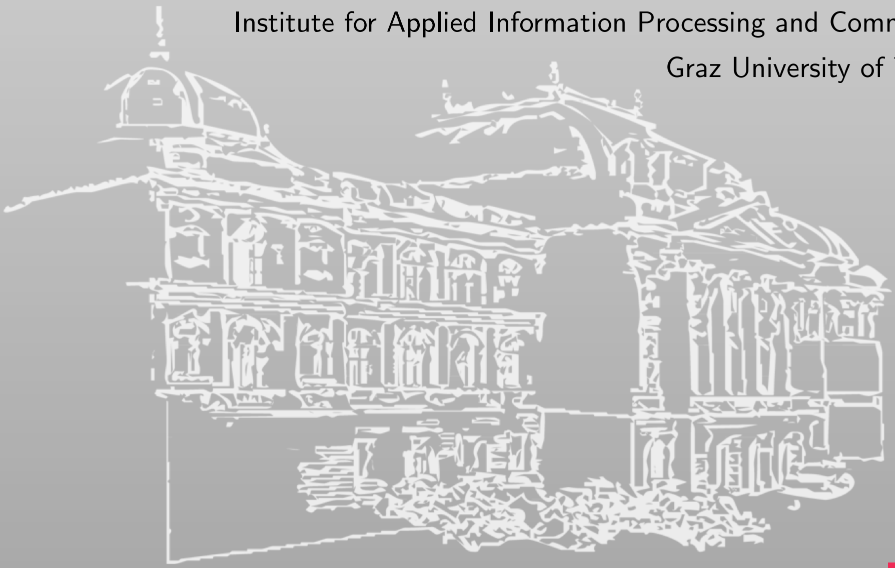
Claudio Canella

Hardening the Kernel Against Unprivileged Attacks

PhD Thesis

Assessors: Daniel Gruss, Frank Piessens

October 2022



Institute for Applied Information Processing and Communications
Graz University of Technology

SCIENCE ▪ PASSION ▪ TECHNOLOGY

Affidavit

Date	Signature

Abstract

Modern computer systems play a significant role in our everyday life. An essential part of these systems is the privileged Operating System (OS), which performs specific tasks for less-privileged applications. However, these higher privileges make it an interesting target for unprivileged attackers. Examples of potential threats are microarchitectural and control-flow-hijacking attacks. With the former, an attacker exploits effects introduced by the actual hardware implementation of an architecture to steal data that would normally be inaccessible. By design, these attacks do not rely on software bugs. With the latter, an attacker exploits a bug to gain control of an unprivileged application. A subsequent goal can be to open a remote shell for additional attacks that try to gain higher privileges or steal data. Typically, this requires the hijacked application to interact with the OS.

This thesis focuses on hardening the OS against such attacks. We show that existing defenses against microarchitectural attacks are insufficient or cause additional security problems. To address the latter, we propose a new software-based defense that fixes the additional security problem with no runtime overhead. We then reduce the attack surface the OS exposes to potentially hijacked applications by restricting their access to the syscall interface. We first discuss 2 automated solutions to achieve this restriction. Our first solution solely restricts access based on whether the application requires the syscall. The second solution more strictly restricts access by checking whether the syscall is part of a pre-determined sequence and originates from the correct syscall instruction. Finally, we propose a concept that allows restricting syscalls based on complex arguments, e.g., strings, which is impossible with state-of-the-art solutions due to time-of-check-time-of-use vulnerabilities.

The first part of this thesis discusses its contributions and provides the necessary background. It also discusses the current state-of-the-art of software-based microarchitectural attacks and control-flow-hijacking attacks and defenses. The second part consists of a selection of my publications.

Acknowledgments

First and foremost, I would like to thank my advisor Daniel Gruss for introducing me to the field of microarchitectural attacks and giving me the opportunity to pursue a PhD. Without your help, I would not be as confident as I am now when it comes to presenting my work.

Similarly, I want to thank Michael Schwarz for helping me throughout my PhD, for providing ideas at the start, and for the continuous support afterward when I pitched a new (potentially stupid) idea. I wish you all the best and a lot of success with your own research group. Your students can be thankful to have such a supporting supervisor.

I want to thank Frank Piessens for his valuable feedback and for taking the time and effort to assess my thesis.

During my PhD, I had the chance to work with incredible people at the institute. I want to thank the other (active and former) members of the group: Moritz Lipp, Martin Schwarzl, Lukas Giner, Andreas Kogler, Catherine Easdon, Stefan Gast, and Jonas Juffinger. I also want to thank all the other members of the institute for a wonderful time, especially Mario Werner, as I also had the pleasure of working with you on a paper that is part of this thesis. Special thanks to everyone on the administrative side of the institute and our sysadmins. You always provided a helping hand when it was needed. Unfortunately, the pandemic cut short all our time together in the office. However, we still had a lot of fun in online meetings, game nights, and the occasional coffee break in the kitchen. Thanks for the interesting discussions and talks, whether they were related to work or something else. Without all of you, my PhD would not have been the same, and I hope the friendships we built last a lifetime.

Over the years, I also had the chance to meet and work with amazing people from all over the world. I want to thank Jo Van Bulck for his help on my first research paper. Your knowledge of the field and guidance in writing the paper has helped me throughout my PhD. Likewise, I want to thank Daniel Moghimi, Berk Sunar, Khaled Khasawneh, Dmitry Evtyushkin, and Tianyin Xu for great discussions and fruitful collaborations.

I also owe a debt of gratitude to the students who contributed to several of my papers. I want to thank Sebastian Dorn, Benjamin von Berg, Philipp Ortner, and Martin Haubenwallner. Supervising your thesis and working with you on exciting research projects was a pleasure.

Special thanks is also necessary for the people who first got me interested in computer science, recognized my talent for it, and supported me while still at school and beyond. I want to thank Gerhard Lederwasch for teaching me my first programming language, which ignited a passion that is still going strong. I also want to thank Christoph Jarosch for all the exciting discussions about computer science and for furthering my passion for it. To Christian Gattermair, for teaching me many new things and supporting me in my pursuit of a job in the industry right after graduating from school. I would have never pursued a PhD in this field if it were not for you and your early support.

I want to thank the most important people in my life: my family and friends. To my mother, Elisabeth Canella, for giving me all the encouragement, love, and support in the world. You are a true role model for me. To my father, Alberto Canella, for his continuing support even when things progressed slowly at times. To my best friends, Marco Carli and Vanessa Huber, for keeping me sane during all my time at the university. To my friends, Markus Schofnegger, Roman Walch, and Patrick Radkohl for listening to me rant when things did not go as planned.

Finally, I want to thank my wonderful girlfriend, Fiona. Even though we met late in my PhD, your support made this thesis possible. Thanks for providing a comforting shoulder at the end of a hard day. For your understanding when I work way too late and do not have as much time for you as I would like to have.

Table of Contents

Part I

Hardening the Kernel Against Unprivileged Attacks

1 Introduction and Contribution

Computers are an essential part of our everyday life, whether for entertainment, health care, travel, or educational purposes. While the applications from these categories are developed for different purposes, they have at least one commonality: they all rely on the more privileged operating system (OS), *i.e.*, the kernel, to perform specific tasks on their behalf. These tasks are manifold: creating new processes and threads, managing the application's memory, handling input and output, and interacting with the underlying hardware. This makes the kernel an interesting target for malicious actors, as subverting it allows them to, for instance, steal or manipulate the data of every other application. While an attacker can rely on many different methods to undermine the kernel, we only focus on two of them: microarchitectural and control-flow-hijacking attacks.

Microarchitectural attacks exploit effects introduced by the actual implementation (microarchitecture) of an architecture. Typical examples of such effects are the differences in runtime or the amount of power consumed. Kocher [165] was the first to mention how caches and their effect on the runtime can be used to break cryptographic algorithms. While subsequent works have demonstrated that these effects can be used to spy on user behavior [122, 182, 209, 313, 341] or break cryptographic algorithms in various ways [15, 24, 46, 92, 114, 122, 125, 127, 139, 148–150, 235, 239, 327, 328, 340], they also affect the security of the kernel. For instance, multiple microarchitectural attacks have been demonstrated that break Kernel Address Space Layout Randomization (KASLR) [50, 53, 85, 118, 134, 155, 163, 170, 181, 183, 184, 264, 318], weakening the kernel's protection against subsequent attacks. However, the most famous examples of microarchitectural attacks that impact the security of the kernel are two transient-execution attacks: Meltdown [185] and Spectre [166]. With Meltdown, an attacker can circumvent the boundary between kernel space and userspace, reading arbitrary memory without having to exploit a bug in the kernel or its interface. Spectre, on the other hand, relies on branch prediction to leak otherwise inaccessible data.

However, these microarchitectural attacks are comparatively new. The predominant way to subvert the system is still via traditional memory safety violations [289] that result in control-flow-hijacking attacks. The attacker exploits a bug in an application running on the victim machine to gain control of the application. This allows them to perform arbitrary tasks with the privileges of the victim application. Typically, an attacker tries to open a remote shell to make subsequent attack steps easier, e.g., by providing additional attack binaries that are then launched on the victim machine. However, the hijacked application must interact with the operating system to perform the necessary tasks.

In this thesis, we harden the kernel against unprivileged attackers performing microarchitectural or control-flow-hijacking attacks. We systematically investigate the field of transient-execution attacks to determine their impact on the kernel, discover new attack variants, and demonstrate that the majority of the proposed defenses do not mitigate these attacks. The reason for this is that they only focus on the cache as a transmission channel, making them easy to circumvent. We even demonstrate that an incomplete hardware mitigation for Meltdown allows an attacker with local code execution to break KASLR, a defense that attempts to make attacks on the kernel harder. Our attack achieves this with perfect accuracy and in a fraction of a second. As the attack exploits the behavior of the hardware, it requires a software solution to fix the issue on existing hardware. We propose and demonstrate such a mitigation that prevents not only our newly found attack but also other ones that rely on hardware behavior to break KASLR [53, 118, 134, 155, 181, 183, 184, 264, 318].

We further demonstrate how the kernel can be hardened by limiting the attack surface that is exposed to hijacked applications. We achieve this by restricting an initially benign but hijacked application's access to the syscall interface. While the Linux kernel already provides a feature to restrict access in the form of Linux Secure Computing (seccomp) [81], it has several shortcomings that limit its effectiveness. First, it is not used by a majority of the available applications. This is due to the significant amount of manual work required as all necessary syscalls must be manually identified. Second, while seccomp can restrict access to the syscall interface based on simple syscall arguments, it cannot do the same for complex arguments like strings or structs due to time-of-check-time-of-use vulnerabilities (TOCTOU) [201]. However, especially syscalls that use such complex arguments, like the *exec* syscalls, are of interest to an attacker as they can severely impact the system's security. Finally, seccomp considers every syscall individually and cannot restrict a syscall based on its predecessor. In this thesis, we propose solutions for all three shortcomings. We demonstrate a novel system that automatically generates the seccomp filters of an application, eliminating all manual effort while drastically reducing the exposed attack surface of the kernel. We show that dynamic and time-restricted isolation enables checks on complex syscall arguments without being susceptible to TOCTOU vulnerabilities. Finally, we propose a novel concept that relies on an application's syscall flow and the origin of syscalls in determining whether access should be restricted or not. In all three cases, we significantly harden the kernel by restricting an attacker's access to the syscall interface better than state-of-the-art solutions.

1.1 Main Contributions

As the field of transient-execution attacks was still in its infancy, the impact on the kernel was largely unknown. We started with the assumption that the field has only been sparsely explored and that a systematic evaluation might reveal new attack variants. Additionally, we assumed that existing defenses were not sufficient. We discovered several new Meltdown variants by performing a systematic analysis, including the first affecting AMD processors. Furthermore, we showed that multiple predictors can be mistrained within and across address spaces, even with congruent addresses. On the defensive side, we demonstrated that most defenses do not mitigate transient-execution attacks. We outlined that new defenses must focus on mitigating the root cause instead of simply focusing on the transmission channel, *i.e.*, the cache. The paper was accepted at USENIX Security Symposium 2019 in collaboration with Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss [51].

Following other works in the field of transient-execution attacks, we targeted the Meltdown hardware mitigations Intel implemented in newly released CPUs. We hypothesized that loads from kernel addresses are still executed but that the loaded value is zeroed out before it is passed on to dependent instructions. We reverse-engineered the hardware fixes using several performance counters, indicating that our hypothesis was correct. This newly gained information led to the discovery of a new microarchitectural KASLR break, called EchoLoad, that allows an attacker to de-randomize the location of the kernel without any false positives. We also demonstrated a Meltdown attack from JavaScript on unpatched x86 CPUs. However, the paper's main contribution was on the defensive side. We analyzed the commonalities of previous microarchitectural KASLR breaks, which resulted in a mitigation called FLARE. A kernel that employs FLARE is protected against all of the analyzed microarchitectural KASLR breaks. The paper was accepted at AsiaCCS 2020 in collaboration with Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss [50].

A year after our systematic evaluation of transient-execution attacks and defense paper [51], we again investigated the whole field of transient-execution attacks as additional attacks had been published. Instead of focusing on differences in the attacks in previous work [51], we focused on their commonalities. We showed that for Meltdown-type attacks, the attacks can be grouped into three groups: deferred permission check, use of intermediate values, and use-after-free. In the second paper, we analyzed defenses, focusing on the differences in proposed defenses by academia and industry, highlighting that academia proposes more radical defenses. We also discussed that many of the proposed defenses still do not mitigate the root cause. Both papers were published at GLSVLSI 2020, with the first being a collaboration with Khaled N. Khasawneh and Daniel Gruss [48]. The second was a collaboration with Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh [49].

