch instructions enable speculative execution of following instructions and other branch instructions delay execution of following instructions until they can execute non-speculatively. The performance of all benchmarks running on the prototype that disabled most speculation (Selective.B, labeled "Mostly non-spec" in Figure 7.1) was better than the Rocket core and at least slightly better than the non-speculative core. The performance of benchmarks running on the prototype that enabled most speculation (Selective.C, labeled "Mostly spec" in Figure 7.1) was generally much closer to the performance of the BOOM core.

Table 7.2: Characteristics of selective speculation variations

Variant	**Speculative**	**Non-Speculative**
Selective.A	Branch if Greater-Than or Equal (`BGE`)	all other branch instructions
Selective.B	Branch if Equal (`BEQ`)	all other branch instructions
Selective.C	all other branch instructions	Branch if Equal (`BEQ`)
Selective.D	all other branch instructions	Branch if Greater-Than or Equal (`BGE`)

Figure 7.3 compares all the variations in Table 7.2[6], showing that performance of the benchmarks by IPC varies depending on the relative proportion of speculative to non-speculative branch instructions. In this particular set of benchmarks, `bge` branch instructions (branch if greater or equal) are relatively rare, so for the Selective.A prototype—which only speculates `bge` branch instructions and runs all other branch instructions as non-speculative—the performance by IPC is only slightly better than the non-speculative core. But, for the Selective.D prototype—which speculates all branch instructions except `bge`—the performance by IPC for most of the benchmarks is only slightly worse than the BOOM core. In three of the benchmarks (`multiply`, `spmv`, and `vvadd`) the performance

[5]This is similar to the way performance on the heterogeneous multicore depended on whether a workload runs on a speculative core or a non-speculative core.

[6]The same as Figure 7.2, we built and ran the benchmarks in Figure 7.3 within the Chipyard framework using the Verilator RTL simulator for a cycle-accurate behavioral model. The SPEC CPU benchmark suite scores used in Figure 7.1 are a measure of normalized execution time, which depends on CPU performance by IPC, but also depends on the performance of other components in the system such as the cache and memory hierarchy, number of cores, number of threads, and clock speed. This means that the SPEC CPU score is a more comprehensive measure of an overall system than IPC alone, but it also means that direct comparison of the score results is most useful when the systems being compared are similar.

on the Selective.D prototype is slightly better than the BOOM core, which could be explained if some `bge` branches were misspeculated on BOOM, so that the non-speculative `bge` ends up performing better by avoiding the hit of misspeculation. In contrast, the `beq` branch instructions (branch if equal) are far more common in the `dhrystone` and `towers` benchmarks in Figure 7.3, so the performance of these benchmarks on the Selective.B prototype—which only speculates `beq` branch instructions—is substantially better than the non-speculative core in the `dhrystone` benchmark, and nearly as good as the BOOM core in the `towers` benchmark. And, for the Selective.C prototype—which speculates all branch instructions except `beq`—the performance by IPC is substantially worse than the BOOM core on the `towers` benchmark and almost as bad as the non-speculative core in the `dhrystone` benchmark. In the other benchmarks, the `beq` branch instruction is rare, so the performance on the Selective.B prototype is about the same as the Selective.A prototype, and the performance on the Selective.C prototype is about the same as the Selective.D prototype.

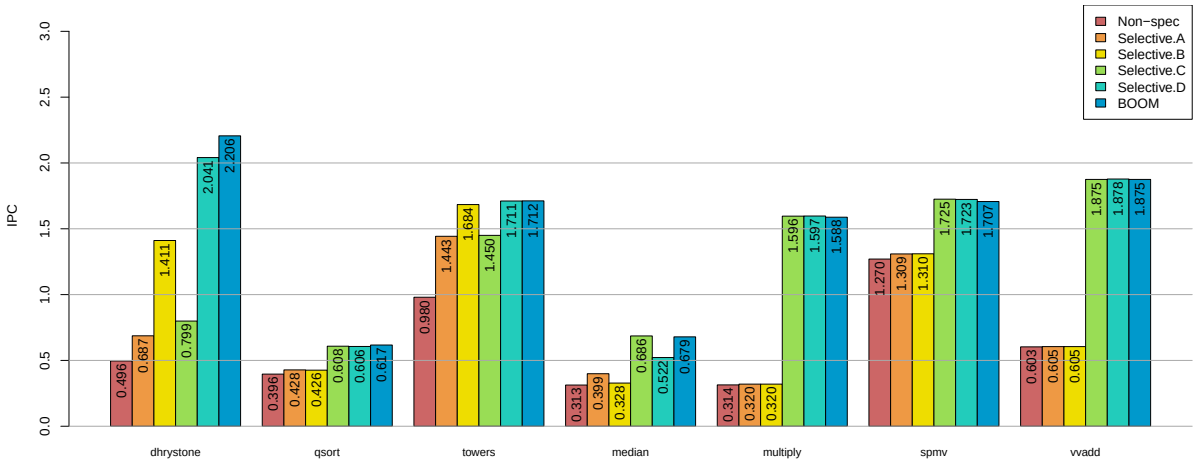

Figure 7.3: Comparing a series of variations on the selective speculation core.

These results demonstrated that while workloads with mixed speculative and non-speculative execution of instructions do pay a performance penalty, they do so in proportion to their use non-speculative execution. Workloads that make only light use of non-speculative execution, suffer only a minor loss of performance. This effect of gradually degraded performance sets the selective speculation approach apart from the common approaches to mitigating the transient execution vulnerabilities at the time of submission. Out of the three approaches prototyped in this section, selective speculation cores are best for general-purpose server infrastructure deployments, because they simplify resource allocation by keeping all cores identical and have no performance penalty for code executed as entirely speculative.

Chapter 8

Conclusions

The publication of the transient execution vulnerabilities in 2018 created a shock wave in hardware and software security research that continues to unfold. Many new variants of the vulnerabilities have been published over the years, usually with a collection of mitigations for the symptoms of each variant. Some mitigations are adopted by major hardware vendors and deployed by public providers of large-scale server hardware. Many proposed mitigations never see significant adoption because the performance penalty of the specific mitigation is too high, or because the performance penalty of deploying all the relevant mitigations is so high that hardware vendors and infrastructure providers choose to adopt only a subset of mitigations with the greatest impact for their particular target use case. If we continue this way, we can look forward to many generations of hardware debilitated by performance penalties from increasing layers of mitigations as new variants are discovered, and yet still vulnerable to variants that have yet to be discovered, or variants that have been discovered by malicious parties and have yet to be reported or mitigated. Aside from the narrow subset of Meltdown-type vulnerabilities, there is no real resolution to the transient execution vulnerabilities on the horizon.

This dissertation has sought to demonstrate the potential of research avenues that fundamentally rethink the role of speculation in modern server hardware. We have explored whether speculation could be partially or completely eliminated, and the security and performance implications of doing so. We conclude that eliminating speculation, partially or completely, is a feasible approach to mitigating the transient execution vulnerabilities on large-scale server hardware, from the perspectives of both security and performance.

Chapters 2 and 3 established the background for this dissertation. We described the unique requirements that large-scale server infrastructure environments have for hardware and software security, and how key concepts of large-scale server infrastructures developed over time. We then outlined how the transient execution vulnerabilities undermine crucial assumptions about hardware security that researchers and the industry have been making for decades, and characterized the many variants and mitigations published in recent years. 7 years on from the first discovery of the speculation-based transient execution vulnerabilities, eliminating or restricting speculation continues to be the only complete and reliable countermeasure, and the performance of non-speculative and selective speculation approaches are comparable to other proposed mitigations, but more secure.

Chapter 4 explored the potential for heterogeneous multicore architectures that combine speculative and non-speculative cores. Heterogeneous multicores make it possible run a process or thread as non-speculative, but cannot provide any tighter level of control over speculation. Performance is determined by which core runs the workload—a big speculative

core in a heterogeneous multicore system performs as well as a single speculative core and a small non-speculative core in a heterogeneous multicore system performs as poorly as a single non-speculative core. Because heterogenous multicores are already shipping in (mobile and laptop) production hardware, they are more amenable to production deployments in the short-term future. However, these heterogeneous multicores are inflexible for resource allocation—the decision of how much capacity to allocate to speculative and non-speculative execution must be made at the time of hardware manufacture—which makes them unsuitable for large-scale server infrastructures.

Chapter 5 explored the logical limits of performance for a modern superscalar architecture without speculation. Non-speculative cores are more secure than either heterogeneous multicores or selective speculation cores, because they completely eliminate the security risk of speculation, but they also cannot use speculation to improve performance even when it would be safe to do so. Performance of a non-speculative core is never as good as an equivalently sized speculative core, but it can be improved by increasing the size of fetch and issue stage components of the pipeline. It is unlikely that non-speculative superscalar cores will ever beat the performance of unmitigated speculative cores, but it is feasible that they may reach the point of consistently performing as well as or better than speculative cores with all relevant transient execution mitigations applied. The performance potential of non-speculative cores, combined with the ability to completely eliminate the speculation-based transient execution vulnerabilities, places them in the running as a viable solution. Non-speculative cores may be suitable for special-purpose server infrastructure deployments that exclusively serve security-critical or privacy-centric workloads, or smaller and short-lived workloads, such as serverless functions, where speculation performs poorly anyway. However, keeping speculative execution features as an option for peak performance will be desirable for most general-purpose server deployments.

Chapter 6 explored the potential for hardware architectures that include both speculative and non-speculative features on a single core. Selective speculation cores give the systems software developer precise control over when to use speculation features. Performance degrades gradually in proportion to the use of non-speculative features—code run as entirely speculative pays no performance penalty, and code run as lightly non-speculative only pays a minor performance penalty—so selective speculation cores have peak performance equivalent to speculative cores, while also allowing for fine-grained control over speculative execution. Selective speculation cores are more flexible than heterogeneous multicores for resource allocation in large-scale server infrastructures, because all cores in the system are identical and equally able to combine speculative and non-speculative execution. Selective speculation cores are most suitable for general-purpose server infrastructure deployments, because they put the choice of when to trade performance for security into the hands of the customers, and maintain flexibility in resource allocation, without sacrificing performance in the common case.