While microarchitectural attacks impact the kernel's security, more traditional attacks, such as control-flow-hijacking attacks, have more commonly threatened the kernel and the system's overall security. Hence, we shifted the focus from hardening the kernel against microarchitectural attacks to reducing the attack surface the kernel exposes to unprivileged applications. Typically, this is achieved by employing sandboxing techniques that limit access to a specific resource. The resource of interest that the kernel exposes to unprivileged userspace applications are syscalls. Linux provides the Secure Computing (seccomp) interface [81] to limit an application's access to the syscall interface. However, seccomp requires manual identification of all the syscalls an application requires, which limits the usage of seccomp in real-world applications. We wanted to automate this process by identifying these syscalls using static analysis either during the compilation of the application or by analyzing an existing binary. The resulting tool, called Chestnut, consists of a compiler extension, a binary analysis tool, and a dynamic refinement tool. We demonstrated that applications that have access to the syscall interface restricted by Chestnut are much less likely to threaten the kernel. The paper was accepted at ACM CCSW 2021, was awarded the best paper award, and was done in collaboration with Mario Werner, Daniel Gruss, and Michael Schwarz [52].

Isolation of different security domains is a critical task, especially in modern systems. Examples of such security domains are the kernel and userspace or the trusted and untrusted part in SGX. Previous work proposed various solutions to isolate security domains from each other [23, 31, 59, 71, 90, 129, 133, 167, 186, 188, 199, 207, 229, 260, 275, 299, 310, 319]. However, these approaches either require code instrumentation, newly developed hardware features or existing but not yet readily available ones. We wanted to develop an approach that runs on commercial off-the-shelf systems. With Domain Page-Table Isolation (DPTI), we proposed a concept for time-limited changes to the memory isolation that can be dynamically applied at security-critical points. The enforcement is done purely by the existing isolation provided by the MMU. DPTI modifies an application's page tables so that only one domain can access a page while it is being used. We demonstrated DPTI in two use cases: deep argument (string) filtering of syscall arguments and isolating the host from a potentially malicious SGX enclave. The former allows us to safely check strings used as syscall arguments, which is impossible with seccomp. The latter prevents a potentially malicious enclave from exploiting the host application and, through it, the kernel with a lower overhead than prior work [319]. The paper is available as a pre-print and was done in collaboration with Andreas Kogler, Lukas Giner, Daniel Gruss, and Michael Schwarz [54].

Control-flow integrity [1] (CFI) is a countermeasure that restricts control-flow transfers to valid targets. By design, CFI is only applicable within the same security domain, *i.e.*, CFI typically does not provide integrity to user-kernel transfers. However, seccomp [81] enables cross-domain checks, but due to the stateless design, it can neither constrain the syscall flow nor provide sufficient integrity to user-kernel transfers. Hence, we propose the concept of coarse-grained syscall-flow-integrity protection (SFIP). Coarse-grained SFIP relies on three pillars: syscall sequences that model the syscall flow of the application, syscall origins that identify locations from which a syscall can originate from, and the enforcement of the information by the kernel. We implemented a proof-of-concept of SFIP, called SysFlow, which extracts the necessary information using static analysis during the compilation of an application. Enforcing this information imposes more significant constraints on the syscall execution than in seccomp, further limiting an attacker's interaction with the kernel through a hijacked application. The paper is available as a pre-print and was done in collaboration with Sebastian Dorn, Daniel Gruss, and Michael Schwarz [47].

1.2 Other Contributions

While working on our systematic analysis of transient-execution attacks and defenses [51], we discovered that most proposed defenses do not protect against such attacks by eliminating the root cause. Instead, they focus on other phases of the attack [48, 51, 324]. Therefore, attacks are still possible by merely modifying the attack. With ConTExT [267], we targeted the actual root cause of Spectre-type attacks. ConTExT consists of two different parts, full ConTExT and ConTExT-light, that rely on annotating secret data in the source code. Such annotated variables are grouped on a page by the compiler and marked as secure. ConTExT-light re-uses existing parts that allow mitigating Spectre-type attacks on existing hardware while full ConTExT requires hardware changes. The paper was accepted at NDSS 2020 and was done in collaboration with Michael Schwarz, Moritz Lipp, Robert Schilling, Florian Kargl, and Daniel Gruss [267].

Based on the systematization paper [51], we investigated different buffers that could be exploited with transient-execution attacks. One such buffer is the store buffer, which we subsequently exploited in Store-to-Leak Forwarding [264] and Fallout [53]. While both attacks target the same microarchitectural element, they significantly differ in the parts they exploit in the store-to-load forwarding logic. In Store-to-Leak Forwarding, we exploited a combination of the true positive and true negative match in the store-to-load forwarding logic. Fallout, on the other hand, exploited the false positive match in the store-to-load forwarding logic. The partial address match results in the incorrect data being forwarded to the register from where it is then used in further computations. Both papers were submitted to CCS 2019 and have been merged as per the program committee's request. Store-to-Leak Forwarding was done in collaboration with

Michael Schwarz, Lukas Giner, and Daniel Gruss. Fallout was originally discovered by Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. The merged CCS 2019 paper was then a collaboration with Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom [53].

Since the discovery of Spectre [166], academia and industry have proposed various countermeasures that try to address the problems raised by speculative execution. However, none of these approaches considered removing all conditional and indirect branches. We proposed a defense that linearizes the entire program flow using a new compiler pass. While the countermeasure is indeed effective in mitigating Spectre attacks that exploit control-flow mispredictions [166, 168, 192], we observed performance overheads of up to factor 1000. The paper was accepted at FC 2021 and was done in collaboration with Martin Schwarzl, Daniel Gruss, and Michael Schwarz [272].

Power side-channel attacks have been known since the mid-90s, but require direct access to the hardware and expensive equipment. Modern CPUs include power management functionality to adapt processing power based on usage and thermal constraints. Unfortunately, the kernel exposes this information to userspace, allowing an attacker to exploit this information for attacks on cryptography in a correlation power analysis attack (CPA). Furthermore, a privileged attacker can exploit this information directly in the kernel to attack SGX, which is allowed under its threat model. The paper was accepted at S&P 2021 and was done in collaboration with Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, and Daniel Gruss [184].

One of the unknowns when it comes to Spectre attacks is the security of various programming languages and their execution enviroments. To provide insights into this field, we evaluated various execution environments and showed that 26 out of 42 do not contain a mitigation against any variant of Spectre attack. We then developed a tool, called Speconnector, that helps in developing proof-of-concept Spectre attacks. Using Speconnector, we implemented several Spectre attacks in various programming languages. Additionally, we also implemented Spectre attacks on code generated by various execution environments. Our results indicate that even years after the disclosure of Spectre, several programming languages and execution enviroments still remain vulnerable. The paper was accepted at ICISSP 2022 and was done in collaboration with Amir Naseredini, Stefan Gast, Martin Schwarzl, Pedro Miguel Sousa Bernardo, Amel Smajic, Martin Berger, and Daniel Gruss [215]

After the publication of Load-Value Injection (LVI) [303], we investigated how one of the proposed attacks, LVI-NULL, can be mitigated in the SGX scenario without too high of an overhead. Our proposed defense, called LVI-NULLify, repurposes segmentation to place memory locations exploited by LVI-NULL under control of the enclave. We achieve this through a modified SGX-SDK and a compiler extension. The paper was accepted at USENIX Security Symposium 2022 and was done in collaboration with Lukas Giner, Daniel Gruss, and Michael Schwarz [103].

In previous work, we focused on automating seccomp filter generation [52] as this normally requires tedious manual effort. The filters that are used by seccomp are expressed in the classic Berkeley Packet Filter [200], which imposes some restrictions on the expressiveness of the seccomp filter. For instance, cBPF is stateless. This statelessness allows seccomp to only perform very simple checks, such as whether the current syscall is allowed. On the other hand, the extended Berkeley Packet Filter allows for more expressiv filters. We hence proposed to support eBPF in seccomp alongside the classic cBPF. By using eBPF, application developers can perform additional checks, such as whether the current syscall is valid in the context of the previous one [47], syscall count limiting, or syscall serialization. The paper was done in collaboration with Jinghao Jia, YiFei Zhu, Yicheng Lu, Hsuan-Chi Kuo, Andrea Arcangeli, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu.

1.3 Outline

The structure of this thesis is as follows. Chapter 2 provides background on essential tasks of operating systems and briefly discusses the idea of sandboxing and isolation. It also discusses the ISA and microarchitecture of modern CPUs and the concept of side channels and microarchitectural attacks. Chapter 3 gives an overview of the state of the art in software-based microarchitectural attacks and mitigations for control-flow-hijacking attacks. In Chapter 4, we conclude our work and provide an outlook on potential future research.

2 Background

This chapter provides the necessary background for this thesis. In Section 2.1, we discuss operating systems (OS), focusing on the concept of virtual memory (Section 2.1.1), and how unprivileged userspace applications can interact with the privileged OS (Section 2.1.2). Afterward, we briefly discuss sandboxing and isolation (Section 2.2). Section 2.3 covers the microarchitectural level, discussing the instruction-set architecture (ISA) and the microarchitecture. In Section 2.3.1, we discuss caches as they are frequently abused in microarchitectural attacks. We discuss how the cache is organized and how it manages the data. Sections 2.3.2 and 2.3.3 discuss other microarchitectural optimizations in the form of instruction pipelining and speculative execution, respectively. Finally, in Section 2.4, we provide a definition of side channels (Section 2.4.1) and discuss the general idea of microarchitectural side-channel attacks that use them to obtain sensitive information (Section 2.4.2).

2.1 Operating Systems

Operating systems are a central piece of our everyday computing infrastructure as they handle several tasks that are essential for a well-functioning system. For instance, the OS is responsible for handling input and output, managing processes and their memory, and directly interacting with the

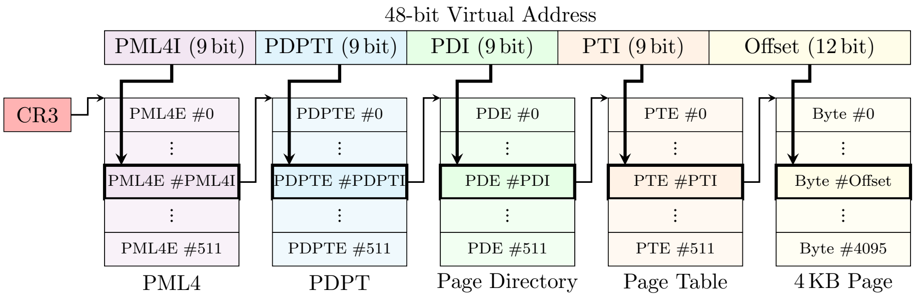


Figure 2.1: The 4-level paging mechanism on the x86_64 architecture [180, 263]. The `CR3` register points to the start of the first level, subsequent levels are determined by blocks of bits of the virtual address that serve as the index to the respective page table.

underlying hardware. The higher privilege level of the kernel makes it a high-value target for unprivileged attackers.

For this thesis, we limit our discussion of operating systems to the parts that are strictly necessary, *i.e.*, the concept of virtual memory and the system-call interface. Other topics, like interrupt handling and scheduling, are out of scope. For a more in-depth discussion of these topics, we refer the reader to the literature [290].

2.1.1 Virtual Memory

Modern systems can run multiple processes simultaneously. To prevent different processes from interfering with each other, the system assigns each process its own virtual address space. Therefore, processes do not directly operate on physical addresses but on virtual addresses. However, the system must still facilitate a mapping between virtual and physical addresses, *i.e.*, the underlying physical memory. This task is delegated to the processor, who performs this task via multi-level page tables defined by the OS for each process. This memory management scheme is typically referred to as paging.

With paging, the memory is managed at the granularity of a memory page, whose size is defined by the architecture. The most common page size is 4 kB. While page tables are primarily necessary for translating virtual to physical addresses, they also store additional information, such as permissions for each page.

In this thesis, we primarily focus on 64-bit x86 CPUs, which use a 64-bit sign-extended virtual address. However, only 48 bits are effectively used together with 4 levels of page tables to translate a virtual to a physical address. Newer processors have increased the number of used bits to 57 bits and 5 levels of page tables.

We illustrate the page-table hierarchy in Figure 2.1. The top level of the page-table hierarchy, the Page-Map Level 4 (PML4), is referenced by the register `CR3`. To support multi-processing, this register is changed upon a context switch to point to the top of the hierarchy for the newly scheduled process. The PML4 consists of 512 PML4 entries, each entry being 64 b wide and containing a flag that indicates whether the respective entry is valid. Such valid entries refer to the next level in the hierarchy, the Page-Directory Pointer Table (PDPT). To determine the correct entry in the PML4 for the virtual address being translated, the processor uses bits 39 to 47 as a lookup index into the PML4. While the PDPT largely follows the same structure as the PML4, there is one key difference. Starting at this level, an additional flag in each entry indicates whether the entry directly refers to a 1 GB range of physical memory or the next level in the hierarchy. The entry is again determined by a subset of bits from the virtual address, *i.e.*, bits 30 to 38. If the entry points directly to physical memory, the remaining 30 bits of the virtual address are used as an offset into the physical page. In the other case, bits 21 to 29 are used to look up the entry in the next level, the Page Directory (PD). There, the respective entry can either point to a 2 MB page of physical memory or the next level, the Page Table (PT). In the former, the remaining 21 bits are used as an offset into the page. If it points to the PT, bits 12 to 20 are used to look up the correct and final entry pointing to a 4 kB page of physical memory, with the remaining 12 bits being used as an offset. Traversing the page tables is referred to as a page-table walk.

In addition to the flags that indicate valid entries and whether the entry refers directly to physical memory or the next level, each entry contains several other flags. These flags can indicate several properties, such as whether the corresponding memory is read-only or writable, present, has recently been accessed, and whether it is accessible to userspace.

As applications frequently need to access physical memory, it is necessary that the translation only introduces a small delay to a memory access. This can be hard to achieve as the paging structures themselves are stored in memory. Hence, caches reduce the number of main memory accesses (cf. Section 2.3.1). The Translation Lookaside Buffer (TLB), a

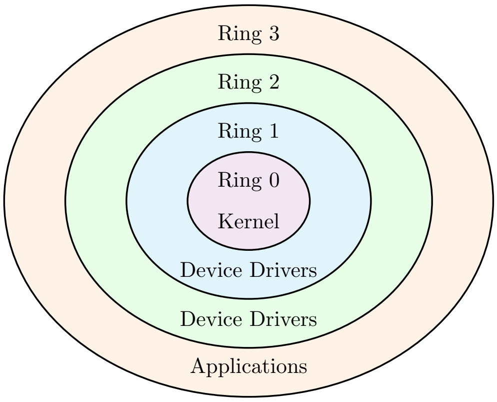


Figure 2.2: The different protection rings on x86 processors [254]. Ring 0 has the highest privilege and ring 3 the lowest. Rings 1 and 2 are rarely used as the device drivers are often directly part of the OS.

dedicated cache that stores the result of recently performed translations, can be used to skip the time-consuming page-table walk.

2.1.2 System-Call Interface

Userspace applications need to rely on the OS for specific tasks, such as memory management, process creation and termination, and hardware interaction. Applications run in what is called ring 3 while these tasks require ring 0 privileges (cf. Figure 2.2). To facilitate this interaction between a userspace application and the OS, the OS defines so-called system calls (syscalls) that expose the necessary functionality to the application. If an application performs a syscall, the processor transitions into a higher privilege level, the kernel is invoked and performs the task on behalf of the application. Once the task has been fulfilled, the kernel drops the higher privilege level and hands control back to the userspace application, which can continue its execution. Naturally, an interface that exposes higher-privilege code to unprivileged code is of great interest to malicious actors.

2.2 Sandboxing and Isolation

The applications that run on our computers can come from various untrusted sources. Once executed, they typically have unrestricted access to the network, storage, or the syscall interface. Unfortunately, they are also often written in memory-unsafe languages like C, enabling attackers to hijack them through memory safety violations [289]. Hence, by hijacking an application, an attacker also gains unrestricted access to the previously discussed resources. With sandboxes, it is possible to create a restricted environment that constrains the resources available to an application to the bare minimum, *i.e.*, the application only gets the resources it needs. Sandboxes can be created and enforced in many different ways, *i.e.*, by the hardware, the OS, or the programming language. Several large projects already use sandboxes. For instance, browsers or mobile operating systems rely on various types of sandboxes to limit the post-exploitation impact of potential attacks [63, 138, 211]. In this thesis, we only focus on sandboxes that restrict an application's access to the syscall interface.

Isolation, on the other hand, attempts to isolate trusted code from an untrusted environment while potentially also restricting access to some resources, e.g., certain instructions or the syscall interface. Enforcement of the isolation is often delegated to the hardware as it is commonly assumed to be trusted. For instance, the hardware is tasked with enforcing the isolation of the kernel from userspace. It is also responsible to enforce isolation in a trusted execution environment like Intel Software Guard Extension (SGX), isolating the trusted enclave from the rest of the system (Section 2.2.1).

2.2.1 Intel SGX

A concrete example of an isolation mechanism is Intel SGX. Starting with the Skylake microarchitecture, Intel introduced the SGX instruction-set extension to isolate trusted code from an untrusted system [143]. SGX splits userspace applications into two parts: a trusted part, the so-called enclave, and an untrusted part. In SGX, the processor enforces the isolation, and no other system part is trusted. This means that neither the application nor the OS can access the enclave's memory. The SGX memory range is encrypted and integrity-protected to protect against bus-probing and cold-boot attacks. This results in a very strong threat model as SGX assumes that everything except the processor could be compromised, including the OS. A typical use case for SGX is digital rights management [320].

Similar to the system-call interface, SGX specifies a well-defined interface for the communication between the application and the enclave. Entering and exiting an enclave is done using a set of pre-defined instructions, *i.e.*, `eenter` and `eexit`. Any other attempt to access the enclave or its memory is prevented by the hardware.

SGX also imposes restrictions on the enclave to improve the security of the isolated code [143]. For instance, enclaves cannot perform any I/O operations, including syscalls. Hence, the enclave must rely on the untrusted application to facilitate communication with the OS. Additionally, in SGXv1, it is also not possible for enclaves to execute specific instructions, such as instructions that access the high-precision timer of the processor [266].

2.3 ISA and Microarchitecture

The ISA defines, among other things, data types, registers, I/O, supported instructions, or virtual memory (cf. Section 2.1.1). Several different ISAs exist, such as x86, POWER, or ARMv8. Any ISA guarantees that an application built for it runs on all CPUs implementing the ISA. However, the runtime performance may differ, independent of the processor's clock speed. The reason for this is that the ISA does not provide any information on how the individual parts should be implemented, *i.e.*, it only provides an abstract definition of a CPU. Therefore, the actual hardware implementation, *i.e.*, the microarchitecture, can differ even though the same ISA is implemented. Consider CPUs from Intel and AMD. Both vendors implement the x86 ISA, but the underlying microarchitectures differ due to different intellectual property being used by each company. Additionally, the microarchitecture does not only differ between companies but also between models of the same company. For instance, the microarchitecture of an Intel Core CPU differs from a Xeon CPU. These differences in the microarchitecture can lead to significant performance differences.

While the microarchitecture can change between different CPU generations, several so-called *microarchitectural elements* are commonly used as they significantly impact the performance. One example are caches, which we discuss in Section 2.3.1. Other examples are out-of-order pipelines that allow for parallel execution of instructions and predictors used in speculative execution to reduce the number of pipeline stalls. We discuss these in more detail in Sections 2.3.2 and 2.3.3, respectively.

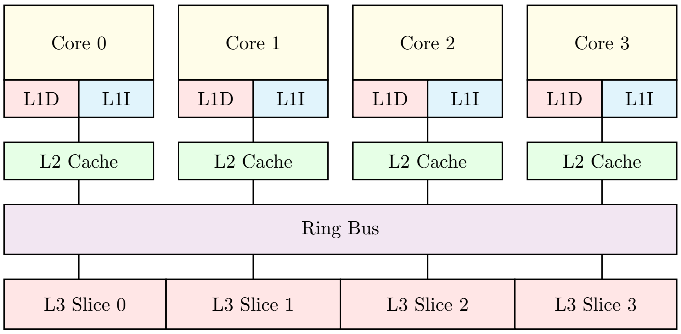


Figure 2.3: The cache hierarchy as found on typical Intel Core CPUs. Each core has a private L1 and L2 cache, with the L1 being split into a data (L1D) and instruction (L1I) cache. The L3 is split into slices and shared across all cores via the ring bus.

2.3.1 Caches

This section discusses the concept of caches. We discuss how they are organized (Section 2.3.1) and how they store data (Section 2.3.1).

Cache Organization

Processors have become faster and faster over the years, but the speed of main memory (DRAM) could not keep up with this development, making DRAM the main bottleneck in our modern systems. To alleviate the problem, vendors added caches to the processors. These caches act as small, high-speed buffers and are located between the CPU cores and DRAM. Hence, accesses to memory typically go through them, and copies of recent data loaded from DRAM are stored in the cache. The cache is checked upon a memory access to determine whether it can serve the memory load. If the data is in the cache and can be served from there, this is called a *cache hit*, which reduces the latency on the load. Otherwise, it is called a *cache miss*, and the data is served from DRAM.

Figure 2.3 shows the typical hierarchical layout of the cache for Intel CPUs. The hierarchy consists of multiple levels, with the one directly connected to the CPU core being the fastest but also the smallest. It is called the first-level (L1) cache and is private to the respective core. The L1 is commonly split into two parts, one for instructions (L1I) and one for data (L1D). The L1 is typically followed by a larger but slower L2 cache, private to the core. The last-level (LLC or L3) cache follows the L2 and is the largest but slowest cache. It is often split into so-called *slices*. While

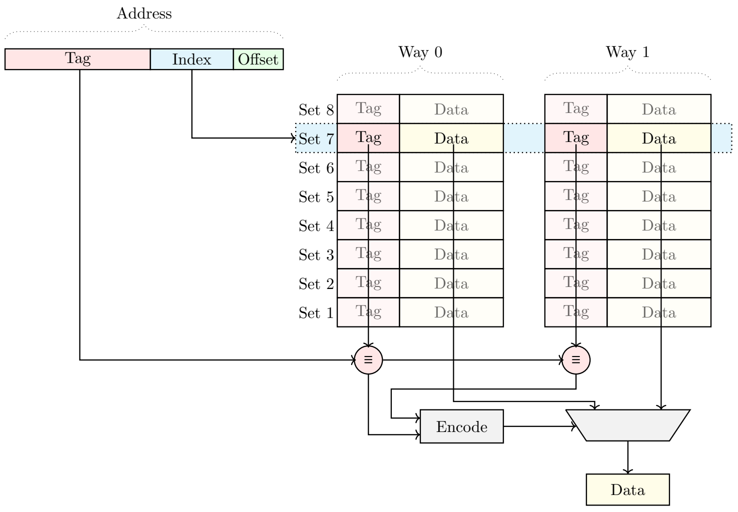


Figure 2.4: An illustration of a 2-way set-associative cache [180]. The lowest bits of the address are used as an offset to the cache line. The subset in the middle is used to determine the set, while the highest bits are used for the tag. When loading new data, the cache-replacement policy determines in which way the loaded data is placed.

every core can access the slice closest to it directly, it can access all others via a ring bus or a mesh. Hence, the L3 is typically shared among all cores of the CPU. With each level, the distance of the respective cache to the core increases. This results in higher access latencies, but also allows for larger caches.

Inclusion Policy. Caches in a hierarchy can follow different policies when it comes to which level holds copies of the data. A higher-level cache that holds all cache lines from a lower level is called *inclusive*. For instance, an L1-inclusive L3 cache also holds all cache lines from the L1. In an *exclusive* cache, data can only reside in a single cache level at the same time. If neither is the case, the cache is called *non-inclusive*, *i.e.*, data might be found on another level.

Set-associative Caches

There are different ways a cache can be designed, each having advantages and disadvantages. The design predominantly used in modern processors is called a set-associative cache design (cf. Figure 2.4). In such a design, the cache is divided into *cache sets*, which are then further sub-divided

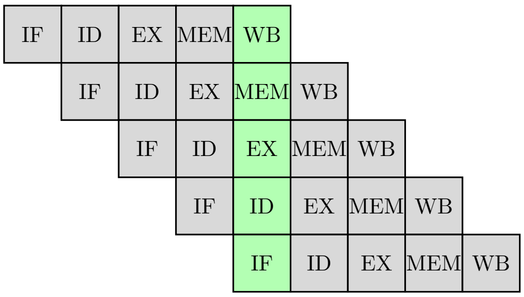


Figure 2.5: An example of a simple 5-stage pipeline. The stages are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

into *cache ways.* A way is also referred to as a *cache line* and is where the actual data is stored. Additionally to the data, each way also stores a *tag* that determines whether a cache line holds the requested data instead of unrelated data within the same set. It also stores additional metadata, such as whether the data has been modified or not.

The address of the data is used to determine the cache set and the tag. The cache way is then determined by the cache-replacement policy. As either the virtual or physical address can be used for the cache set and tag, there are 4 different types of caches [342]: virtually indexed and virtually tagged (VIVT), physically indexed and physically tagged (PIPT), physically indexed and virtually tagged (PIVT), and virtually indexed and physically tagged (VIPT). On Intel processors, the L1 is typically VIPT, while the higher levels use a PIPT design.

2.3.2 Instruction Pipelining

To increase the throughput of modern processors, vendors rely on instruction pipelining. The processor splits the execution of an instruction into several, processor-defined number of stages. Each stage is executed one after the other, buffering intermediate results between the individual stages. Once a stage has finished processing an instruction, the instruction can move on to the next stage of the pipeline at the next clock cycle. The freed stage can then process the next instruction. Figure 2.5 illustrates a simple 5-stage pipeline consisting of the following stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)

3. Execute (EX)

4. Memory Access (MEM)

5. Write-back (WB)

As the names indicate, each of the 5 stages outlined above is responsible for a specific task when executing an instruction. In the first stage (IF), the instruction that is to be executed next is fetched and passed on to the second stage (ID), where it gets decoded. Once decoded, the instruction is moved to the third stage (EX) where it is executed using the responsible execution unit. A potential memory access is then handled in the fourth stage (MEM) before the final results of the instruction are written back to the register file in the final stage (WB).

While the simple design of a pipeline above already employs parallelization, *i.e.*, instruction $i + 1$ is fetched while instruction i is decoded, modern processors go even further. These processors use an out-of-order execution design in which decoded instructions are placed in the order of the instruction stream in a buffer, the so-called Reorder Buffer (ROB), until the required execution unit is free and all operands are ready. Once this is the case, the processor moves the ready instruction to the execution stage even if it is not the next instruction in the instruction stream, *i.e.*, instruction $i + 1$ is executed before instruction i. A successfully executed instruction is marked as valid and complete in the ROB, making its results available to dependent instructions. If an instruction marked as valid and complete reaches the top of the ROB, it is retired, making its result architecturally visible. This ensures that instructions are retired in the order they appear in the instruction stream. This type of execution is based on Tomasulo's algorithm [296].

One important part that allows such pipelines to work efficiently is knowing which instruction needs to be fetched next. If the next instruction is known, the processor can already fetch it into the pipeline. One case where this is not possible is conditional branches, as their outcome is only known after the execute stage. Therefore, the pipeline must be stalled until the conditional branch has finished executing, causing a performance penalty. We discuss how modern processors can use speculative execution to prevent such stalls in Section 2.3.3. Other instances where the pipeline must be stalled are data hazards. An example of this are read-after-write hazards, which occur if an instruction requires the computed value of a previous instruction still in the pipeline.