8.1 Future work

The scope of this dissertation has been limited to what one researcher can accomplish within the short period of a PhD research program. While the work is enough to reveal promising potential, a complete hardware design and implementation based on the concepts discussed is an extensive and multi-year research agenda for a group of researchers. This section briefly discusses some future research directions suggested by this work.

8.1.1 Heterogeneous multicores

While heterogeneous multicores are not a good choice for large-scale server infrastructures, they may be a reasonable approach to consider for mobile and laptop devices, which have less extreme requirements for flexible resource allocation. The disruption of shifting to heterogeneous multicores would be particularly minimal in products where the hardware is already running a combination of big and little cores for performance and energy consumption considerations, as in some modern smartphones and laptops.

It would be worth exploring whether the performance gap between the speculative and non-speculative cores of a heterogeneous multicore system could be improved by implementing the "little" cores as non-speculative superscalar cores as discussed in Chapter 5 instead of in-order scalar cores. Such a system would be less useful for reducing energy consumption in mobile and laptop devices, but could be more useful for improving security on desktop devices without sacrificing performance.

8.1.2 Non-speculative cores

Several further research avenues are worth considering for improving the performance of non-speculative superscalar cores. Increasing the size of fetch and issue stage components in the pipeline showed potential in the prototype in Section 7.3, and could be explored further. The largest non-speculative prototypes in Section 7.3 were too large to fit on an AWS F1 FPGA, but FireSim's Golden Gate [274] compiler does have some limited ability to split RTL simulations across multiple FPGAs, while still producing bit-identical, cycle-accurate results—which may make it possible to test larger cores on FireSim, instead of only on Verilator. Multi-threading is another avenue worth exploring for non-speculative cores, for efficiency through parallelism rather than speculation. Running multiple threads on each core means the pipeline is less likely to stall, because even if it is held up on one workload waiting for a non-speculative branch instruction to resolve or a memory load to complete, it can still keep instructions for other workloads flowing through the execution stage. The addition of dedicated functional units for evaluating branch conditions, separate from the general ALUs, might be worth exploring as a way to improve throughput by ensuring that branch conditions are always evaluated as soon as possible, and not held up by other arithmetic or logic instructions in the pipeline. These research avenues could potentially also benefit heterogeneous multicores or selective speculation cores.

8.1.3 Selective speculation

The greatest challenges for selective speculation countermeasures for the transient execution vulnerabilities, as discussed in Section 3.3.2.4 and Chapter 6 is determining where speculation is safe or unsafe, and how to disable speculation with the least possible disruption to legacy software stacks while providing strong security guarantees. The approach to selective speculation described in Chapter 6 and prototyped in Section 7.4 is manual—the decision of whether speculation is safe or unsafe is left to the software or compiler developer at the granularity of a single instruction, so the approach can never provide strong security guarantees. The approaches to selective speculation in subsequent work by others discussed in Section 3.3.2.4 are more automated—the pipeline makes all the decisions about where speculation is safe or unsafe—but those approaches still do not manage to provide strong security guarantees because they miss some scenarios where

speculation is unsafe or because the implementation fails to disable speculation where the design intended. There is room for a middle-ground approach that provides strong security guarantees by disabling speculation for a security domain—such as a container, VM, secure enclave, serverless function, or small region of code—to protect code within the security domain from both cross-domain transient execution attacks launched outside the security domain and same-domain attacks launched within the security domain, and also serve as a sandbox preventing code inside the security domain from launching cross-domain attacks on any other part of the system.

One approach to the resource allocation problem on heterogeneous multicores that could be worth exploring is a core configuration option to enable or disable speculation for an entire core—similar to the way Intel and AMD's Indirect Branch Restricted Speculation (IBRS) or Single Thread Indirect Branch Predictors (STIBP) features can be configured. While IBRS and STIBP were ineffective countermeasures with prohibitive performance penalties, a feature to completely disable speculation would provide stronger security guarantees, and implementing the non-speculative features more like a selective speculation non-speculative security domain could potentially achieve reasonable performance.

Bibliography

[1] O. Aciiçmez, "Yet another MicroArchitectural Attack: Exploiting I-Cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, New York, NY, USA: Association for Computing Machinery, Nov. 2007, pp. 11–18.

[2] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New Results on Instruction Cache Attacks," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds., Berlin, Heidelberg: Springer, 2010, pp. 110–124.

[3] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, New York, NY, USA: Association for Computing Machinery, Mar. 2007, pp. 312–320.

[4] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting Secret Keys Via Branch Prediction," in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed., Berlin, Heidelberg: Springer, 2006, pp. 225–242.

[5] O. Aciicmez and J.-P. Seifert, "Cheap Hardware Parallelism Implies Cheap Security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, Sep. 2007, pp. 80–91.

[6] W. B. Ackerman and W. W. Plummer, "An Implementation of a Multiprocessing Computer System," in *Proceedings of the First ACM Symposium on Operating System Principles*, New York, NY, USA: ACM, 1967, pp. 5.1–5.10.

[7] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy, "A Virtual Machine System for the 360/40," IBM Cambridge Scientific Center, Cambridge, MA, USA, Tech. Rep. 36.010, May 1966.

[8] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: ACM, 2006, pp. 2–13.

[9] B. A. Ahmad, *Real time Detection of Spectre and Meltdown Attacks Using Machine Learning*, Jun. 2020.

[10] I. Ahmad, J. M. Anderson, A. M. Holler, R. Kambo, and V. Makhija, "An analysis of disk performance in VMware ESX server virtual machines," in *2003 IEEE International Conference on Communications (Cat. No.03CH37441)*, Oct. 2003, pp. 65–76.

[11] S. Ainsworth, "GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 592–606.

[12] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 132–144.

[13] S. Ainsworth and T. M. Jones, "The Guardian Council: Parallel Programmable Hardware Security," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 1277–1293.

[14] M. Alam, S. Bhattacharya, and D. Mukhopadhyay, "Victims Can Be Saviors: A Machine Learning–based Detection for Micro-Architectural Side-Channel Attacks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 2, pp. 1–31, Apr. 2021.

[15] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port Contention for Fun and Profit," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 870–887.

[16] Amazon, *Firecracker*, 2019. [Online]. Available: `https://firecracker-microvm.github.io/`.

[17] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 87–101, Apr. 1964.

[18] A. Amid *et al.*, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.

[19] A. Aminot, Y. Lhuillier, A. Chateigner, and H.-P. Charles, "On the advantage of time-varying diversity of workload on functionally asymmetric multi-core," in *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, New York, NY, USA: Association for Computing Machinery, Jan. 2014, pp. 11–13.

[20] N. Amit, F. Jacobs, and M. Wei, "{JumpSwitches}: Restoring the Performance of Indirect Branches In the Era of Spectre," Renton, WA, USA: USENIX Association, Jul. 2019, pp. 285–300.

[21] J. Anderson, S. Godfrey, and R. N. Watson, "Towards oblivious sandboxing with Capsicum," *FreeBSD Journal*, vol. Security, no. May/Jun 2017, May 2017. [Online]. Available: `https://www.freebsdfoundation.org/past-issues/security/`.

[22] R. Anderson and M. Kuhn, "Tamper Resistance – a Cautionary Note," in *2nd USENIX Workshop on Electronic Commerce (EC 96)*, Oakland, CA: USENIX Association, Nov. 1996.

[23] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 623–639.

[24] A. L. D. Antón, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Fault Attacks on Access Control in Processors: Threat, Formal Analysis and Microarchitectural Mitigation," *IEEE Access*, vol. 11, pp. 52 695–52 711, 2023.

[25] S. Anzai, M. Misono, R. Nakamura, Y. Kuga, and T. Shinagawa, "Towards isolated execution at the machine level," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 68–77.

[26] A. Arcangeli, *[PATCH] seccomp: Secure computing support*, Mar. 2005. [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554`.

[27] K. Asanović *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2016-17, Apr. 2016, p. 9.

[28] T. Asheim, B. Grot, and R. Kumar, "A Specialized BTB Organization for Servers," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Chicago Illinois: ACM, Oct. 2022, pp. 548–549.

[29] T. Asheim, T. A. Khan, B. Kasicki, and R. Kumar, "Impact of microarchitectural state reuse on serverless functions," in *Proceedings of the Eighth International Workshop on Serverless Computing*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 7–12.

[30] R. Bahmani *et al.*, "{CURE}: A Security Architecture with {CUstomizable} and Resilient Enclaves," Aug. 2021, pp. 1073–1090.

[31] J. Balkind *et al.*, "BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 699–714.

[32] J. Balkind *et al.*, "OpenPiton: An Open Source Manycore Research Framework," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 217–232, Mar. 2016.

[33] J. Balkind *et al.*, "OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores," in *Proceedings of the 3rd Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, 2019, p. 6.

[34] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA: USENIX Association, 1999, pp. 45–58.

[35] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019, pp. 151–164.

[36] K. Barber, M. Ghaniyoun, Y. Zhang, and R. Teodorescu, "A Pre-Silicon Approach to Discovering Microarchitectural Vulnerabilities in Security Critical Applications," *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 9–12, Jan. 2022.

[37] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," Aug. 2022, p. 18. [Online]. Available: `https://www.vusec.net/projects/bhi-spectre-bhb/`.

[38] A. Barenghi and G. Pelosi, "Side-channel security of superscalar CPUs: Evaluating the impact of micro-architectural features," in *Proceedings of the 55th Annual Design Automation Conference*, San Francisco, California: Association for Computing Machinery, Jun. 2018, pp. 1–6.

[39] P. R. Barham, "A fresh approach to file system quality of service," in *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97)*, May 1997, pp. 113–122.

[40] P. Barham *et al.*, "Xen and the Art of Virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, 2003, pp. 164–177.

[41] J. Barr, *Amazon EC2 Beta*, Aug. 2006. [Online]. Available: `https://aws.amazon.com/blogs/aws/amazon_ec2_beta/`.