2.3.3 Speculative Execution

The instruction stream of applications is almost never purely linear. Instead, it contains a multitude of branches that divert the control flow. However, the result of a branch instruction is only known after the execute stage, *i.e.*, after it has finished executing. In that case, the application does not know beforehand what the next instruction to fetch is and would need to stall, negatively impacting the performance. To minimize the number of stalls or even prevent them entirely, CPUs use a Branch Prediction Unit comprised of various individual *branch predictors*. By using these predictors, the CPU tries to predict the outcome of a control-flow change before the result is known and fetches the next instruction based on this prediction. If the prediction is correct, the CPU has already performed the necessary calculations and retires the respective instructions. Hence, the CPU saved time that would have been lost to the stall while waiting for the outcome of the control-flow decision. If the prediction was wrong, the CPU has to squash the results and start execution along the correct path. The feature of executing code that is potentially not needed is called *speculative execution*.

Branches can occur in various forms, *i.e.*, conditional branches, indirect jumps, calls, or returns. Modern processors contain a predictor for each one of these forms. To predict conditional branches, two different approaches can be taken: either through a *static* or a *dynamic* prediction. In the former, the branch is predicted based on the instruction itself, e.g., conditional branches are always considered to be taken [130]. With the latter, the processor gathers runtime information and bases the prediction on this information, which allows for a better predictor [60]. Common designs for such a dynamic predictor use a Pattern History Table (PHT) containing saturating counters [87]. The table can be indexed by the branch history or parts of the branch's address.

Another predictor, the branch-target predictor, tries to determine the destination of a branch instead of the outcome of a conditional branch. To make the prediction, it relies on the Branch Target Buffer (BTB), which stores the destination of the most recent target based on the source of the control-flow change [166].

The Return Stack Buffer (RSB) is used to predict the destination of a function return and is built on the observation that the sequence of calls and returns are tightly tied together [168, 192]. Hence, each call instruction pushes the return address on top of the RSB. Upon a return, the top of the RSB is popped and used as the predicted return destination.

In all cases, the CPU starts speculatively executing based on the prediction and later verifies whether the prediction was correct. Correct predictions result in the speculatively executed instructions being retired, improving performance. Wrong predictions cause the instructions and their results to be squashed [166].

While other prediction-based mechanisms like data prefetching [156], value speculation [233], or way prediction [240] have been described and deployed on modern systems, we do not consider such mechanisms in this thesis.

2.4 Side-Channel and Microarchitectural Attacks

Side-channel attacks differ from traditional attacks that exploit memory safety violations to leak information. While memory safety violations can lead to direct disclosure of the information the attacker wants to extract, side-channel attacks must infer the data from side effects of the execution of an application. An attacker can rely on many different side effects, e.g., electromagnetic emission [11, 247], power [193], or heat [136]. For a long time, such side-channel attacks required an attacker to have physical access to a device, e.g., to connect a device to measure the power consumption, as well as sophisticated and expensive hardware. However, physical access is no longer necessary as side-channel attacks have been demonstrated purely from software and even in a remote setup. In the former, the requirements are reduced to local code execution on the victim device while this requirement is completely removed for the latter.

2.4.1 Side Channels

We define a side channel as a channel that does not directly reveal data. Instead, it exposes metadata, allowing the actual data to be inferred with a certain probability [116].

We discuss this definition on the example of a frequently exploited cryptographic algorithm, *i.e.*, RSA. The modular exponentiation of RSA can be implemented using square-and-multiply [110]. In square-and-multiply, the bits of the secret key determine the type of operation that is performed. A '1'-bit indicates a square-and-multiply operation; a '0'-bit only performs a square operation. From a mathematical point of view, we assume the algorithm to be secure. We also assume that the algorithm is implemented without any software bugs, ensuring that an attacker cannot directly leak the secret key. However, the software implementation can still leak the key through observable metadata. The reason for this are side effects in the execution, *i.e.*, square-and-multiply takes longer or requires more energy than just a square operation. An attacker who can observe this metadata can reconstruct the key bits, exploiting the side-channel information for information disclosure.

2.4.2 Microarchitectural Side-Channel Attacks

The above example describes a side channel in the implementation of an algorithm. However, side channels can also originate in the microarchitecture of a processor. For instance, the internal state of some microarchitectural elements, such as caches, depends on the previously processed data. Observing this internal state allows an attacker to infer the sensitive information of the victim that led to the internal state. We refer to attacks that exploit such microarchitectural side channels as microarchitectural side-channel attacks.

The internal state of microarchitectural elements can be observed through various information sources on the microarchitectural or architectural side. The microarchitectural side exposes mechanisms that allow an attacker a view of individual elements of the microarchitecture. An example of such a mechanism are so-called performance counters, which provide fine-grained information on the performance and usage of various CPU elements. For instance, performance counters track whether a load resulted in a cache hit or miss, which reveals whether data was recently accessed. However, access to performance counters is typically restricted, forcing attackers to rely on unprivileged information exposed on the architectural side. One such information source is timing. An attacker can perform a timing-based attack by using the x86 `rdtsc` instruction to obtain a high-resolution time-stamp-counter value. By being able to accurately measure time, an attacker can, for instance, determine whether data was in the cache or not by observing how long an access takes.

3 State of the Art

In this chapter, we discuss state-of-the-art microarchitectural and control-flow-hijacking attacks, as both affect the security of the kernel. Section 3.1 discusses microarchitectural side-channel and transient-execution attacks. Section 3.2 focuses on traditional control-flow-hijacking attacks and defenses.

3.1 Software-based Microarchitectural Attacks

The internal state of microarchitectural elements can leak an application's sensitive information if an attacker can observe it. While modern processors contain numerous microarchitectural elements that can leak such information, the one that is most commonly used is the cache. Still, other microarchitectural elements, e.g., predictors, have also been exploited. More recently, the class of transient-execution attacks has been discovered. These are microarchitectural attacks that rely on microarchitectural side channels to directly extract sensitive information without having to infer it from side-channel information.

Many works have systematized the field of microarchitectural attacks over the years [9, 26, 96, 115, 189, 214, 284, 288, 334, 339]. In this thesis, we provide an additional overview of the state-of-the-art and how we extended it.

3.1.1 Cache Attacks

In a cache attack, an attacker exploits the measurable timing difference between data that is already present in the cache compared to data that must be fetched from main memory. This timing difference can leak sensitive information of a victim application as it allows an attacker to infer the memory locations the victim application recently accessed. The state-of-the-art presents several primitives that allow an attacker to observe the internal state.

Eviction sets. Eviction-based cache attacks like Evict+Time, Prime+Probe, and Evict+Reload rely on an attacker being able to evict a specific cache set that the victim uses. These attacks require an attacker to have a minimal eviction set and an eviction strategy [119]. The former is a set of addresses where each one falls into the same cache set as the victim's address. The latter defines the pattern in which the addresses of the eviction set are accessed to reliably evict the victim data. If the eviction set and strategy used by the attacker are correct, all data in the targeted cache set are replaced with attacker-controlled data.

The construction of an efficient eviction set depends on the CPU in use. While some CPUs directly use a subset of physical address bits to determine the cache set, other CPUs further partition the cache using cache slices. For the former, several works have demonstrated how to build an efficient eviction set [182, 225, 285]. While several works have done the same for the latter, it is significantly more difficult as an undocumented mapping function maps physical addresses to cache slices [120, 134, 140, 147, 187, 197, 329].

One difficulty in reverse engineering these mapping functions is the lack of physical address information. While older systems exposed the necessary information to unprivileged users, modern systems have restricted access to it. Therefore, attackers rely on dynamic and static approaches to build efficient eviction sets [282, 308]. If the system allows an attacker to use 2 MB pages, it becomes again trivial as the least-significant 21 bits of the virtual address are identical to the physical bits [119, 146, 187, 198, 270]. This information is sufficient to determine the cache set. Other approaches rely only on timing, allowing them to be used in attacks mounted from JavaScript [38, 99, 113, 119, 224, 279, 308]. Several works presented different eviction strategies and proposed methods to find and evaluate them [41, 119].

Evict+Time

Cryptographic algorithms were the first target of cache attacks [25, 234]. These attacks were subsequently generalized by Osvik et al. [225] as Evict+ Time. An attacker using Evict+Time first manipulates the internal cache state by evicting a specific set used by the victim, *i.e.*, the attacker primes the cache set with its own data. The next step is then to measure the runtime of the target algorithm or application. By observing an increased runtime, the attacker can infer that the victim has accessed data that falls into the primed cache set, *i.e.*, the victim data had to be fetched from main memory. However, if the runtime does not increase, the victim has not accessed an address that falls into the primed cached set.

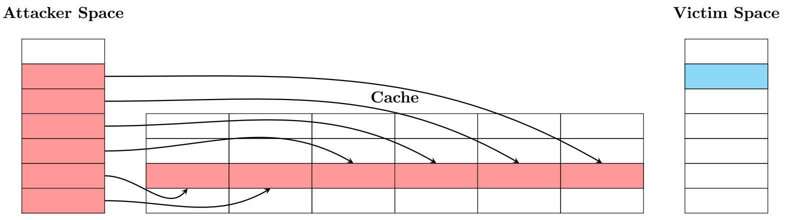
Due to its coarse-grained nature, Evict+Time is very susceptible to noise, forcing an attacker to perform multiple measurements. Evict+Time has been used to attack cryptographic implementations [25, 152, 182, 202, 225, 285, 298], and to break ASLR [113] and KASLR [134]

Prime+Probe

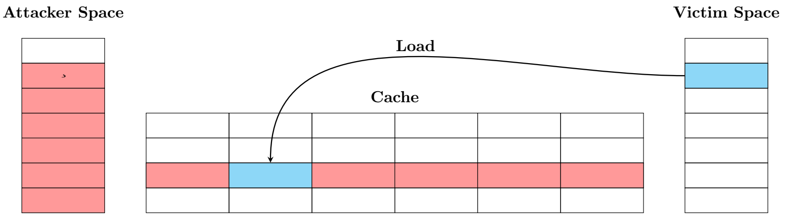
Prime+Probe [25, 225, 234] is a more powerful attack compared to Evict+ Time as the attacker does not need to measure the runtime of the victim. Instead, the attacker measures how long it takes to refill, *i.e.*, prime, the target cache set. Figure 3.1 illustrates a Prime+Probe attack.

In the first step, the attacker primes the target cache set using data from memory locations under their direct control (Figure 3.1a). In the second step, the victim performs some operations, potentially accessing data that falls into the primed cache set (Figure 3.1b). However, Prime+ Probe might be mounted in parallel to the victim instead of running interleaved. In the third and final step, the attacker probes the cache set and measures how long it takes (Figure 3.1c). This step implicitly primes the cache set for the next attack round. A high runtime in the probe step indicates victim activity in the target cache set, while a small runtime indicates no activity.

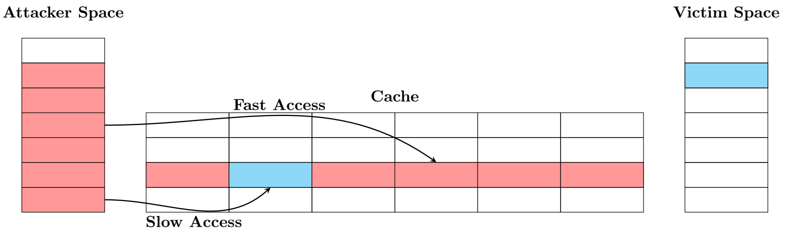
Using Prime+Probe, it is possible to target either the L1 caches, *i.e.*, instruction [5, 7, 8, 234, 341] or data cache [4, 6, 35, 43, 217, 225, 298], or the last-level cache [69, 126, 140, 146, 158, 187, 196, 197, 248, 250, 314]. A primary target on both the L1 cache and L3 are again cryptographic algorithms [97, 139, 146, 158, 182, 187]. However, Prime+Probe is not limited to cryptographic algorithms. Oren et al. [224] performed a Prime+ Probe attack in JavaScript, allowing them to spy on user behavior. Others used Prime+Probe to establish a covert channel in the cloud [198, 322,



(a) Step 1: Prime Cache Set



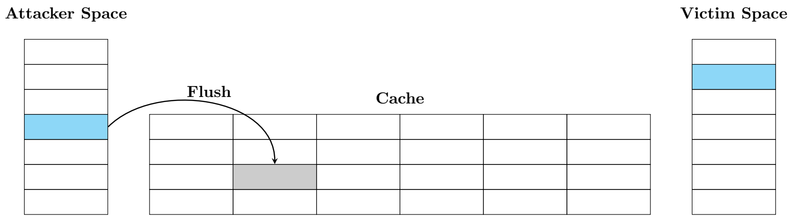
(b) Step 2: Schedule Victim



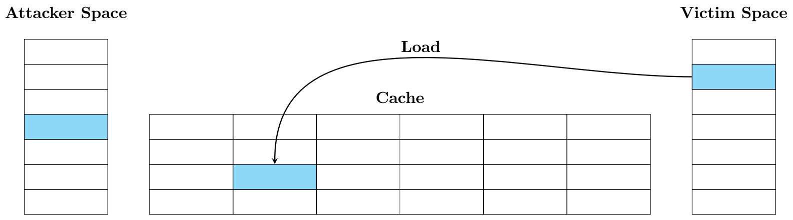
(c) Step 3: Probe and Measure

Figure 3.1: A Prime+Probe attack consists of 3 steps [180]. First, the attacker primes an entire cache set. Second, the victim performs a memory access, loading data into the cache and evicting the attacker's data. Finally, the attacker probes the cache set and measures how long it takes.