[42] M. Behnia *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 1046–1060.

[43] J. Behrens, A. Belay, and M. F. Kaashoek, "Performance evolution of mitigating transient execution attacks," in *Proceedings of the Seventeenth European Conference on Computer Systems*, Rennes France: ACM, Mar. 2022, pp. 251–265.

[44] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA: USENIX Association, Apr. 2005, pp. 41–41.

[45] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *Proceedings of the 15th USENIX Security Symposium*, Vancouver, Canada: USENIX Association, Aug. 2006, pp. 305–320.

[46] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[47] D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., 2004, p. 37. [Online]. Available: `http://cr.yp.to/papers.html#cachetiming`.

[48] V. Berstis, "Security and Protection of Data in the IBM System/38," in *Proceedings of the 7th Annual Symposium on Computer Architecture*, New York, NY, USA: ACM, 1980, pp. 245–252.

[49] A. Bhattacharyya *et al.*, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 785–800.

[50] S. Bhunia and M. Tehranipoor, *Hardware Security: A Hands-on Learning Approach*. Morgan Kaufmann, Oct. 2018.

[51] E. W. Biederman, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, vol. 1, Ottowa, Canada, 2006, pp. 101–112.

[52] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef, "Leveraging the Serverless Architecture for Securing Linux Containers," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Jun. 2017, pp. 401–404.

[53] T. M. Birhanu, Z. Li, H. Sekiya, N. Komuro, and Y.-J. Choi, *Efficient Thread Mapping for Heterogeneous Multicore IoT Systems*, Research Article, Feb. 2017. [Online]. Available: `https://www.hindawi.com/journals/misy/2017/3021565/`.

[54] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor," *IEEE Micro*, vol. 35, no. 2, pp. 46–55, Mar. 2015.

[55] M. Bognar, H. Winderix, J. V. Bulck, and F. Piessens, "MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling," in *Proceedings of the 8th IEEE European Symposium on Security and Privacy*, Delft, Netherlands: IEEE, Feb. 2023.

[56] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. Brinkmann, "Non-intrusive virtualization management using libvirt," European Design and Automation Association, Mar. 2010, pp. 574–579.

[57] J. Boulle, *Celebrating the Open Container Initiative Image Specification*, Apr. 2016. [Online]. Available: `https://coreos.com/blog/oci-image-specification.html`.

[58] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA: Association for Computing Machinery, Oct. 2019, pp. 42–56.

[59] A. Bradbury, G. Ferris, and R. Mullins, "Tagged memory and minion cores in the lowRISC SoC," University of Cambridge, Computer Laboratory, Technical Report lowRISC-MEMO 2014-001, Dec. 2014.

[60] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith, "VM-based Security Overkill: A Lament for Applied Systems Security Research," in *Proceedings of the 2010 New Security Paradigms Workshop*, New York, NY, USA: ACM, 2010, pp. 51–60.

[61] E. A. Brewer, "Kubernetes and the Path to Cloud Native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, New York, NY, USA: ACM, 2015, pp. 167–167.

[62] M. S. Brunella, G. Bianchi, S. Turco, F. Quaglia, and N. Blefari-Melazzi, "Foreshadow-VMM: Feasibility and Network Perspective," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2019, pp. 257–259.

[63] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization*. Morgan & Claypool, 2017.

[64] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, Nov. 1997.

[65] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, 12:1–12:51, Nov. 2012.

[66] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of Side-Channel Models via Observation Refinement," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 578–591.

[67] Y. Bulygin, *CPU side-channels vs. virtualization malware: The good, the bad, or the ugly*, Seattle, WA, Apr. 2008.

[68] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Queue*, vol. 14, no. 1, 10:70–10:93, Jan. 2016.

[69] J. P. Buzen and U. O. Gagliardi, "The Evolution of Virtual Machine Architecture," in *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, New York, NY, USA: ACM, 1973, pp. 291–299.

[70] S. Canakci *et al.*, "ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance," in *In Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, San Jose, CA, May 2023.

[71] C. Canella, S. M. Pudukotai Dinakarrao, D. Gruss, and K. N. Khasawneh, "Evolution of Defenses against Transient-Execution Attacks," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 169–174.

[72] C. Canella *et al.*, "A Systematic Evaluation of Transient Execution Attacks and Defenses," *arXiv:1811.05441 [cs]*, May 2019.

[73] *Capabilities(7) man page*, Feb. 2018. [Online]. Available: `http://man7.org/linux/man-pages/man7/capabilities.7.html`.

[74] S. Carnà, S. Ferracci, F. Quaglia, and A. Pellegrini, "Fight Hardware with Hardware: System-wide Detection and Mitigation of Side-Channel Attacks using Performance Counters," *Digital Threats: Research and Practice*, Apr. 2022.

[75] C. Carruth, *Introduce the "retpoline" x86 mitigation technique for variant #2 of the speculative execution vulnerabilities disclosed today, specifically identified by CVE-2017-5715, "Branch Target Injection", and is one of the two halves to Spectre.*. Jan. 2018. [Online]. Available: `https://reviews.llvm.org/D41723`.

[76] C. Carruth, *Speculative Load Hardening (a Spectre variant #1 mitigation)*, Mar. 2018. [Online]. Available: `https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html`.

[77] L. Catuogno and C. Galdi, "On the Evaluation of Security Properties of Containerized Systems," in *2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS)*, Dec. 2016, pp. 69–76.

[78] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical Foundations for Software Spectre Defenses," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 666–680.

[79] S. Cauligi *et al.*, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 913–926.

[80] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "BOOM v2: An open-source out-of-order RISC-V core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep. 2017.

[81] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun. 2015.

[82] A. Chakraborty, N. Singh, S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "Timed speculative attacks exploiting store-to-load forwarding bypassing cache-based countermeasures," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 553–558.

[83] R. Chandramouli, A. Singhal, D. Wijesekera, and C. Liu, "A Methodology for Enabling Forensic Analysis Using Hypervisor Vulnerabilities Data," National Institute of Standards and Technology, Tech. Rep. NIST Internal or Interagency Report (NISTIR) 8221, Jun. 2019. [Online]. Available: `https://csrc.nist.gov/publications/detail/nistir/8221/final`.

[84] S. Chattopadhyay and A. Roychoudhury, "Symbolic Verification of Cache Side-Channel Freedom," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2812–2823, Nov. 2018.

[85] O. Chatzopoulos, G.-M. Fragkoulis, G. Papadimitriou, and D. Gizopoulos, "Towards Accurate Performance Modeling of RISC-V Designs," in *roceedings of Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*, Jun. 2021.

[86] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A Formal Approach to Secure Speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, Jun. 2019, pp. 288–28 815.

[87] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, Jun. 2019, pp. 142–157.

[88] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 16–32.

[89] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*, First. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[90] A. Choudhari, S. Guilley, and K. Karray, "SpecDefender: Transient Execution Attack Defender using Performance Counters," in *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 15–24.

[91] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 607–622.

[92] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2016, pp. 713–717.

[93] E. F. Codd, "Multiprogramming," in *Advances in Computers*, F. L. Alt and M. Rubinoff, Eds., vol. 3, Elsevier, Jan. 1962, pp. 77–153.

[94] E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi, "Multiprogramming STRETCH: Feasibility Considerations," *Communications of the ACM*, vol. 2, no. 11, pp. 13–17, Nov. 1959.

[95] Y. Cohen *et al.*, "HammerScope: Observing DRAM Power Consumption Using Rowhammer," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 547–561.

[96] P. Colp *et al.*, "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, 2011, pp. 189–202.

[97] T. Combe, A. Martin, and R. D. Pietro, "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[98] *Control Program-67 Cambridge Monitor System*. Hawthorne, New York: IBM Corporation, Oct. 1971.

[99] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 45–60.

[100] F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics: The First Seven Years," in *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, New York, NY, USA: ACM, 1972, pp. 571–583.

[101] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-sharing System," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, New York, NY, USA: ACM, 1962, pp. 335–344.

[102] J. Corbet, "A page-table isolation update," *LWN.net*, Apr. 2018. [Online]. Available: `https://lwn.net/Articles/752621/`.

[103] J. Corbet, *Kernel Summit 2006: Paravirtualization and containers*, Jul. 2006. [Online]. Available: `https://lwn.net/Articles/191923/`.

[104] J. Corbet, "Meltdown strikes back: The L1 terminal fault vulnerability," *LWN*, Aug. 2018. [Online]. Available: `https://lwn.net/Articles/762570/`.

[105] J. Corbet, *Process containers*, May 2007. [Online]. Available: `https://lwn.net/Articles/236038/`.

[106] J. Corbet, "Taming STIBP," *LWN.net*, Nov. 2018. [Online]. Available: `https://lwn.net/Articles/773118/`.

[107] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious Server Security," in *Proceedings of the 14th USENIX Conference on System Administration*, New Orleans, Louisiana: USENIX Association, 2000, pp. 355–368.

[108] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, Sep. 1981.

[109] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE," in *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual: Internet Society, 2021.

[110] P. J. Denning, "Fault Tolerant Operating Systems," *ACM Comput. Surv.*, vol. 8, no. 4, pp. 359–389, Dec. 1976.

[111] P. J. Denning, "Performance Modeling: Experimental Computer Science As Its Best," *Communications of the ACM, President's Letter*, vol. 24, no. 11, pp. 725–727, Nov. 1981.

[112] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[113] J. Depoix and P. Altmeyer, "Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning," 2018.

[114] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments," Aug. 2020, pp. 451–468.

[115] G. Dessouky *et al.*, "HardFails: Insights into Software-Exploitable Hardware Bugs," 2019, pp. 213–230.

[116] A. Dhavlle, S. Rafatirad, H. Homayoun, and S. M. P. Dinakarrao, "CR-spectre: Defense-aware ROP injected code-reuse based dynamic spectre," in *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, Leuven, BEL: European Design and Automation Association, May 2022, pp. 508–513.