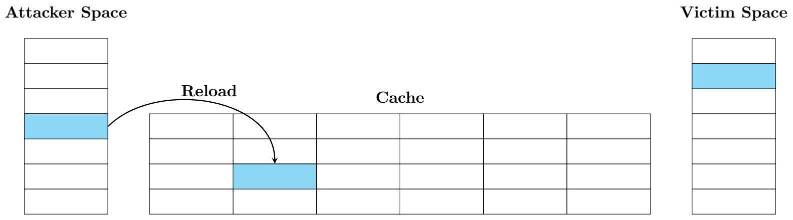
325]. Prime+Probe attacks have also been demonstrated from PNaCl and WebAssembly [99]. Schwarz et al. [268] proposed Multi-Prime+Probe, *i.e.*, spying on multiple cache sets in parallel, enabling attacks on user input. Kurth et al. [172] demonstrated a network-based Prime+Probe attack by abusing Intel's Direct Cache Access technology. Intel SGX enclaves have also been targeted [40, 69, 111, 205] or were used to hide such an attack [266]. While Prime+Probe attacks have pre-dominantly targeted inclusive caches, they have also been demonstrated on non-inclusive caches by targeting cache directories [326]. Furthermore, Prime+Probe has also been used in transient-execution attacks to leak the encoded information [297].



(a) Step 1: Flush Shared Cache Line



(b) Step 2: Schedule Victim



(c) Step 3: Reload and Measure

Figure 3.2: A Flush+Reload attack consists of 3 steps [180]. First, the attacker flushes the shared cache line. Second, the victim accesses the shared cache line, reloading it into the cache. Finally, the attacker reloads the data and measures the execution time of the reload.

Flush+Reload

Previously discussed attacks require an attacker to evict a cache set and try to determine changes within the entire set, *i.e.*, they work on a cache set granularity. However, flush-based attacks are also possible, as was first demonstrated by Gullasch et al. [125]. This attack led to the discovery of the more generic Flush+Reload attack [328], which measures the time required to reload a single cache line after being flushed.

We illustrate the three phases of a Flush+Reload attack in Figure 3.2. In the first phase, the attacker uses the flush instruction provided by the respective ISA, e.g., `clflush` on x86, to invalidate a cache line (Figure 3.2a). Similar to Prime+Probe, the second phase consists of the victim being scheduled and performing some task, e.g., an encryption (Figure 3.2b).

In the third and final phase, the attacker reloads the address used in the flush phase and measures how long it takes to reload it (Figure 3.2c).

If the victim accessed the address, the data is brought back into the cache. The subsequent access by the attacker then results in a cache hit, indicated by the load's low latency. However, if the victim does not access the invalidated address, the access by the attacker must be served from main memory, which requires significantly more time.

Flush+Reload has some advantages over previous attacks. Compared to Evict+Time and Prime+Probe, Flush+Reload has cache-line granularity and is virtually noise-free, enabling very stable and precise results. Second, no knowledge about physical addresses is required as the flush instructions use virtual addresses. Additionally, no eviction set is required as the data is directly flushed. However, it also has 2 limitations that hinders its usability in an attack. The first limitation is that it requires a dedicated instruction that flushes a cache line based on a virtual address. While both x86 and ARMv8 provide such an instruction, e.g., `clflush` and `dc civac` respectively, unprivileged access to the instruction can be disabled for the latter. These instructions are also not available in restricted environments such as SGX or JavaScript. Second, Flush+Reload requires that the targeted memory location is shared between the victim and the attacker. As the cache uses physical addresses, shared memory only resides in the cache once. Flushing it in one process flushes it for all other processes as well. If the memory is not shared, the attacker cannot flush the memory location for the victim. A typical target for Flush+Reload is the victim executable or shared libraries [122].

Similar to Prime+Probe and Evict+Time, Flush+Reload has been used to attack cryptographic algorithms [15, 24, 46, 92, 114, 122, 125, 127, 139, 148–150, 235, 239, 327, 328, 340], spy on user behavior [122, 182, 209, 313, 341], build a covert communication channel [51, 166, 182, 185], to detect double-fetch bugs [265], or as a building block for transient-execution attacks [51].

Flush+Flush. Flush+Flush [120] is a variant of Flush+Reload that does not require an explicit reload of the memory location by the attacker. Instead, it relies only on flush instructions, measuring how long it takes the attacker to flush the memory location again instead of reloading it. If the victim accessed the memory location, the flush instruction requires more time than when the location was not accessed.

Flush+Flush has been used to attack cryptographic algorithms [42, 120], page tables [301], and to spy on user behavior [120].

Evict+Reload. Restricted environments, such as JavaScript, do not provide a way to execute the flush instruction. However, attackers still want to profit from the fine-grained resolution of Flush+Reload. Evict+Reload [122, 182] provides the same granularity as Flush+Reload. Instead of relying on a flush instruction in Figure 3.2a, the attacker uses eviction. All other steps remain the same as in Flush+Reload. This enables Flush+Reload-type attacks in restricted environments [79, 113, 166, 257, 264] and on ARM-based systems where the flush instructions can be privileged [182].

Other Cache Attacks

Several works have abused the LRU replacement policy to mount attacks [41, 243, 323]. Others have introduced congestion to the mesh interconnect to leak cryptographic keys [312] or used timing differences introduced by dirty cache lines [68]. With Prime+Abort [78, 121], the timing of the probe step is replaced with the abort semantics of Intel TSX, eliminating the requirement of a high-resolution timer. Lipp et al. [183] exploited timing differences introduced by cache-way predictors. With CacheBleed, Yarom et al. [330] abused cache-bank conflicts to recover a victim's secret information, such as RSA keys.

3.1.2 TLB Attacks

Modern CPUs feature other types of caches besides the instruction and data caches. One such cache is the translation lookaside-buffer (TLB), which stores the result of recent virtual-to-physical address translations. Similar to previous caches, the idea behind the TLB is to gain performance by removing the requirement of virtual-to-physical address translations for recently accessed pages. However, this difference in performance also leads to a timing side channel, which has been exploited in several previous works. Some have used the information on whether a translation is present in the TLB to de-randomize the kernel address space [53, 118, 134, 155, 170, 264]. Others have attacked cryptographic algorithms [112] or spied on user behavior [264]. Van Schaik et al. [304] performed an indirect cache attack by abusing translation operations of the MMU. Gras et al. [112] and Tatar et al. [291] reverse-engineered the TLB to leak cryptographic keys or to perform Rowhammer attacks.

3.1.3 Attacks on Predictors

While caches try to hide the latency of memory accesses or virtual-to-physical address translations, predictors are used to avoid pipeline stalls. Predictors typically try to predict either control or data flow. However, they can also be abused by malicious actors as predictions are based on previously observed data.

Branch Predictors. The first work that exploited branch predictors measured timing differences caused by branch mispredictions to recover cryptographic keys [10]. Evtyushkin et al. [84] built a Prime+Probe covert channel on branch predictors. Subsequent work has then abused branch predictors to break KASLR [85]. Lee et al. [177] demonstrated that a BTB side-channel attack allows an attacker to infer the control flow of an SGX enclave. Huo et al. [135] targeted Intel SGX with a PHT-based side-channel attack. Evtyushkin et al. [86] exploited the directional branch predictor to attack Intel SGX.

Prefetchers. CPUs try to predict future memory access, issuing them before they may be necessary. If successful, the necessary data might have been loaded into one of the caches before it is needed. Reverse-engineering these hardware prefetchers and their replacement policies has led to a prefetcher-aware Prime+Probe attack [314]. Others have also reverse-engineered the prefetcher on out-of-order Intel CPUs and used the resulting information to attack cryptographic algorithms [27, 276]. Rohan et al. [251] established a covert channel between two processes on the same physical core by reverse-engineering the L2 hardware prefetcher. A similar covert channel was built by Cronin and Yang [67].

Memory Disambiguation. Before each load, the CPU checks for previously performed stores to the same address where the data has not yet been propagated to the cache hierarchy, *i.e.*, it still resides in the store buffer. If such a store exists, the data is forwarded to the load, ensuring no stale data is used. This optimization is called store-to-load forwarding. However, the check does not use the full address but only the lower 12 bits to determine whether the load must be reissued or not [142]. Additional, partial checks of the phyiscal address are also performed [151]. Like other optimizations, store-to-load forwarding has also been exploited in various ways. For instance, the partial address check allows an attacker to establish a covert channel [287]. Moghimi et al. [204] recovered cryptographic keys after injecting false dependencies of memory read-after-write hazards. Schwarz et al. [264] combined store-to-load forwarding with the TLB to break KASLR. Islam et al. [151] accelerated Prime+Probe and Rowhammer attacks by abusing the dependency resolution logic to recover physical addresses.

3.1.4 Transient-Execution Attacks

Transient-execution attacks are a new class of attacks that allow attackers to leak a victim's data directly instead of only metadata. This makes them much more powerful than traditional microarchitectural side-channel attacks. Transient-execution attacks rely on instructions that are executed transiently, *i.e.*, instructions that have been executed by the CPU but whose results are never committed to the architectural state [51, 166, 185, 302]. This can, for instance, occur if the instruction is part of the predicted path following a control or data flow prediction, but the prediction was wrong. Another example are instructions that are executed during out-of-order execution after an exception has occurred. While the result of these transient instructions is never made architecturally visible, they leave microarchitectural side effects. Using traditional microarchitectural side-channel attacks, an attacker can make them visible on the architectural level.

Transient-execution attacks can be distinguished by their trigger and grouped into three subclasses [180]. Spectre-type attacks are triggered by a misprediction, while Meltdown-type attacks rely on illegal data flow following various faults or assists. On the other hand, LVI-type attacks reverse Meltdown by not extracting data but injecting it into the victim domain. The remainder of this section provides an overview of the three subclasses.

Spectre

The first class of transient-execution attacks is Spectre-type attacks, exploiting control- and data-flow mispredictions. To perform such an attack, an attacker must mistrain the underlying prediction unit so that the CPU is tricked into executing code that it should not execute based on the architectural instruction stream. These code snippets, called gadgets, encode the victim's sensitive data into a microarchitectural element from which the attacker can extract it using traditional side-channel attacks, bringing it to the architectural level.

When first disclosed, Spectre variants exploiting the Pattern History Table (PHT) and the Branch Target Buffer (BTB) were demonstrated in userspace and the kernel [166]. We discuss the idea of a Spectre-type attack using Spectre-PHT as an example. The PHT is responsible for making predictions on whether a branch should be taken or not. An attacker mistrains a branch by repeatedly providing a value that results in the branch being taken, e.g., values that are in-bounds in the case of a bounds check. When the victim reaches the branch with an out-of-bounds value, the PHT predicts that the branch is taken. This results in an out-of-bounds memory access in the transient domain. Typically, the out-of-bounds loaded data is encoded in the microarchitectural state, e.g., caching of a shared memory location based on the value of the accessed data. The attacker can then recover the actual value using traditional side-channel attacks instead of having to infer it from extracted metadata.

The first disclosure of these types of attacks spurred a significant amount of further research. For instance, the PHT has also been exploited for transient buffer overflows [164]. Subsequent work exploited the Return Stack Buffer (RSB) [168, 192] and store-to-load forwarding (STL) [132]. Recently, Barberis et al. [20] showed that cross-privilege attacks via the branch history state are possible despite countermeasures. Schwarz et al. [271] demonstrated that Spectre attacks can even be abused remotely. Intel SGX [58, 222] and System Management Mode [80] have also been targeted. Göktaş et al. [105] combined Spectre with a memory corruption vulnerability while Bhattacharyya et al. [28] demonstrated a Spectre-style code-reuse attack. Wampler et al. [311] hid malware from static and dynamic analysis by leveraging speculative execution.

While the cache has been predominantly exploited in Spectre-type attacks [22, 51, 105, 164, 166, 168, 192], other side channels have been used as well. Bhattacharyya et al. [29] demonstrated that port contention can be used to transmit the secret data. Fustos and Yun [91] also relied on contention-based side channels. Schwarz et al. [271] demonstrated that timing differences introduced by the AVX unit can be exploited. Weber et al. [318] exploited the property that an `x87-FPU` instruction triggers a faster reset of the AVX unit. Further work by Schwarz et al. [264] exploited effects of the TLB in combination with the store buffer. Lipp et al. [183] used AMD's cache way predictor. Finally, Ren et al. [249] relied on the μOP cache.

Besides being able to mistrain the chosen predictor, an attacker must also find a suitable gadget to encode the victim's sensitive data. Several works have explored ways to find these gadgets. For instance, several works performed taint analyis [34, 157, 244] while others relied on taint tracking [316]. Other works instead relied on symbolic execution [123, 315]. Fuzzing of instrumented binaries has also been used to detect Spectre gadgets [223]. Finally, neural embeddings were used to both construct and detect gadgets [295].

Our systematic evaluation of Spectre-type attacks [51] provided a classification based on the prediction unit that the attacker mistrains, *i.e.*, the Pattern History Table (PHT), Branch Target Buffer (BTB), Return Stack Buffer (RSB), and Store-to-load forwarding (STL). In addition, we demonstrated that several mistraining strategies exist. These mistraining strategies allow attackers to mistrain the predictor from the same or another address space. We showed that shadow branches, *i.e.*, branches that are congruent to the victim branch, can also be used for mistraining.

Meltdown

The second class of transient-execution attacks is Meltdown-type attacks, which exploit that data is erroneously forwarded during out-of-order execution following a fault or assist. These events require a pipeline flush or a selective replay of instructions [180]. However, while still in the transient domain, the attacker can encode the data in the microarchitectural state and recover it afterward using traditional side-channel attacks. Spectre-type attacks require an attacker to trick the victim application into performing the secret access and subsequent encoding. Meltdown-type attacks are bypassing protections to directly access the victim's sensitive data. Hence, no gadgets are necessary as all the code is under the direct control of the attacker.