[117] L. I. Dickman, "Small Virtual Machines: A Survey," in *Proceedings of the Workshop on Virtual Computer Systems*, New York, NY, USA: ACM, 1973, pp. 191–202.

[118] X. Dong, Z. Shen, J. Criswell, A. Cox, and S. Dwarkadas, "Spectres, Virtual Ghosts, and Hardware Support," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA: ACM, Jun. 2018, 5:1–5:9.

[119] W. Drewry, *Dynamic seccomp policies (using BPF filters)*, Jan. 2012. [Online]. Available: `https://lwn.net/Articles/475019/`.

[120] S. W. Dunwell, "Design Objectives for the IBM Stretch Computer," in *Papers and Discussions Presented at the December 10-12, 1956, Eastern Joint Computer Conference: New Developments in Computers*, New York, NY, USA: ACM, 1957, pp. 20–22.

[121] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," 2022, pp. 3861–3877.

[122] J. P. Eckert, "UNIVAC-Larc, the Next Step in Computer Design," in *Papers and Discussions Presented at the December 10-12, 1956, Eastern Joint Computer Conference: New Developments in Computers*, New York, NY, USA: ACM, 1957, pp. 16–20.

[123] H. Eißfeldt, "POSIX: A Developer's View of Standards," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA: USENIX Association, 1997, pp. 24–24.

[124] D. M. England, "Capability Concept Mechanism and Structure in System 250," in *Proceedings of the International Workshop on Protection in Operating Systems*, France: Institut de Recherche d'Informatique et de Automatique (IRIA), Aug. 1974, pp. 63–82.

[125] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, and J. Szefer, "Survey of Approaches and Techniques for Security Verification of Computer Systems," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 19, no. 1, 6:1–6:34, Jan. 2023.

[126] M. Escouteloup, J. Fournier, J.-L. Lanet, and R. Lashermes, "Recommendations for a radically secure ISA," in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Valencia, Spain: ACM, May 2020, p. 7.

[127] M. Escouteloup, R. Lashermes, J. Fournier, and J.-L. Lanet, "Under the Dome: Preventing Hardware Timing Information Leakage," in *Smart Card Research and Advanced Applications*, V. Grosso and T. Pöppelmann, Eds., Cham: Springer International Publishing, Mar. 2022, pp. 233–253.

[128] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[129] D. Evtyushkin, R. Riley, N. C. a. E. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 693–707.

[130] X. Fabian, M. Guarnieri, and M. Patrignani, "Automatic Detection of Speculative Execution Combinations," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 965–978.

[131] R. S. Fabry, "A user's view of capabilities," University of Chicago, ICR Quarterly Report 15, Nov. 1967.

[132] R. S. Fabry, "Preliminary description of a supervisor for a machine oriented around capabilities," University of Chicago, ICR Quarterly Report 18, Aug. 1968.

[133] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, Jul. 2020, pp. 1–6.

[134] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2019, pp. 994–999.

[135] M. R. Fadiheh *et al.*, "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 222–235, Jan. 2023.

[136] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172.

[137] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 37–48.

[138] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 555–568, Apr. 2017.

[139] L. Fiolhais and L. Sousa, "Transient-Execution Attacks: A Computer Architect Perspective," *ACM Computing Surveys*, Jun. 2023.

[140] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[141] M. J. Flynn and A. Podvin, "Shared Resource Multiprocessing," *Computer*, vol. 5, no. 2, pp. 20–28, Feb. 1972.

[142] A. Fogh, *Negative Result: Reading Kernel Memory From User Mode*, Jul. 2017. [Online]. Available: `https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/`.

[143] J. M. Frankovich and H. P. Peterson, "A Functional Description of the Lincoln TX-2 Computer," in *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, New York, NY, USA: ACM, 1957, pp. 146–155.

[144] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: Leaking Secrets to Past Instructions," in *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 117–126.

[145] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, Jun. 2019, pp. 1–6.

[146] J. Galbally, "A new Foe in biometrics: A narrative review of side-channel attacks," *Computers & Security*, vol. 96, p. 101 902, Sep. 2020.

[147] S. W. Galley, "PDP-10 virtual machines," ACM, Mar. 1973, pp. 30–34.

[148] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2017, pp. 237–248.

[149] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is Not Transparency: VMM Detection Myths and Realities," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, Berkeley, CA, USA: USENIX Association, 2007, p. 6.

[150] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, vol. 1, 2003, pp. 253–285.

[151] J. Ge, Y. Li, Y. Zheng, Y. Liu, and S. M. Habib, "More Secure Collaborative APIs resistant to Flush-Based Cache Attacks on Cortex-A9 Based Automotive System," in *Proceedings of the 6th ACM Computer Science in Cars Symposium*, New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 1–9.

[152] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time Protection: The Missing OS Abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–17.

[153] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr. 2018.

[154] Q. Ge, Y. Yarom, and G. Heiser, "No Security Without Time Protection: We Need a New Hardware-Software Contract," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–9.

[155] D. Genkin and Y. Yarom, "Whack-a-Meltdown: Microarchitectural Security Games [Systems Attacks and Defenses]," *IEEE Security Privacy*, vol. 19, no. 1, pp. 95–98, Jan. 2021.

[156] M. Ghaniyoun, *A Short Review of Performance and Security of SpecTerminator*, Jan. 2023.

[157] M. Ghaniyoun, K. Barber, Y. Xiao, Y. Zhang, and R. Teodorescu, "TEESec: Pre-Silicon Vulnerability Discovery for Trusted Execution Environments," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 1–15.

[158] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "INTROSPECTRE: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2021, pp. 874–887.

[159] S. Gill, "Parallel Programming," *The Computer Journal*, vol. 1, no. 1, pp. 2–10, Jan. 1958.

[160] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative Probing: Hacking Blind in the Spectre Era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1871–1885.

[161] R. P. Goldberg, "Architectural Principles for Virtual Computer Systems," PhD Thesis, Harvard University, Cambridge, MA, 1972.

[162] R. P. Goldberg, "Architecture of Virtual Machines," in *Proceedings of the AFIPS National Computer Conference*, New York, NY, USA: ACM, Jun. 1973, pp. 309–318.

[163] R. P. Goldberg, "Survey of Virtual Machine Research," *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[164] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanović, "Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture," in *Proceedings of Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, Phoenix, AZ, USA: ACM, Jun. 2019, p. 7.

[165] Google, *Chrome OS Virtual Machine Monitor*, Sep. 2018. [Online]. Available: `https://chromium.googlesource.com/chromiumos/platform/crosvm/`.

[166] Google, *Fuchsia is not Linux: A modular, capability-based operating system*, Oct. 2018. [Online]. Available: `https://fuchsia.dev/fuchsia-src`.

[167] Google, *gVisor - Container Runtime Sandbox*, Feb. 2019. [Online]. Available: `https://github.com/google/gvisor`.

[168] S. Graber, *LXD 2.0*, Mar. 2016. [Online]. Available: `https://stgraber.org/2016/03/11/lxd-2-0-blog-post-series-012/`.

[169] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2020.

[170] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," 2018, pp. 955–972.

[171] C. Green, C. Nelson, M. Thottethodi, and T. N. Vijaykumar, *SafeBet: Secure, Simple, and Fast Speculative Execution*, Jun. 2023.

[172] M. Griffin and B. Dongol, "Verifying Secure Speculation in Isabelle/HOL," in *Formal Methods*, M. Huisman, C. Păsăreanu, and N. Zhan, Eds., Cham: Springer International Publishing, 2021, pp. 43–60.

[173] A. Grünbacher, "POSIX Access Control Lists on Linux," in *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, Texas: USENIX Association, Jun. 2003, pp. 259–272.

[174] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds., Cham: Springer International Publishing, 2017, pp. 161–176.

[175] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., Cham: Springer International Publishing, 2016, pp. 279–299.

[176] L. Guan *et al.*, "Building a Trustworthy Execution Environment to Defeat Exploits from both Cyber Space and Physical Space for ARM," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 3, pp. 438–453, May 2019.

[177] R. Guanciale, M. Balliu, and M. Dam, "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1853–1869.

[178] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled Detection of Speculative Information Flows," *arXiv:1812.08639 [cs]*, Jul. 2019.

[179] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-Software Contracts for Secure Speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 1868–1883.

[180] P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy, and D. A. Patterson, "A Performance Evaluation of the Intel iAPX 432," *SIGARCH Comput. Archit. News*, vol. 10, no. 4, pp. 17–26, Jun. 1982.

[181] N. Hardy, "KeyKOS Architecture," *SIGOPS Operating Systems Review*, vol. 19, no. 4, pp. 8–25, Oct. 1985.

[182] Z. He, G. Hu, and R. Lee, "New Models for Understanding and Reasoning about Speculative Execution Attacks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 40–53.

[183] Z. He, G. Hu, and R. B. Lee, "CloudShield: Real-time Anomaly Detection in the Cloud," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, New York, NY, USA: Association for Computing Machinery, Apr. 2023, pp. 91–102.

[184] G. Heiser, "For Safety's Sake: We Need a New Hardware-Software Contract!" *IEEE Design Test*, vol. 35, no. 2, pp. 27–30, Apr. 2018.

[185] J. Hennessy, "The era of security: Introduction," in *Proceedings of the 2018 IEEE Hot Chips Symposium*, Cupertino, CA: IEEE, Aug. 2018. [Online]. Available: `https://youtu.be/d5XzVF0sAZo`.

[186] J. Horn, *Speculative execution, variant 4: Speculative store bypass*, Feb. 2018. [Online]. Available: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[187] S. Hosseinzadeh, S. Laurén, and V. Leppänen, "Security in Container-based Virtualization Through vTPM," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, New York, NY, USA: ACM, 2016, pp. 214–219.

[188] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "IBM System/38 Support for Capability-based Addressing," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 341–348.

[189] W. Hu, A. Althoff, A. Ardeshiricham, and R. Kastner, "Towards Property Driven Hardware Security," in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec. 2016, pp. 51–56.

[190] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware Information Flow Tracking," *ACM Computing Surveys*, vol. 54, no. 4, 83:1–83:39, May 2021.