The original Meltdown attack [182], which we classified in our work as Meltdown-US [51], performs an access to a kernel address from an unprivileged userspace application. Due to the lack of privileges, *i.e.*, access to another security domain, the load faults and triggers an exception. However, exceptions are only raised at the retirement of the faulting instruction. This leaves enough time for subsequent transient instructions in the out-of-order execution to encode the loaded, usually inaccessible data into the cache. The attacker can recover the encoded data by using traditional cache side channels, such as Flush+Reload.

While Meltdown-US breaks down the isolation between kernel and userspace, Foreshadow [302] exploited effects around the present bit in the page tables to attack SGX enclaves. Kiriansky and Waldspurger [164] explored Meltdown effects around the read-only bit.

Meltdown-type attacks have been used to leak data from various domains: the operating system or hypervisor [53, 185, 257, 269, 321], virtual machines [269, 286], other userspace applications [185, 257, 269, 286], and trusted-execution environments [245, 258, 269, 302]. In these attacks, the L1 cache is predominantly used to leak data. However, several works have studied the line-fill buffers and load ports more thoroughly [206, 257, 269] after the original Meltdown paper [185] already connected the line-fill buffer to some leakage. With TSX Asynchronous Abort [255, 256, 258, 269], researchers have demonstrated that Intel TSX can be abused to leak data on CPUs that contain fixes against other attack variants discussed in the ZombieLoad paper [269]. Moghimi et al. [206] proposed a fuzzing approach to generate new attack variants that attempt to leak from different buffers. Various registers have also been targeted in attacks, such as floating-point registers [145, 286] and staging buffers shared between cores [245].

In our work [51], we systematically analyzed the remaining bits in the page tables and other ways of triggering exceptions, which led to the discovery of two new variants. The first exploited Meltdown effects around Intel's memory protection keys. The second triggered Meltdown effects through bound-checking exceptions, which was also the first time that AMD processors were shown to be affected by Meltdown effects. In addition, we provided a classification based on the fault condition. The results of subsequent work [206], which used performance counters to understand better which hardware paths are exploited, demonstrated that all variants fit into the provided naming scheme.

Load-Value Injection

Meltdown-type attacks directly leak sensitive data from the victim domain. LVI-type attacks [303], on the other hand, trick the victim into revealing its sensitive data by injecting malicious values into the victim domain. However, LVI differs from Spectre-type attacks as no prediction history is poisoned. Similar to Spectre-type attacks, an attacker must identify a gadget in the victim domain that accesses and sends the sensitive information. The gadgets required for LVI-type attacks are much simpler than for Spectre, potentially as simple as a single load operation.

In an LVI-type attack, the victim must perform an operation that results in a fault or assist. While these faults or assists can occur naturally in the victim's execution, LVI is easier to exploit in a scenario where the attacker can easily induce them, such as in Intel SGX. In addition to being able to induce the fault precisely, this scenario allows the attacker to repeatedly attempt injecting the malicious data by zero-stepping the targeted instruction [300]. However, Van Bulck et al. [303] demonstrated that a user-to-kernel attack is also possible.

A special case of LVI is LVI-NULL [103, 303], where it is sufficient for the attacker to inject a zero value. CPUs containing Meltdown fixes make this attack variant easier as the illegally loaded value is zeroed out before being passed on to dependent instructions [50, 144].

3.2 Memory Safety

One of the oldest problems in computer security are memory safety violations [289], which have been extensively studied. Previous work has investigated various memory safety violations [82, 289], provides a historical overview [306], or discusses protection mechanisms [332]. One of the main goals of an attacker exploiting a memory safety violation is to gain code execution on the victim machine by hijacking an application [289]. This type of attack is commonly referred to as a control-flow-hijacking attack. They are a focus of this thesis as they can significantly impact the kernel's security. Specifically, this thesis focuses on hardening the kernel by restricting a hijacked applications access to the syscall interface. In this section, we provide a coarse overview of the state-of-the-art on existing control-flow-hijacking attacks and defenses.

3.2.1 Control-Flow-Hijacking Attacks

A control-flow-hijacking attack consists of several steps with the goal of redirecting control flow to an attacker-chosen location. In a first step, an attacker exploits a memory safety violation, such as a buffer overflow, to overwrite an existing code pointer. Typical targets are the return address on the stack or a function pointer used by an indirect jump.

Once the code pointer has been corrupted, its value must be loaded into the instruction pointer [289]. However, the instruction pointer can only be modified by control-flow transfer instructions, *i.e.*, jumps, calls, and returns. The attacker must perform an indirect call or jump using the corrupted pointer or execute a return instruction, which uses the modified return address on the stack. The final step is then the execution of the attacker-chosen malicious code.

Shellcode. In the most straightforward case, the attacker injects custom, executable code into the victim's address space, the so-called shellcode, to which control flow is redirected [13]. This shellcode can then exploit further vulnerabilities to elevate privileges or simply open a reverse shell through which the attacker can perform a variety of other attacks. AlephOne [13] demonstrated this by overflowing a buffer on the stack with shellcode and redirecting control flow to the buffer. However, modern CPUs provide a non-executable bit that can be set on a page-level granularity [203, 289]. If set, the page's content cannot be executed by the CPU. When loading an application, the operating system marks all pages that contain data, like the stack or heap, as non-executable. On the other hand, code pages are marked as executable but not writable, resulting in the W⊕X (Write XOR Execute) policy [231]. As the attacker code is injected into the victim's address space as data, the code must be written to writable data pages. The injected code can then not be executed. Due to the W⊕X policy, already existing and executable code can also not be modified by the attacker, providing code integrity [289]. Hence, the combination of W⊕X and data execution prevention prevents a straight-up shellcode attack. However, W⊕X cannot be fully enforced in the presence of just-in-time compilation or self-modifying code [289].

Return-to-libc. Due to the above restrictions, attackers must rely on other methods to achieve their goals. As only the victim's code is executable, attackers rely on the victim's code to perform malicious tasks, *i.e.*, they perform a code-reuse attack. Such an attack was first demonstrated by Solar Designer in the form of a return-to-libc (ret2libc) attack [281]. In a ret2libc attack, an attacker redirects control flow to a function within libc (or another mapped library) after preparing the necessary parameters on the stack. Nergal [216] extended ret2libc attacks to allow for unlimited chaining of function calls. However, changes to the calling convention on x86_64 made ret2libc attacks harder as the function arguments were no longer expected to be on the stack but in registers. On x86_64, simply pushing all arguments onto the stack and calling a libc function is no longer possible [171].

Borrowed Code Chunks Technique. To solve this problem, Krahmer [171] proposed the *borrowed code chunks technique.* Instead of re-using entire functions, the approach re-uses parts of functions to fill the required registers with the function arguments. With the registers prepared, the attacker invokes the target function, *i.e.*, performs a traditional ret2libc

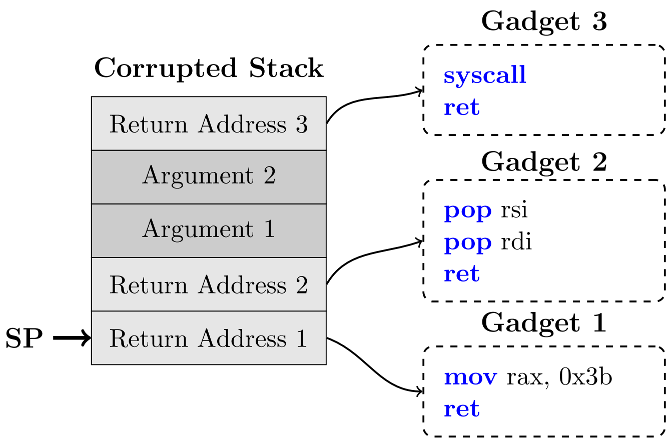


Figure 3.3: An illustration of a ROP attack. The attacker prepares the stack with the return addresses for the individual gadgets and the required arguments for each one.

attack. However, even ret2libc attacks performed in this way are still limited to executing code in a straight line, *i.e.*, no loops or branches are possible [274].

Return-Oriented Programming. With Return-Oriented Programming (ROP), Shacham [274] generalized ret2libc attacks and the borrowed code chunks technique. In a ROP attack, an attacker does not invoke any functions but instead re-uses existing code chunks, so-called gadgets, to perform the attack (Figure 3.3). Each such gadget is a sequence of instructions followed by a return instruction, e.g., `pop rdi; ret`. On x86, with its variable instruction length, these sequences might not have been placed there implicitly by the compiler. We show an example of this in Listing 3.2.1. First, the byte sequence disassembles to a simple subtraction with a register and a constant value. However, by skipping the first byte, we obtain a sequence that moves a value into the `al` register, pops a value into the `rdi` register, and finally executes a return instruction. This property of x86 allows an attacker to use gadgets that are not intended by the compiler. Barrantes et al. [21] have shown that a random byte stream can be interpreted as a valid instruction sequence with a high probability due to the denseness of the x86 instruction encodings. Furthermore, Shacham [274] and Homescu et al. [131] demonstrated that, in a large enough binary, ROP is Turing complete. Hence, ROP provides a fully

```
// 2d b0 03 5f c3
sub eax, 0xc35f03b0

// b0 03 5f c3
mov al, 0x3
pop rdi
ret
```

Listing 3.2.1: Disassembling the first byte sequence results in a simple sub instruction with a register and a constant value. However, skipping the first byte of the sequence results in a ROP gadget that moves a value into `al`, pops a value into `rdi`, and then returns.

functional language, allowing an attacker to perform any task necessary in an attack.

ROP Variants. While ROP has been demonstrated to be a powerful attack and one that is hard to eliminate, other variants also exist. For instance, Checkoway et al. [57] and Bletsch et al. [33] demonstrated Jump-Oriented Programming (JOP), a code-reuse attack that uses jump instructions instead of returns. Abusing indirect jump instructions can eliminate the requirement of modifying the stack, although it is still possible. However, maintaining control of the program's execution is significantly more difficult than in ROP attacks. Bletsch et al. [33] solve this problem with a new type of gadget, called the *dispatcher gadget*, which is responsible for determining which gadget is to be invoked next. It does this by relying on a dispatcher table and a virtual program counter. To stay in control, the dispatcher gadget ensures that each invoked gadget transfers control back to it at the end of its execution by executing a jump instruction. Checkoway et al. [57] instead rely on a trampoline gadget, which relies on stack modifications instead of a dispatcher table and virtual program counter. Pure-Call Oriented Programming (PCOP) is a variant of a code-reuse attack in which gadgets do not end with a return or jump instruction but a call instruction [252]. Their work improved upon previous Call-Oriented Programming work, which still required jump and return gadgets [33, 56, 106]. Bittau et al. [30] proposed Blind ROP (BROP), an attack technique that allows an attacker to find ROP gadgets remotely in an unknown binary. One requirement of BROP is that the service must restart in the case of an error. A successful BROP attack results in a ROP chain that executes the *write* syscall and sends the unknown binary to the attacker. Subsequent work improved BROP in the form of Signal Enhanced Blind Return-Oriented Programming (SeBROP) [338]. Bosman and Bos [37] proposed another alternative of ROP called Sigreturn-Oriented Programming (SROP). The idea is to push a forged sigcontext structure onto the stack, overwriting the correct return address with the address of a gadget. Once constructed, the attacker executes the *sigreturn* syscall to complete the attack. The advantage of SROP is that a single execution of the *sigreturn* syscall can set all necessary registers for the attack. Schuster et al. [261] proposed Counterfeit Object-Oriented Programming (COOP). COOP attacks rely on an attacker chaining existing C++ virtual functions together to perform malicious tasks. Carlini et al. [55] proposed Printf-Oriented Programming, demonstrating that a single call to *printf()* enables Turing-complete computations. Loop-Oriented Programming [174] relies on entire functions as building blocks, *i.e.*, gadgets. These gadgets are chained together so that control flow follows the call-ret-pairing process. One gadget, a function containing a loop, is used to chain all other gadgets together. This function is called a loop gadget.

ROP as a Building Block. While ROP and its variants -complete [37, 131, 261, 274], performing the entire attack in this form is tedious. Hence, attackers often use ROP as a building block to enable shellcode. In such an attack, the attacker first injects shellcode into the victim's address space. Next, the attacker builds a ROP chain that moves the shellcode address, its size, and the correct permissions into the registers defined by the calling convention. Once the registers are prepared, the attacker performs an *mprotect* syscall to make the shellcode executable. The final step of the ROP chain is to return to the shellcode location to start its execution.

Building a ROP chain is a critical part of these attacks. While manually building the ROP chain is possible, it requires significant manual effort and is error-prone. Hence, tools exist that help in the building of ROP chains [253, 273].

Benign Use Cases of ROP. While ROP has been proposed as a powerful attack technique, a significant amount of research relies on ROP to perform benign tasks. For instance, several works have evaluated ROP in the context of software watermarking [16, 191]. Borrello et al. [36], Mu et al. [212], and Shrivastava and Hota [278] used it to obfuscate control flow. Andriesse et al. [17] and Shrivastava and Hota [277] employed

ROP for code integrity verification. Finally, ROP has also been used for steganography [190].

Control-Flow-Hijacking Attacks on the Kernel

Control-flow-hijacking attacks have been demonstrated on userspace applications and the kernel. Due to its elevated privileges, gaining control of kernel execution is extremely valuable to attackers. While the previously discussed control-flow-hijacking attacks apply to the kernel as well, additional attacks are possible. We again consider non-control-flow-hijacking attacks on the kernel out of scope, e.g., data-only kernel exploits targeting process credentials or page tables.