[191] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, "An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1010–1038, Jun. 2021.

[192] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs," 2018, pp. 255–266.

[193] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia, "Efficiently Tracking Application Interactions Using Lightweight Virtualization," in *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, New York, NY, USA: ACM, 2008, pp. 19–28.

[194] T. Huo *et al.*, "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 321–347, 2020. [Online]. Available: `https://tches.iacr.org/index.php/TCHES/article/view/8401`.

[195] J. Hur, S. Song, S. Kim, and B. Lee, "SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1473–1487.

[196] S. Hykes, *Docker 0.9: Introducing execution drivers and libcontainer*, Apr. 2014. [Online]. Available: `https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/`.

[197] S. Hykes, *Introducing runC: A lightweight universal container runtime*, Jun. 2015. [Online]. Available: `https://blog.docker.com/2015/06/runc/`.

[198] *iAPX 432 General Data Processor Architecture Reference Manual*. Aloha, Oregon: Intel Corporation, 1981.

[199] A. Ibrahim, H. Nemati, T. Schlüter, N. O. Tippenhauer, and C. Rossow, "Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1489–1503.

[200] "Intel Analysis of Speculative Execution Side Channels," Intel Corporation, White Paper 336983-001, Jan. 2018.

[201] K. Ishiguro and K. Kono, "Hardening Hypervisors Against Vulnerabilities in Instruction Emulators," in *Proceedings of the 11th European Workshop on Systems Security*, New York, NY, USA: ACM, 2018, 7:1–7:6.

[202] S. Islam *et al.*, "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks," *arXiv:1903.00446 [cs]*, Jun. 2019.

[203] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith, "Ghostbusting: Mitigating spectre with intraprocess memory isolation," in *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 1–11.

[204] Z. Jian and L. Chen, "A Defense Method Against Docker Escape Attack," in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, New York, NY, USA: ACM, 2017, pp. 142–146.

[205] H. Jin, Z. He, and W. Qiang, "SpecTerminator: Blocking Speculative Side Channels Based on Instruction Classes on RISC-V," *ACM Transactions on Architecture and Code Optimization*, Nov. 2022.

[206] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "KASPER: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *Proceedings 2022 Network and Distributed System Security Symposium*, San Diego, CA, USA: Internet Society, 2022.

[207] C. Joly and F. Serman, "Evaluation of tail call costs in eBPF," 2020.

[208] A. K. Jones, R. J. Chansler Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, 1979, pp. 117–127.

[209] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, Mar. 2015, pp. 342–346.

[210] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, Maastricht, The Netherlands, 2000.

[211] J. A. Kappel, A. Velte, and T. Velte, *Microsoft Virtualization with Hyper-V: Manage Your Datacenter with Hyper-V, Virtual PC, Virtual Server, and Application Virtualization*. McGraw Hill Professional, Sep. 2009.

[212] S. Karandikar *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA: IEEE, Jun. 2018, pp. 29–42.

[213] *Kata Containers - The speed of containers, the security of VMs*, May 2018. [Online]. Available: `https://katacontainers.io/`.

[214] *Kata Containers Architecture*, Jan. 2019. [Online]. Available: `https://github.com/kata-containers/documentation`.

[215] Kernel Developers, *Capacity Aware Scheduling*, 2020. [Online]. Available: `https://www.kernel.org/doc/html/latest/scheduler/sched-capacity.html`.

[216] Kernel Developers, *Energy Aware Scheduling*, Nov. 2020. [Online]. Available: `https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html`.

[217] Kernel Developers, *Energy Model of devices*, 2020. [Online]. Available: `https://www.kernel.org/doc/html/latest/power/energy-model.html`.

[218] B. Kernighan and M. McIlroy, *UNIX Time-sharing System: UNIX Programmer's Manual*, 7th. Murray Hill, New Jersey: Bell Telephone Laboratories, Incorporated, 1979, vol. 1.

[219] M. Kerrisk, *Namespaces in operation, part 1: Namespaces overview*, Jan. 2013. [Online]. Available: `https://lwn.net/Articles/531114/`.

[220] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, Jun. 2019, pp. 1–6.

[221] S. Kim *et al.*, "ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage," in *2020 IEEE Secure Development (SecDev)*, Sep. 2020, pp. 96–107.

[222] 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)*, Oct. 2018, pp. 974–987.

[223] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757 [cs]*, Jul. 2018.

[224] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[225] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, vol. 1666, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[226] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," *arXiv:1801.01203 [cs]*, Jan. 2018.

[227] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed., Berlin, Heidelberg: Springer, 1996, pp. 104–113.

[228] A. Kogler *et al.*, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, USA: USENIX Association, Aug. 2023.

[229] R. M. Kogut, "The Segment Based File Support System," ACM, Mar. 1973, pp. 35–42.

[230] L. Kohn and N. Margulis, "Introducing the Intel i860 64-bit microprocessor," *IEEE Micro*, vol. 9, no. 4, pp. 15–30, Aug. 1989.

[231] R. Koller and D. Williams, "Will Serverless End the Dominance of Linux in the Cloud?" In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, New York, NY, USA: ACM Press, 2017, pp. 169–173.

[232] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," Jul. 2018.

[233] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation," IEEE Computer Society, May 2020, pp. 39–53. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2020/349700a860/1j2LfWz9VpC.

[234] Á. Kovács, "Comparison of different Linux containers," in *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, Jul. 2017, pp. 47–51.

[235] J. Krude and U. Meyer, "A Versatile Code Execution Isolation Framework with Security First," in *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, New York, NY, USA: ACM, 2013, pp. 1–10.

[236] T. Kulik *et al.*, "A Survey of Practical Formal Methods for Security," *Formal Aspects of Computing*, vol. 34, no. 1, 5:1–5:39, Jul. 2022.

[237] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, USA: IEEE Computer Society, Dec. 2003, p. 81.

[238] H. Kuzuno and T. Yamauchi, "Mitigating Foreshadow Side-channel Attack Using Dedicated Kernel Memory Mechanism," *Journal of Information Processing*, vol. 30, no. 0, pp. 796–806, Sep. 2022. [Online]. Available: https://www.jstage.jst.go.jp/article/ipsjjip/30/0/30_796/_article.

[239] A. B. Kvalsvik, P. Aimoniotis, S. Kaxiras, and M. Själander, "Doppelganger Loads: A Safe, Complexity-Effective Optimization for Secure Speculation Schemes," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 1–13.

[240] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[241] B. W. Lampson and H. E. Sturgis, "Reflections on an Operating System Design," *Commun. ACM*, vol. 19, no. 5, pp. 251–265, May 1976.

[242] M. Larabel, *Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower*, Nov. 2018. [Online]. Available: https://www.phoronix.com/review/linux-420-bisect.

[243] H. C. Lauer and C. R. Snow, "Is Supervisor-State Necessary?" In *Procedings of the ACM AICA International Computing Symposium*, Venice, Italy: University of Newcastle upon Tyne, Computing Laboratory, 1972.

[244] H. C. Lauer and D. Wyeth, "A recursive virtual machine architecture," ACM, Mar. 1973, pp. 113–116.

[245] A.-T. Le, B.-A. Dao, K. Suzaki, and C.-K. Pham, "Experiment on Replication of Side Channel Attack via Cache of RISC-V Berkeley Out-of-Order Machine (BOOM) Implemented on FPGA," in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Virtual, May 2020, p. 4.

[246] A.-T. Le, T.-T. Hoang, B.-A. Dao, A. Tsukamoto, K. Suzaki, and C.-K. Pham, "A cross-process Spectre attack via cache on RISC-V processor with trusted execution environment," *Computers and Electrical Engineering*, vol. 105, p. 108 546, Jan. 2023.

[247] Lee and Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan. 1984.

[248] J. Lee, J. Lee, T. Suh, and G. Koo, "CacheRewinder: Revoking speculative cache updates exploiting write-back buffer," in *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, Leuven, BEL: European Design and Automation Association, May 2022, pp. 514–519.

[249] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside {SGX} Enclaves with Branch Shadowing," 2017, pp. 557–574.

[250] H. Ponce-de Leon and J. Kinder, "Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 235–248.

[251] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Digital Press, 1984.

[252] C. Li and J.-L. Gaudiot, "Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, Jul. 2019, pp. 588–597.

[253] C. Li and J.-L. Gaudiot, "Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sep. 2018, pp. 25–28.

[254] P. Li, R. Hou, L. Zhao, Y. Zhu, and D. Meng, "Conditional address propagation: An efficient defense mechanism against transient execution attacks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 547–552.

[255] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 264–276.

[256] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan. 2010, pp. 1–12.

[257] T. Li and S. Parameswaran, "FaSe: Fast selective flushing to mitigate contention-based cache timing attacks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 541–546.

[258] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance Overhead Comparison between Hypervisor and Container Based Virtualization," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, Mar. 2017, pp. 955–962.

[259] A. Liguori, *QEMU 1.3.0 release*, Dec. 2012. [Online]. Available: `https://lists.gnu.org/archive/html/qemu-devel/2012-12/msg00123.html`.

[260] *Linux Containers - LXC - Introduction*, 2018. [Online]. Available: `https://linuxcontainers.org/lxc/introduction/`.

[261] S. B. Lipner *et al.*, "Security Kernels," in *Proceedings of the AFIPS National Computer Conference*, New York, NY, USA: ACM, 1974, pp. 973–980.

[262] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 813–825.

[263] M. Lipp *et al.*, "Meltdown," *arXiv:1801.01207 [cs]*, Jan. 2018.

[264] M. Lipp *et al.*, "Meltdown: Reading Kernel Memory from User Space," 2018, pp. 973–990.

[265] M. Lipp *et al.*, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 355–371.

[266] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 605–622.

[267] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Berkeley, CA, USA: USENIX Association, Jun. 2001, pp. 29–42.

[268] R. Lottiaux and C. Morin, "Containers: A sound basis for a true single system image," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2001, pp. 66–73.

[269] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, *A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks and Defenses in Cryptography*, Mar. 2021.

[270] K. Loughlin, I. Neal, and J. Ma, "DOLMA: Securing Speculation with the Principle of Transient Non-Observability," USENIX Association, Aug. 2021.

[271] A. Luzzardi, *Announcing Swarm 1.0: Production-ready clustering at any scale*, Nov. 2015. [Online]. Available: `https://blog.docker.com/2015/11/swarm-1-0/`.

[272] I. Madhu, R. Brandon, P. Samir, K. Vikram, A. Laurent, and F. Richard, *Enabling Security Formal Verification in ARM CPUs using SPV*, Oct. 2022.

[273] S. E. Madnick and J. J. Donovan, "Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation," in *Proceedings of the Workshop on Virtual Computer Systems*, New York, NY, USA: ACM, 1973, pp. 210–224.

[274] A. Magyar, D. Biancolin, J. Koenig, S. Seshia, J. Bachrach, and K. Asanović, "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[275] G. Maisuradze and C. Rossow, "Ret2spec: Speculative Execution Using Return Stack Buffers," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2109–2122, Oct. 2018.

[276] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Two methods for exploiting speculative control flow hijacks," Aug. 2019.

[277] F. Manco *et al.*, "My VM is Lighter (and Safer) Than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, New York, NY, USA: ACM, 2017, pp. 218–233.

[278] D. Margery, R. Lottiaux, and C. Morin, "Capabilities for per Process Tuning of Distributed Operating Systems," INRIA, Research Report RR-5411, 2004, p. 13.

[279] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – Vulnerability Analysis," *Computer Communications*, vol. 122, pp. 30–43, Jun. 2018.

[280] N. Mathure, S. K. Srinivasan, and K. K. Ponugoti, "A Refinement-Based Approach to Spectre Invulnerability Verification," *IEEE Access*, vol. 10, pp. 80 949–80 957, 2022.

[281] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of linux containers," in *2015 IEEE Conference on Communications and Network Security (CNS)*, Sep. 2015, pp. 559–567.

[282] J. N. Matthews *et al.*, "Quantifying the Performance Isolation Properties of Virtualization Systems," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, New York, NY, USA: ACM, 2007.

[283] A. J. W. Mayer, "The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?" *SIGARCH Comput. Archit. News*, vol. 10, no. 4, pp. 3–10, Jun. 1982.

[284] S. Mazor, "Intel's 8086," *IEEE Annals of the History of Computing*, vol. 32, no. 1, pp. 75–79, Jan. 2010.

[285] S. McFarling and J. Hennesey, "Reducing the cost of branches," in *Proceedings of the 13th annual international symposium on Computer architecture*, Washington, DC, USA: IEEE Computer Society Press, May 1986, pp. 396–403.

[286] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv:1902.05178 [cs]*, Feb. 2019.

[287] M. K. McKusick, M. J. Karels, K. Sklower, K. Fall, M. Teitelbaum, and K. Bostic, "Current Research by The Computer Systems Research Group of Berkeley," in *Proceedings of the European UNIX Users Group*, Brussels, Belgium, Apr. 1989.

[288] M. K. McKusick, "Twenty Years of Berkeley Unix - From AT&T-Owned to Freely Redistributable," in *Open Sources: Voices from the Open Source Revolution*, O'Reilly Media, Inc., Jan. 1999.

[289] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd Edition. Addison-Wesley Professional, Sep. 2014.

[290] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[291] A. Milburn, K. Sun, and H. Kawakami, *You Cannot Always Win the Race: Analyzing the LFENCE/JMP Mitigation for Branch Target Injection*, Mar. 2022.

[292] A. Miller and L. Chen, "An Exercise in Secure High Performance Virtual Containers," Las Vegas, NV, USA, Jul. 2012, p. 5.

[293] M. S. Miller, K.-P. Yee, and J. Shapiro, "Capability Myths Demolished," Johns Hopkins University, Systems Research Laboratory, Baltimore, Maryland, Technical Report SRL2003-02, 2003, p. 15.

[294] M. Minkin *et al.*, "Fallout: Reading Kernel Writes From User Space," *arXiv:1905.12701 [cs]*, May 2019.

[295] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, Aug. 2019.

[296] D. Moghimi, "Downfall: Exploiting Speculative Data Gathering," in *32nd USENIX Security Symposium (USENIX Security 23)*, USENIX Association, Aug. 2023.

[297] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *Proceedings of the 29th USENIX Conference on Security Symposium*, USA: USENIX Association, Aug. 2020, pp. 1427–1444.

[298] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*, Mar. 2015, pp. 386–393.

[299] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee, "Towards an efficient single system image cluster operating system," in *Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings.*, Oct. 2002, pp. 370–377.

[300] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic hardware-software contracts for security," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 72–86.

[301] *Nabla containers: A new approach to container isolation*, Aug. 2018. [Online]. Available: `https://nabla-containers.github.io/`.

[302] S. Nagar *et al.*, "Class-based Prioritized Resource Control in Linux," in *Proceedings of the Linux Symposium*, Ottowa, Canada, Jul. 2003, p. 21.

[303] S. Nanba, N. Ohno, H. Kubo, H. Morisue, T. Ohshima, and H. Yamagishi, "VM/4: ACOS-4 Virtual Machine Architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 171–178.

[304] S. Narayan *et al.*, "Swivel: Hardening WebAssembly against Spectre," in *Proceedings of the 30th USENIX Security Symposium*, USENIX Association, Aug. 2021, p. 19.

[305] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its protection system," in *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, Nov. 1977, pp. 1–10.

[306] R. A. Nelson, "Mapping Devices and the M44 Data Processing System," IBM Thomas J. Watson Research Center, Yorktown Heights, NY, Research Report RC-1303, 1964, p. 44.

[307] *NEMU - Modern Hypervisor for the Cloud.* Dec. 2018. [Online]. Available: `https://github.com/intel/nemu`.

[308] P. G. Neumann, "A Provably Secure Operating System: The system, its applications, and proofs," Computer Science Laboratory, SRI International, Tech. Rep., 1980.

[309] P. G. Neumann and R. J. Feiertag, "PSOS revisited," in *Proceedings of the 19th Annual Computer Security Applications Conference*, Dec. 2003, pp. 208–216.

[310] R. M. Norton, "Hardware support for compartmentalisation," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-887, May 2016, p. 86.

[311] M. Nosek and K. Szczypiorski, "An Evaluation of Meltdown Vulnerability," in *2022 9th international Conference on Wireless Communication and Sensor Networks (ICWCSN)*, New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 35–41.

[312] E. J. Ojogbo, M. Thottethodi, and T. N. Vijaykumar, "Secure automatic bounds checking: Prevention is simpler than cure," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, San Diego, CA, USA: Association for Computing Machinery, Feb. 2020, pp. 43–55.

[313] D. O'Keeffe *et al.*, *Spectre attack against SGX enclave*, Jan. 2018. [Online]. Available: `https://github.com/lsds/spectre-attack-sgx`.

[314] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box CPUs against speculation contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 226–239.

[315] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, *Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing*, Jan. 2023.

[316] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, *You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass*, Oct. 2018.

[317] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "{SpecFuzz}: Bringing Spectre-type vulnerabilities to the surface," 2020, pp. 1481–1498.

[318] A. Opler and N. Baird, "Multiprogramming: The Programmer's View," in *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery*, New York, NY, USA: ACM, 1959, pp. 1–4.

[319] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *Proceedings of the 5th Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[320] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed., Berlin, Heidelberg: Springer, 2006, pp. 1–20.

[321] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal*, vol. 11, no. 2, pp. 99–130, 1972.

[322] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, "Fast, Robust and Accurate Detection of Cache-Based Spectre Attack Phases," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 1–9.

[323] M. Patrignani and M. Guarnieri, "Exorcising Spectres with Secure Compilers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 445–461.

[324] D. A. Patterson and D. R. Ditzel, "The Case for the Reduced Instruction Set Computer," *SIGARCH Comput. Archit. News*, vol. 8, no. 6, pp. 25–33, Oct. 1980.

[325] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface*. Elsevier Science, May 2017. [Online]. Available: `https://books.google.com/books?id=H7wxDQAAQBAJ`.

[326] D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 443–457.

[327] H. Pearce, R. Karri, and B. Tan, "High-Level Approaches to Hardware Security: A Tutorial," *ACM Transactions on Embedded Computing Systems*, Jan. 2023.

[328] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, security threats, and solutions," *ACM Computing Surveys*, vol. 45, no. 2, pp. 1–39, Feb. 2013.

[329] N. Pemberton and A. Amid, "FireMarshal: Making HW/SW Co-Design Reproducible and Reliable," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2021, pp. 299–309.

[330] C. Percival, "Cache Missing for Fun and Profit," in *Proceedings of BSDCan 2005*, Ottowa, Canada, 2005, p. 13.

[331] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers," in *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, New York, NY, USA: ACM, 2013, pp. 3–10.

[332] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "{DRAMA}: Exploiting {DRAM} Addressing for {Cross-CPU} Attacks," 2016, pp. 565–581.

[333] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, "The Use of Name Spaces in Plan 9," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 2, pp. 72–76, Apr. 1993.

[334] G. Popek and C. Kline, "A verifiable protection system," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 294–304, Jun. 1975.

[335] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[336] J. Postel, "Internet Protocol," Internet Engineering Task Force (IETF), Defense Advanced Research Projects Agency (DARPA), Marina del Rey, California, Request for Comments 791, Sep. 1981, p. 51.

[337] D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," in *Proceedings of the 18th USENIX Conference on System Administration*, Berkeley, CA, USA: USENIX Association, 2004, pp. 241–254.

[338] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged Containers for User-defined Software Stacks in HPC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA: ACM, 2017, 36:1–36:10.

[339] *Propolis*, Dec. 2022. [Online]. Available: `https://github.com/oxidecomputer/propolis`.

[340] "Protection, Audit and Control Interfaces," IEEE, Draft POSIX Standard 1003.1e, Oct. 1997.