One of the most prominent examples of such an additional attack on the kernel is return-to-user (ret2user). In a ret2user attack, an attacker exploits a memory safety violation in kernel code to overwrite kernel data, typically function pointers, with userspace addresses [159, 160, 236]. This allows an attacker to redirect the kernel's control flow to malicious code locations in userspace. As a result, the kernel executes attacker-chosen or controlled code with elevated privileges. Hence, ret2user attacks are an instance of the confused deputy problem [128].

A variant of ret2user attacks are return-to-direct-mapped memory (ret2dir) attacks [159]. These attacks allow an attacker to bypass all protections implemented against ret2user attacks and entirely circumvent kernel isolation. At the root of ret2dir attacks lies implicit page frame sharing, which allows an attacker to "mirror" userspace data, e.g., the malicious payload, in the kernel address space. Hence, an attacker can perform the equivalent of a ret2user attack without ever reaching userspace by simply redirecting control flow to the direct-physical map.

3.2.2 Mitigating Control-Flow-Hijacking Attacks

This section discusses proposed defenses against control-flow-hijacking attacks. We only focus on defenses that try to restrict control-flow-hijacking attacks. Defenses that try to prevent the memory safety violation that leads to the hijack, e.g., buffer overflow, are considered out of scope. While defenses against control-flow-hijacking attacks can be primarily placed into two groups [289], *i.e.*, code pointer integrity (CPI) and control-flow integrity (CFI), we only focus on the latter due to its relevance to this thesis. For a more in-depth discussion, we refer the reader to prior work [44, 289].

Address Space Layout Randomization. Control-flow-hijacking attacks require an attacker to have precise knowledge of virtual addresses so that control flow can be correctly redirected. Address Space Layout Randomization (ASLR) is a probabilistic countermeasure in which every newly launched process has a new randomized virtual address space, *i.e.*, the location of the code segment, libraries, stack, and heap are randomized upon program launch [230]. This imposes restrictions on an attack as ASLR needs to be broken before an attacker can, for instance, prepare the stack with the correct return addresses for a ROP attack. However, several works have demonstrated that ASLR can be bypassed via pointer leaks [74, 280] or even brute-forced [30, 38, 95, 108]. Still, most major operating systems have enabled ASLR by default [108]

Stack Canary. One of the best-known control-flow-hijacking attacks is the previously discussed stack smashing attack by AlephOne [13, 289]. Stack canaries are intended to mitigate the attack [65]. The stack canary, a secret value, is placed on the stack between local variables and the return address [289]. If an attacker naively modifies the return address on the stack, the canary is likely modified as well. Before returning from a function, the stack canary is checked, and a deviation from the original value results in program termination. However, an attacker can leak the canary using an arbitrary read or format string vulnerability. Additionally, they can brute-force the canary as it stays the same after a fork [194]. In either case, the attacker can then corrupt the stack while keeping the canary intact.

Both gcc and clang can automatically add stack canaries if requested by the application developer [124]. The Linux kernel provides a build option that enables the feature for the kernel [161].

Shadow Stack. An alternative solution that is not vulnerable to information leaks and overwriting of the canary are shadow stacks [307]. With a shadow stack, each call to a function creates two copies of the return address: one on the usual stack and one on the shadow stack. Before returning from a function, the two values are compared, and a deviation again results in program termination. Shadow stacks increase the difficulty of an attack as the attacker must be able to corrupt both locations. Chiueh and Hsu [62] propose to either use guard pages to protect the shadow stack or to modify its write permissions. However, the latter introduces significant performance overheads. With CET, Intel provides hardware support for shadow stacks [141]. Entries on the CET shadow stack cannot be modified by regular store instructions, making CET more secure than software-only shadow stacks.

Burrow et al. [45] provide a comprehensive analysis of different shadow stack mechanisms.

Control-Flow Integrity. Stack canaries and shadow stacks only protect function returns, but not indirect control-flow transfers to new targets. To solve this problem, Abadi et al. [1] introduced the concept of Control-Flow Integrity (CFI). CFI tries to limit the targets of indirect control-flow transfer instructions, *i.e.*, the forward edge, and returns, *i.e.*, the backward edge, to a set of pre-determined valid targets. If successful, an attack is stopped before the attacker can, for instance, redirect control flow to the injected shellcode.

The set of valid targets is typically determined using static analysis [1, 44]. Each destination of an indirect control-flow transfer is instrumented with an ID, identifying an equivalence class. Indirect call sites within the application are instrumented with runtime checks that verify whether the destination has the correct ID. This ensures that the called target is part of the pre-determined set. In the proposal by Abadi et al. [1], function returns are protected using the previously discussed shadow stack concept.

CFI can be subdivided into two groups: coarse- and fine-grained CFI [55]. In a fine-grained CFI implementation, the set of pre-determined targets is as small as possible, *i.e.*, an application can deviate much less from the intended control flow. On the other hand, coarse-grained solutions allow for a more relaxed estimation of target destinations, *i.e.*, an application can deviate further before an error is detected. Coarse-grained solutions only validate that function returns target a location following a call instruction while the fine-grained solution by Abadi et al. [1] ensures that control is returned to the instruction following the actual caller [75]. However, the advantage of coarse-grained solutions is that they incur less of a performance penalty, but this comes at the cost of reduced security [75, 98, 337].

Since the original proposal by Abadi et al. [1], CFI has been a popular research field which resulted in several different solutions for kernel and userspace [32, 39, 61, 66, 72, 73, 77, 94, 98, 154, 162, 169, 173, 178, 195, 208, 210, 218–221, 228, 232, 238, 241, 259, 294, 305, 317, 331, 333, 335, 336]. Recently, Intel released CET [141], bringing hardware-supported CFI to commodity off-the-shelf systems. Support for CET within the Linux kernel is in development [175]. We refer to prior work for a discussion on the effectiveness and performance of various CFI solutions [44].

CFI has limitations that have partially hindered its adoption in the real world. First, several attacks have demonstrated that it can be circumvented [56, 75, 83, 106, 107, 262]. For instance, by relying on various types of gadgets, Göktas et al. [106] demonstrated that coarse-grained CFI can be circumvented. Similarly, Davi et al. [75] demonstrated that several coarse-grained CFI implementations can be circumvented even under weak adversarial assumptions. Evans et al. [83] demonstrated that fine-grained CFI can also be bypassed. Second, enforcing the correct control flow across dynamically-linked libraries is a complex problem as CFI information may not be available for these libraries [289]. Third, CFI requires a precise Control-Flow Graph (CFG) to determine the targets of a control-flow transfer accurately. However, obtaining a precise CFG, which requires sound and complete pointer analysis, is an undecidable problem [246]. This makes it necessary for CFI solutions to overapproximate the number of possible targets, which leads to a reduction in the provided security [44, 55]. Finally, determining the effectiveness of CFI solutions is a non-trivial task. Current solutions typically rely on the average indirect target reduction (AIR) metric proposed by Zhang and Sekar [337]. AIR is a metric that measures the average reduction of allowed targets across every indirect control-flow transfer instruction. Carlini et al. [55] showed that AIR is an imperfect metric that does not fully allow to reason about the effectiveness of a CFI solution. The reason is that AIR only measures the average reduction in targets but does not consider the usefulness of different basic blocks [89]. Other proposed metrics like QuantitativeSecurity [44], AIA [98], and CTR [213] all suffer from the same problem. However, recent work has proposed newer metrics that consider the usefulness of basic blocks [89].

Mitigating Control-Flow-Hijacking Attacks on the Kernel

As the kernel has complete control of the hardware and the applications that run on top of it, it must be protected as well as possible. Hence, the kernel applies several techniques to harden itself from being directly compromised [161]. In addition, it also tries to limit the post-exploitation impact in case an attacker was able to redirect control flow. Previously discussed countermeasures like stack canaries, shadow stacks, and CFI can also be applied to the kernel. As we discussed these concepts before, we do not discuss them further.

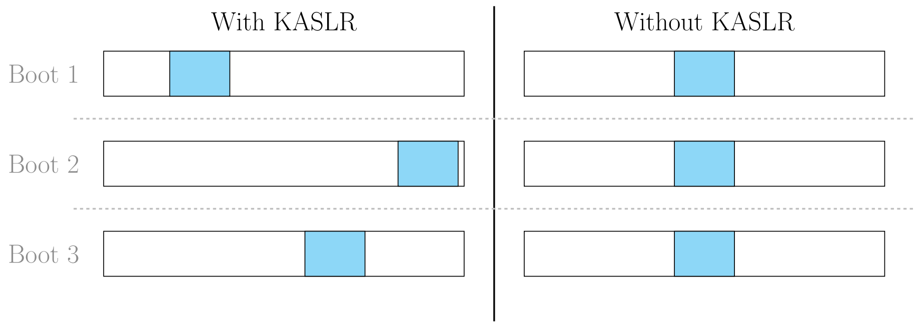


Figure 3.4: KASLR randomizes the location of the kernel text segment once at every boot. Without KASLR, the text segment is always loaded at the same location.

KASLR. Control-flow-hijacking attacks require an attacker to know the addresses of relevant functions and data. Like userspace ASLR, the kernel randomizes the location of various parts, e.g., the text segment and the direct physical map, once at boot time. This is called Kernel Address Space Layout Randomization (KASLR) and is illustrated in Figure 3.4 for the kernel's text segment. Accardi [2] proposed a function-granular KASLR solution that randomizes the kernel code on a per-function level, but this is not used by default.

With KASLR active, attacks that rely on precise information of kernel virtual addresses become unlikely in the absence of a KASLR break, as an access to an invalid address most likely results in a system crash. However, several KASLR breaks have been demonstrated that exploit side-channel information [50, 53, 85, 118, 134, 155, 163, 170, 181, 183, 184, 264, 318]. With KAISER (renamed to KPTI by the kernel developers), Gruss et al. [117] proposed an approach that tries to prevent side-channel attacks on KASLR. KAISER ultimately protects against the previously discussed Meltdown attack [104, 185].

In this thesis, we demonstrate a novel microarchitectural KASLR break, called EchoLoad, which abuses the incomplete hardware fixes implemented by Intel for Meltdown [50]. EchoLoad can even de-randomize the kernel location in the presence of KPTI as parts of the kernel must remain mapped. We propose a mitigation, called Fake Load Address REspone (FLARE), which prevents all microarchitectural KASLR breaks known at the time, including our new EchoLoad attack.

KERNEXEC and UDEREF. KERNEXEC and UDEREF [283, 292, 293] are two patches proposed by the PaX Team that prevent an attacker from redirecting control flow from kernel space to userspace. The implementation of the two patches differs based on the underlying architecture. On x86, the patches use segmentation, mapping the kernel into a segment that returns a fault when privileged code accesses userspace. On x86_64, userspace is mapped into a non-executable area upon entering the kernel and restored on exit. However, remapping userspace can incur a significant performance overhead [159].

SMAP and SMEP. Preventing access from privileged to unprivileged memory is an important security feature that must also be fast. Therefore, modern Intel CPUs provide hardware support to improve performance significantly. With Supervisor Mode Execute Protection [161] (SMEP), Intel introduced a feature that prevents the kernel from executing userspace code. Supervisor Mode Access Prevention [64] (SMAP), on the other hand, prevents the kernel from accessing userspace. If active, any attempt to access or execute userspace memory triggers a page fault. Other vendors, such as ARM, provide similar features, *i.e.*, Privileged Execute-Never [19] (PXN) and Privileged Access-Never (PAN) [18], respectively. On Linux, these features are active by default [161, 176].

kGuard. To prevent ret2user attacks, Kemerlis et al. [160] proposed kGuard. kGuard is a compiler plugin that identifies all indirect control-flow transfers within the kernel and instruments them with Control-Flow Assertions (CFA). Before any indirect control-flow transfer, the CFAs perform a runtime check to determine whether the target is located within the kernel, *i.e.*, they ensure that control flow is never redirected to userspace. Additionally, kGuard includes two code diversification techniques to thwart attacks that rely on bypass trampolines, *i.e.*, an indirect branch of which the attacker has control over the operands. The introduced diversification techniques randomize, at compile and runtime, the locations of the indirect branches and their CFAs.

XPFO. kGuard and other defenses were successfully bypassed by the ret2dir attack proposed by Kemerlis et al. [159]. Hence, they proposed a separate solution for ret2dir attacks called Exclusive Page-Frame Ownership (XPFO). The idea behind XPFO is that at any point in time, only one context, *i.e.*, kernel or userspace, can have a mapping to the physical page. If the kernel allocates memory for userspace, the kernel removes its own mapping to the same physical page in the direct physical map. An attacker performing a ret2dir attack can no longer redirect control flow to the “mirrored” page as this results in a fault. XPFO also ensures that all pages freed by userspace are zeroed out and returned to the direct physical map.

Syscall Filtering. Reducing the attack surface of the kernel is one of the goals of the Kernel Self-Protection Project [161] A part of this is to limit access to the interfaces exposed to (potentially hijacked) userspace applications, such as the syscall interface. Several works have been published on syscall checking [3, 14, 70, 88, 93, 102, 109, 153, 179, 237, 242, 309], but frequently rely on syscall interposition. In syscall interposition systems, the syscall is intercepted and redirected to a secure montior, e.g., a trusted userspace application, which checks and performs the syscall on the application’s behalf. A significant drawback of these systems is the increased runtime. Hence, the Linux Kernel uses none of them and instead relies on the Secure Computing (Seccomp) interface to restrict a userspace application’s access to syscalls [81]. In the beginning, seccomp only allowed a static set of 4 syscalls, *i.e.*, *read*, *write*, *exit*, and *sigreturn*. However, building a complex application that only uses these 4 syscalls is impractical.

With “seccomp mode 2”, *i.e.*, “seccomp filter mode”, the kernel developers extended seccomp so that a userspace application can inform the kernel about the syscalls it intends to perform over its lifetime. Seccomp requires the developer to specify these resulting filters in the Berkeley Packet Filter [200] (BPF) language. Since this change, several large-scale projects have integrated seccomp, such as browsers [63, 211] and Android [138]. However, adoption in other projects is slow as seccomp requires a developer to manually identify the set of syscalls required by an application, which is a labor-intensive and error-prone task if done manually.