[341] A. Purnal, M. Bognar, F. Piessens, and I. Verbauwhede, "ShowTime: Amplifying Arbitrary CPU Timing Side Channels," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 205–217.

[342] P. Qiu *et al.*, "PMU-Leaker: Performance Monitor Unit-Based Realization of Cache Side-Channel Attacks," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 664–669.

[343] P. Qiu *et al.*, "PMU-Spill: A New Side Channel for Transient Execution Attacks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Aug. 2023.

[344] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards," in *Smart Card Programming and Security*, I. Attali and T. Jensen, Eds., Berlin, Heidelberg: Springer, 2001, pp. 200–210.

[345] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 775–787.

[346] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks across Cores Are Real," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 338–353.

[347] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 1451–1468.

[348] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, Nov. 2015, pp. 1–8.

[349] C. Rajapaksha, L. Delshadtehrani, M. Egele, and A. Joshi, "SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels," in *In Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Antwerp, Belgium, Apr. 2023.

[350] A. Randal, "Ghosting the spectre: Fine-grained control over speculative execution," University of Cambridge, Computer Laboratory, Tech. Rep., Dec. 2021.

[351] A. Randal, "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers," *ACM Computing Surveys*, vol. 53, no. 1, 5:1–5:31, Feb. 2020.

[352] A. Randal, "This is How You Lose the Transient Execution War," *arXiv:2309.03376 [cs]*, Sep. 2023.

[353] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM pointer authentication with speculative execution," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 685–698.

[354] C. Reis, *Mitigating Spectre with Site Isolation in Chrome*, Jul. 2018. [Online]. Available: `https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html`.

[355] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2021, pp. 361–374.

[356] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of OS-Level Virtualization Technologies," in *Secure IT Systems*, K. Bernsmed and S. Fischer-Hübner, Eds., Springer International Publishing, 2014, pp. 77–93.

[357] "Retpoline: A Branch Target Injection Mitigation," Intel Corporation, White Paper 337131-003, Jun. 2018, p. 22.

[358] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, no. 12, pp. 1405–1411, Dec. 1972.

[359] D. Ritchie, "The Evolution of the Unix Time-Sharing System," in *Proceedings of a Symposium on Language Design and Programming Methodology*, vol. 79, London, UK, UK: Springer-Verlag, 1980, pp. 25–36.

[360] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor," in *Proceedings of the 9th USENIX Security Symposium*, Denver, CO: USENIX Association, 2000, pp. 129–144.

[361] N. Rochester, "The Computer and Its Peripheral Equipment," in *Papers and Discussions Presented at the the November 7-9, 1955, Eastern Joint AIEE-IRE Computer Conference: Computers in Business and Industrial Systems*, New York, NY, USA: ACM, 1955, pp. 64–69.

[362] C. Rodrigues, D. Oliveira, and S. Pinto, "BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect," *To appear in IEEE Symposium on Security and Privacy 2024*, May 2024.

[363] S. Rokicki, "GhostBusters: Mitigating spectre attacks on a DBT-based processor," in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, Grenoble, France: EDA Consortium, Mar. 2020, pp. 927–932.

[364] S. Rokicki, E. Rohou, and S. Derrien, "Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1872–1885, Oct. 2019.

[365] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[366] M. Sabbagh, Y. Fei, and D. Kaeli, "Secure Speculative Execution via RISC-V Open Hardware Design," in *Proceedings of the Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*, Jun. 2021.

[367] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "Undo" Approach to Safe Speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 73–86.

[368] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander, "Ghost loads: What is the cost of invisible speculation?" In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 153–163.

[369] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction," in *Proceedings of the 46th International Symposium on Computer Architecture*, New York, NY, USA: ACM, Jun. 2019, pp. 723–735.

[370] C. Sakalis, S. Kaxiras, and M. Själander, "Delay-on-Squash: Stopping Microarchitectural Replay Attacks in Their Tracks," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, 9:1–9:24, Nov. 2022.

[371] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975.

[372] J. R. Sanchez Vicarte *et al.*, "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2021, pp. 347–360.

[373] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the Migration of Virtual Computers," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 377–390, Dec. 2002.

[374] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX Fails in Practice," Tech. Rep., 2020. [Online]. Available: `https://sgaxeattack.com/`.

[375] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions," *arXiv:2006.13353 [cs]*, Jun. 2020.

[376] S. van Schaik *et al.*, "RIDL: Rogue In-Flight Data Load," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 88–105.

[377] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: Characterization and optimization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 757–770.

[378] G. Schimunek, D. Dupuche, T. Fung, P. Kirkdale, E. Myhra, and H. Stein, *Slicing the AS/400 with Logical Partitioning: A How to Guide*. IBM Corporation, Aug. 1999.

[379] D. Schor, *ARM Neoverse N1 Microarchitecture*, 2019. [Online]. Available: `https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1`.

[380] D. Schor, *Intel Sunny Cove Microarchitecture*, 2020. [Online]. Available: `https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove`.

[381] M. Schwarz, C. Canella, L. Giner, and D. Gruss, *Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs (Updated and Extended Version)*, Mar. 2021.

[382] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, *ConTExT: Leakage-Free Transient Execution*, May 2019.

[383] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," *arXiv:1807.10535 [cs]*, Jul. 2018.

[384] M. Schwarz *et al.*, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 753–768.

[385] M. Schwarzl, C. Canella, D. Gruss, and M. Schwarz, "Specfuscator: Evaluating Branch Removal as a Spectre Mitigation," in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds., Berlin, Heidelberg: Springer, Oct. 2021, pp. 293–310.

[386] M. Schwarzl *et al.*, "Robust and Scalable Process Isolation Against Spectre in the Cloud," in *Computer Security – ESORICS 2022*, V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, Eds., Cham: Springer Nature Switzerland, 2022, pp. 167–186.

[387] M. Seddigh, M. Esfahani, S. Bhattacharya, M. R. Aref, and H. Soleimany, "Breaking KASLR on Mobile Devices without Any Use of Cache Memory," in *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 45–54.

[388] B. Semal, K. Markantonakis, R. N. Akram, and J. Kalbantner, "Leaky Controller: Cross-VM Memory Controller Covert Channel on Multi-core Systems," in *ICT Systems Security and Privacy Protection*, M. Hölbl, K. Rannenberg, and T. Welzer, Eds., Cham: Springer International Publishing, 2020, pp. 3–16.

[389] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, New York, NY, USA: ACM, 2007, pp. 335–350.

[390] A. Seznec, "A new case for the TAGE branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA: Association for Computing Machinery, Dec. 2011, pp. 117–127.

[391] Z. Shen *et al.*, "X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, Providence, RI, USA: ACM, Apr. 2019, p. 15.

[392] Z. Shen, J. Zhou, D. Ojha, and J. Criswell, *Restricting Control Flow During Speculative Execution with Venkman*, Mar. 2019.

[393] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 131–145.

[394] M. Silberstein, O. Oleksenko, C. Fetzer, and 2018, "Speculating about speculation: On the (lack of) security guarantees of Spectre-V1 mitigations," *ACM SIGARCH Computer Architecture News*, Jul. 2018. [Online]. Available: `https://www.sigarch.org/speculating-about-speculation-on-the-lack-of-security-guarantees-of-spectre-v1-mitigations/`.

[395] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling microarchitectural replay attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 318–331.

[396] D. Skarlatos, Z. N. Zhao, R. Paccagnella, C. W. Fletcher, and J. Torrellas, "Jamais vu: Thwarting microarchitectural replay attacks," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 1061–1076.

[397] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds., Berlin, Heidelberg: Springer, 2003, pp. 2–12.

[398] A. J. Smith, "Directions for memory hierarchies and their components: Research and development," in *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.*, Nov. 1978, pp. 704–709.

[399] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[400] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[401] "Software Techniques for Managing Speculation on AMD Processors," AMD, White Paper, May 2023, p. 9.

[402] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA: ACM, 2007, pp. 275–287.

[403] F. G. Soltis, *Fortress Rochester: The Inside Story of the IBM ISeries*. System iNetwork, 2001.

[404] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 800-190, Sep. 2017.

[405] "Speculation Behavior in AMD Micro-architectures," AMD, White Paper, May 2019.

[406] "Speculative Execution Side Channel Mitigations," Intel Corporation, Tech. Rep. 336996-003, Jul. 2018, p. 19.

[407] "Speculative Execution Side Channel Mitigations," Intel Corporation, Tech. Rep., May 2021. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html`.

[408] R. J. Srodawa and L. A. Bates, "An efficient virtual machine implementation," in *Proceedings of the workshop on virtual computer systems*, ACM, Mar. 1973, pp. 43–73.

[409] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," *arXiv:1806.07480 [cs]*, Jun. 2018.

[410] "Straight-line Speculation," ARM, Whitepaper Version 1.0, Jun. 2020.

[411] H. E. Sturgis, "A postmortem for a time sharing system," PhD Thesis, University of California at Berkeley, Berkeley, CA, Jun. 1973.

[412] A. Suda, *Allow running dockerd as a non-root user (Rootless mode)*, Feb. 2019. [Online]. Available: `https://github.com/moby/moby/pull/38050`.

[413] A. Suda and G. Scrivano, *Rootless Kubernetes*, Brussels, Belgium, Feb. 2019. [Online]. Available: `https://fosdem.org/2019/schedule/event/containers_k8s_rootless/`.

[414] S. Swanson, L. K. McDowell, M. M. Swift, S. J. Eggers, and H. M. Levy, "An evaluation of speculative instruction execution on simultaneous multithreaded processors," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 314–340, Aug. 2003.

[415] M. H. Syed and E. B. Fernandez, "A Reference Architecture for the Container Ecosystem," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, New York, NY, USA: ACM, 2018, 31:1–31:6.

[416] M. H. Syed and E. B. Fernandez, "The Container Manager Pattern," in *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, New York, NY, USA: ACM, 2017, 28:1–28:9.

[417] J. Szefer, "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, pp. 219–234, Sep. 2019.

[418] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the Hypervisor Attack Surface for a More Secure Cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, New York, NY, USA: ACM, 2011, pp. 401–412.

[419] L. Szekeres, M. Payer, Tao Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA: IEEE, May 2013, pp. 48–62.

[420] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management," 2017.

[421] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "SecSMT: Securing SMT Processors against Contention-Based Covert Channels," 2022, pp. 3165–3182.

[422] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA: Association for Computing Machinery, Apr. 2019, pp. 395–410.

[423] J. P. Thoma, J. Feldtkeller, M. Krausz, T. Güneysu, and D. J. Bernstein, "BasicBlocker: ISA Redesign to Make Spectre-Immune CPUs Faster," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 103–118.

[424] J. P. Thoma and T. Güneysu, "Write Me and I'll Tell You Secrets – Write-After-Write Effects On Intel CPUs," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 72–85.

[425] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2009, pp. 493–504.

[426] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 109–120, Mar. 2009.

[427] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, vol. C-19, no. 10, pp. 889–895, Oct. 1970.

[428] V. Tkachenko, *20-30% Performance Hit from the Spectre Bug Fix on Ubuntu*, Jan. 2018. [Online]. Available: `https://www.percona.com/blog/20-30-performance-hit-spectre-bug-fix-ubuntu/`.

[429] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 681–698.

[430] M. C. Tol, B. Gulmezoglu, K. Yurtseven, and B. Sunar, "FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, Sep. 2021, pp. 616–632.

[431] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[432] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, "Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 241–254.

[433] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 947–960.

[434] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing New Attack Surfaces with Training in Transient Execution," in *32nd USENIX Security Symposium (USENIX Security 23)*, USENIX Association, Aug. 2023.

[435] P.-A. Tsai, Y. L. Gan, and D. Sanchez, "Rethinking the Memory Hierarchy for Modern Languages," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 203–216.

[436] P. Turner, "Retpoline: A software construct for preventing branch-target-injection - Google Help," Google, Technical Report, 2018. [Online]. Available: `https://support.google.com/faqs/answer/7625886`.

[437] G. Tyson and M. Farrens, "Evaluating the Effects of Predicated Execution on Branch Prediction," *International Journal of Parallel Programming*, vol. 24, no. 2, pp. 159–186, Apr. 1996.

[438] *User_namespaces(7) man page*, Feb. 2018. [Online]. Available: `http://man7.org/linux/man-pages/man7/user_namespaces.7.html`.

[439] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: ACM, 2018, pp. 178–195.

[440] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets without Page Faults: Stealthy Page {Table-Based} Attacks on Enclaved Execution," 2017, pp. 1041–1056.

[441] J. Van Bulck *et al.*, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 991–1008.

[442] J. Van Bulck *et al.*, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 54–72.

[443] M. Varian, "VM and the VM Community: Past, Present, and Future," in *SHARE '89*, 1989, p. 70.

[444] M. Vassena *et al.*, "Automatically eliminating speculative leaks from cryptographic code with blade," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 49:1–49:30, Jan. 2021.

[445] A. Vasudevan, J. M. McCune, N. Qu, L. Van Doorn, and A. Perrig, "Requirements for an Integrity-protected Hypervisor on the x86 Hardware Virtualized Architecture," in *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, Berlin, Heidelberg: Springer-Verlag, 2010, pp. 141–165.

[446] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, New York, NY, USA: ACM, 2015, 18:1–18:17.

[447] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: Mitigating Branch Predictor Side-Channels," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 466–477.

[448] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[449] J. Wampler, I. Martiny, and E. Wustrow, "ExSpectre: Hiding Malware in Speculative Execution," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019.

[450] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 3, 14:1–14:31, Jun. 2020.

[451] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "Oo7: Low-overhead Defense against Spectre Attacks via Program Analysis," *arXiv:1807.05843 [cs]*, Nov. 2019.

[452] K. Wang, X. Qin, Z. Yang, W. He, Y. Liu, and J. Han, "SVP: Safe and Efficient Speculative Execution Mechanism through Value Prediction," in *Proceedings of the Great Lakes Symposium on VLSI 2023*, New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 433–437.

[453] W. Wang *et al.*, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 2421–2434.

[454] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning Power {Side-Channel} Attacks Into Remote Timing Attacks on x86," 2022, pp. 679–697.

[455] Z. Wang and R. Lee, "Covert and Side Channels Due to Processor Architecture," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Miami Beach, FL, USA: IEEE, Dec. 2006, pp. 473–482.

[456] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating Commodity Hosted Hypervisors with HyperLock," in *Proceedings of the 7th ACM European Conference on Computer Systems*, New York, NY, USA: ACM, 2012, pp. 127–140.

[457] R. N. M. Watson *et al.*, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 20–37.

[458] R. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium*, vol. 19, Washington, DC, USA: ACM Press, Aug. 2010.

[459] R. Watson *et al.*, "CHERI: A research platform deconflating hardware virtualization and protection," in *Unpublished workshop paper for RESoLVE'12*, London, UK, Mar. 2012.

[460] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "A Taste of Capsicum: Practical Capabilities for UNIX," *Communications of the ACM*, vol. 55, no. 3, pp. 97–104, Mar. 2012.

[461] R. N. M. Watson *et al.*, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-864, Dec. 2014.

[462] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated Discovery of Microarchitectural Side Channels," 2021, pp. 1415–1432.

[463] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture - MICRO '52*, Columbus, OH, USA: ACM Press, 2019, pp. 572–586.

[464] O. Weisse *et al.*, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," Tech. Rep., Aug. 2018, p. 7.

[465] A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," University of Washington, Tech. Rep., 2002.

[466] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: A Scalable Isolation Kernel," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, New York, NY, USA: ACM, 2002, pp. 10–15.

[467] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 195–209, Dec. 2002.

[468] J. Wikner, C. Giuffrida, V. Amsterdam, and K. Razavi, "Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks," San Francisco, CA, USA: IEEE, May 2022.

[469] J. Wikner and K. Razavi, "RETBLEED: Arbitrary Speculative Code Execution with Return Instructions," 2022, pp. 3825–3842.

[470] M. V Wilkes, *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. Amsterdam, The Netherlands: North-Holland Publishing Co., 1979.

[471] M. V. Wilkes and D. W. Willis, "A magnetic-tape auxiliary storage system for the EDSAC," *Proceedings of the IEE - Part B: Radio and Electronic Engineering*, vol. 103, no. 2, pp. 337–345, Apr. 1956.

[472] M. V. Wilkes, *Time-Sharing Computer Systems*, 2nd ed. MacDonald & Co., 1968.

[473] D. Williams and R. Koller, "Unikernel Monitors: Extending Minimalism Outside of the Box," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO: USENIX Association, 2016, p. 6.

[474] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels As Processes," in *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA: ACM, 2018, pp. 199–211.

[475] D. Williams, R. Koller, and B. Lum, "Say Goodbye to Virtualization for a Safer Cloud," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA: USENIX Association, 2018.

[476] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, "Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, Feb. 2021, pp. 627–632.

[477] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, "Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core," in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Valencia, Spain, May 2020, p. 7.

[478] N. Wistoff, M. Schneider, F. K. Gürkaynak, G. Heiser, and L. Benini, *Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning*, Feb. 2022.

[479] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A Survey on Assertion-based Hardware Verification," *ACM Computing Surveys*, vol. 54, no. 11s, 225:1–225:33, Sep. 2022.

[480] H. Witharana and P. Mishra, "Speculative Load Forwarding Attack on Modern Processors," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 1–9.

[481] J. Woodruff *et al.*, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[482] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Module Framework," in *Proceedings of the Ottawa Linux Symposium*, Ottowa, Canada, Jun. 2002, pp. 604–617.

[483] P. Wright, *Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer*. Viking Press, 1987.

[484] Y. Wu and X. Qian, *RCP: A Low-overhead Reversible Coherence Protocol*, Jul. 2022.

[485] Y. Wu and X. Qian, *ReversiSpec: Reversible Coherence Protocol for Defending Transient Attacks*, Jun. 2020.

[486] W. Wulf *et al.*, "HYDRA: The Kernel of a Multiprocessor Operating System," *Commun. ACM*, vol. 17, no. 6, pp. 337–345, Jun. 1974.

[487] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA-C. Mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[488] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 233–240.

[489] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities," in *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, Feb. 2020.

[490] Q. Xu, H. Naghibijouybari, S. Wang, N. Abu-Ghazaleh, and M. Annavaram, "GPUGuard: Mitigating contention based side and covert channel attacks on GPUs," in *Proceedings of the ACM International Conference on Supercomputing*, New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 497–509.

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

[492] Y. Yang, S. Lau, T. Bourgeat, and M. Yan, "Pensieve: Microarchitectural Modeling for Security Evaluation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, Orlando, FL, USA: ACM, Jun. 2023.

[493] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732.

[494] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing," in *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2019. [Online]. Available: `https://www.ndss-symposium.org/ndss-paper/data-oblivious-isa-extensions-for-side-channel-resistant-and-high-performance-computing/`.

[495] J. Yu, T. Jaeger, and C. W. Fletcher, "All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 1–14.

[496] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, Virtual Event: IEEE Press, Sep. 2020, pp. 707–720.

[497] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. 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*, New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 954–968.

[498] Y. Zhai, L. Yin, J. Chase, T. Ristenpart, and M. Swift, "CQSTR: Securing Cross-Tenant Applications with Cloud Containers," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, New York, NY, USA: ACM, 2016, pp. 223–236.

[499] Z. Zhang, Y. Cheng, and S. Nepal, *GhostKnight: Breaching Data Integrity via Speculative Execution*, Jan. 2022.

[500] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: Taking Speculative Load Hardening to the Next Level," Anaheim, CA, USA: USENIX Association, Aug. 2023.

[501] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Valencia, Spain, May 2020.

[502] Z. N. Zhao, H. Ji, A. Morrison, D. Marinov, and J. 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*, New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 314–328.

[503] Z. N. Zhao *et al.*, "Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020, pp. 1138–1152.