Seccomp has several limitations. While seccomp can filter syscalls not just by their number but also their arguments, it is limited to arguments that are fully contained in a register, *i.e.*, integers. Arguments not fully contained in a register, like strings or structs, cannot be checked due to time-of-check-time-of-use (TOCTOU) vulnerabilities [201, 265]. Filtering string values, which is of interest for syscalls like the *exec** syscalls, is not possible. Due to the reliance on BPF, seccomp is stateless, *i.e.*, every syscall is checked independent of previously executed syscalls. Therefore, every syscall that is part of the filter can be executed at any point in time and originate from any location, *i.e.*, also possible to invoke a syscall from injected shellcode or unaligned instructions.

Multiple papers in this thesis improve upon seccomp. First, we propose a solution, called Chestnut, that automatically extracts the set of syscalls an application performs, either during compilation or by statically analyzing an existing binary [52]. Others have proposed similar approaches in concurrent work [76, 100, 101, 137, 226, 227]. Second, we propose Domain Page-Table Isolation (DPTI), a method for dynamic and time-restricted isolation. DPTI allows the kernel to restrict syscalls based on complex syscall arguments which are not fully contained in a register [54]. For instance, a kernel employing DPTI can restrict the applications that can be launched via the *exec** syscalls. DPTI does not require hardware changes or features that are not available in commodity systems. Instead, it only requires the memory isolation already enforced by the MMU. Finally, we propose the concept of Syscall-Flow-Integrity Protection (SFIP) [47]. Compared to seccomp, SFIP provides integrity to user-kernel transitions, enforces the syscall flow of an application through syscall sequences, and enforces the origin of individual syscalls. Enforcing an application's syscall flow also partially limits its control flow. SFIP is fully compatible with CFI and provides additional protection if userspace CFI has been circumvented. Enforcing syscall origins removes the possibility of an attacker using syscalls in injected shellcode or via unaligned instructions. In addition, every individual syscall is restricted to a predetermined set of locations, *i.e.*, *mprotect* cannot be invoked through a syscall instruction belonging to *write*.

4 Conclusion

In this thesis, we researched methods to harden the kernel against unprivileged attackers performing microarchitectural or control-flow-hijacking attacks. Our results allow us to draw 3 conclusions and provide directions for future research.

Existing Defenses are Circumventable. In this thesis, we showed that existing defenses against microarchitectural attacks can be circumvented [50, 51]. We demonstrated that the majority of proposed defenses against transient-execution attacks do not mitigate the attack as they only focus on the cache as the communication channel. Hence, an attacker can easily circumvent the defense by simply relying on another communication channel, such as port contention [12, 29]. Furthermore, we showed that the incomplete hardware mitigation for Meltdown even causes problems for the kernel as it allows an attacker to derandomize it with perfect accuracy.

Future work should investigate more defensive mechanisms that target the actual root cause instead of only focusing on the communication channel. However, the security implications of these new defenses must also be considered to ensure they do not cause additional problems.

Ease of Use. Defenses like seccomp can harden the kernel by limiting the exposed attack surface [161]. However, its problem is that it is hard to use in large-scale applications. A developer must have detailed knowledge of the application and its used libraries to identify all necessary syscalls. We demonstrated that this manual process can be reduced to an automated one [47, 52], making it more likely to be used in real-world applications. Unfortunately, this automation comes at the cost of precision due to the automated construction of the Control-Flow Graph (CFG). Generating a precise CFG in the presence of indirect control-flow transfers is known to be an undecidable problem [246]. This can result in overapproximation, *i.e.*, the provided security is not as high as theoretically possible. Nevertheless, a reduced protection is better than no protection at all. Any progress in precise CFG construction significantly improves the security of the automated approaches proposed in this thesis.

We also demonstrated a concept that allows for complex syscall arguments to be safely filtered [54] but relied on manual identification of the important arguments. Future work should investigate whether this process can be automated as well.

Similar Ideas, Different Contexts. CFI is a defensive mechanism that attempts to limit the control flow of an application [1]. While CFI has been demonstrated on the kernel as well [77, 98, 178, 210], we demonstrated that a similar idea can be applied to limit the attack surface exposed by the kernel. Instead of enforcing an application's control flow, we enforce its correct syscall flow so that only valid sequences of syscalls are possible [47]. Therefore, we harden the kernel by better restricting a userspace application's access to the syscall interface.

Future work can investigate whether other defensive mechanisms are applicable to the kernel, either to limit its exposed attack surface or to remove the impact of bugs in the kernel itself.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity." In: *CCS*. 2005.

[2] Accardi, Kristen Carlson. *Function Granular KASLR*. 2020.

[3] Anurag Acharya and Mandar Raje. "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications." In: *USENIX Security Symposium*. 2000.

[4] Onur Acıiçmez. "Advances in Side-Channel Cryptanalysis: MicroArchitectural Attacks." PhD thesis. Oregon State University, 2007.

[5] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. "New Results on Instruction Cache Attacks." In: *CHES*. 2010.

[6] Onur Acıiçmez and Çetin Kaya Koç. "Trace-Driven Cache Attacks on AES (Short Paper)." In: *International Conference on Information and Communications Security*. 2006.

[7] Onur Acıiçmez and Werner Schindler. "A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL." In: *CT-RSA*. 2008.

[8] Onur Acıiçmez. "Yet Another MicroArchitectural Attack: Exploiting I-cache." In: *CSAW*. 2007.

[9] Onur Acıiçmez and Cetin Kaya Koç. "Microarchitectural attacks and countermeasures." In: *Cryptographic Engineering*. 2009.

[10] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. "Predicting secret keys via branch prediction." In: *CT-RSA*. 2007.

[11] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. "The EM side—channel (s)." In: *CHES*. 2002.

[12] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit." In: *S&P*. 2019.

[13] AlephOne. "Smashing the Stack for Fun and Profit." In: *Phrack* (1996).

[14] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. *Consh: Confined Execution Environment for Internet Computations*. Tech. rep. The University of California, Santa Barbara, 1999.

[15] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. “Amplifying Side Channels Through Performance Degradation.” In: *ACSAC*. 2016.

[16] Ashwag Alrehily and Vijey Thayananthan. “Software watermarking based on return-oriented programming for computer security.” In: *International Journal of Computer Applications* (2017).

[17] Dennis Andriesse, Herbert Bos, and Asia Slowinska. “Parallax: Implicit code integrity verification using return-oriented programming.” In: *DSN*. 2015.

[18] ARM. *ARM Architecture Reference Manual Supplement ARMv8.1, for ARMv8-A architecture profile*. ARM Limited, 2016.

[19] ARM Limited. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. ARM Limited, 2012.

[20] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. “Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks.” In: *USENIX Security Symposium*. 2022.

[21] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. “Randomized Instruction Set Emulation.” In: *TISSEC* (2005).

[22] Mohammad Behnia et al. “Speculative Interference Attacks: Breaking Invisible Speculation Schemes.” In: *ASPLOS*. 2021.

[23] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. “Dune: Safe User-level Access to Privileged CPU Features.” In: *OSDI*. 2012.

[24] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. “Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way.” In: *CHES*. 2014.

[25] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[26] Johann Betz, Dirk Westhoff, and Günter Müller. “Survey on covert channels in virtual machines and cloud computing.” In: *Trans. Emerg. Telecommun. Technol.* (2017).

[27] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. “Hardware prefetchers leak : A revisit of SVF for cache-timing attacks.” In: *MICRO*. 2012.

[28] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. "SpecROP: Speculative Exploitation of ROP Chains." In: *RAID*. 2020.

[29] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention." In: *CCS*. 2019.

[30] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. "Hacking blind." In: *S&P*. 2014.

[31] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments." In: *NSDI*. 2008.

[32] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. "Mitigating Code-Reuse Attacks with Control-Flow Locking." In: *ACSAC*. 2011.

[33] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack." In: *AsiaCCS*. 2011.

[34] Roderick Bloem, Swen Jacobs, and Yakir Vizel. "Efficient Information-Flow Verification Under Speculative Execution." In: *Symposium on Automated Technology for Verification and Analysis*. 2019.

[35] Joseph Bonneau and Ilya Mironov. "Cache-collision timing attacks against AES." In: *CHES*. 2006.

[36] Pietro Borrello, Emilio Coppa, and Daniele Cono D'Elia. "Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation." In: *DSN*. 2021.

[37] Erik Bosman and Herbert Bos. "Framing Signals - A Return to Portable Shellcode." In: *S&P*. 2014.

[38] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector." In: *S&P*. 2016.

[39] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. "Protecting C++ Dynamic Dispatch Through VTable Interleaving." In: *NDSS*. 2016.

[40] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *WOOT*. 2017.

[41] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. “RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks.” In: *USENIX Security Symposium.* 2020.

[42] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M. Moya. “Cache Misses and the Recovery of the Full AES 256 Key.” In: *Applied Sciences* (2019).

[43] Billy Brumley and Risto Hakala. “Cache-Timing Template Attacks.” In: *AsiaCrypt.* 2009.

[44] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. “Control-Flow Integrity: Precision, Security, and Performance.” In: *ACM Computing Surveys* (2017).

[45] Nathan Burow, Xinping Zhang, and Mathias Payer. “SoK: Shining light on shadow stacks.” In: *S&P.* 2019.

[46] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. “Cache-Timing Attacks on RSA Key Generation.” In: *CHES* (2019).

[47] Claudio Canella, Sebastian Dorn, Daniel Gruss, and Michael Schwarz. “SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems.” In: *arXiv:2202.13716* (2022).

[48] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. “The Evolution of Transient-Execution Attacks.” In: *GLSVLSI.* 2020.

[49] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. “Evolution of Defenses against Transient-Execution Attacks.” In: *GLSVLSI.* 2020.

[50] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat.” In: *AsiaCCS.* 2020.

[51] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses.” In: *USENIX Security Symposium.* Extended classification tree and PoCs at https://transient.fail/. 2019.

[52] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. "Automating Seccomp Filter Generation for Linux Applications." In: *CCSW*. 2021.

[53] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-resistant CPUs." In: *CCS*. 2019.

[54] Canella, Claudio and Kogler, Andreas and Giner, Lukas and Gruss, Daniel and Schwarz, Michael. "Domain Page-Table Isolation." In: *arXiv:2111.10876* (2021).

[55] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity." In: *USENIX Security Symposium*. 2015.

[56] Nicholas Carlini and David A Wagner. "ROP is Still Dangerous: Breaking Modern Defenses." In: *USENIX Security Symposium*. 2014.

[57] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns." In: *CCS*. 2010.

[58] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution." In: *EuroS&P*. 2019.

[59] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. "Shreds: Fine-Grained Execution Units with Private Memory." In: *S&P*. 2016.

[60] Chih-Cheng Cheng. "The schemes and performances of dynamic branch predictors." In: *Berkeley Wireless Research Center, Tech. Rep* (2000).

[61] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack.(2014)." In: *NDSS*. 2014.

[62] Tzi-cker Chiueh and Fu-Hau Hsu. "RAD: A compile-time solution to buffer overflow attacks." In: *Conference on Distributed Computing Systems*. 2001.

[63] Chromium. *Linux Sandboxing*. URL: `https://chromium.googlesource.com/chromium/src/+/0e94f26e8/docs/linux_sandboxing.md`.

[64] Jonathan Corbet. *Supervisor mode access prevention*. 2012. URL: `https://lwn.net/Articles/517475/`.

[65] Crispan Cowan et al. "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." In: *USENIX Security Symposium*. 1998.

[66] John Criswell, Nathan Dautenhahn, and Vikram Adve. "KCoFI: Complete control-flow integrity for commodity operating system kernels." In: *S&P*. 2014.

[67] Patrick Cronin and Chengmo Yang. "A Fetching Tale: Covert Communication with the Hardware Prefetcher." In: *HOST*. 2019.

[68] Yujie Cui, Chun Yang, and Xu Cheng. "Abusing Cache Line Dirty States to Leak Information in Commercial Processors." In: *HPCA*. 2022.

[69] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks." In: *CHES*. 2018.

[70] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitaram. *Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code*. Tech. rep. 1997.

[71] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation." In: *ASPLOS*. 2015.

[72] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." In: *NDSS*. 2012.

[73] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation." In: *DAC*. 2014.

[74] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming." In: *NDSS*. 2015.

[75] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection." In: *USENIX Security Symposium.* 2014.

[76] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. "sysfilter: Automated System Call Filtering for Commodity Software." In: *RAID.* 2020.

[77] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. "Camouflage: Hardware-assisted CFI for the ARM Linux kernel." In: *DAC.* 2020.

[78] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX." In: *USENIX Security Symposium.* 2017.

[79] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. "Rapid Prototyping for Microarchitectural Attacks." In: *USENIX Security Symposium.* 2022.

[80] Eclypsium. *System Management Mode Speculative Execution Attacks.* 2018. URL: `https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/`.

[81] Jake Edge. *A seccomp overview.* 2015. URL: `https://lwn.net/Articles/656307/`.

[82] Úlfar Erlingsson, Yves Younan, and Frank Piessens. "Low-Level Software Security by Example." In: *Handbook of Information and Communication Security.* 2010.

[83] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity." In: *CCS.* 2015.

[84] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Covert channels through branch predictors: a feasibility study." In: *HASP.* 2015.

[85] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR." In: *MICRO.* 2016.

[86] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor." In: *ASPLOS.* 2018.

[87] Agner Fog. *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers.* 2021. URL: `https://www.agner.org/optimize/microarchitecture.pdf`.

[88] T. Fraser, L. Badger, and M. Feldman. "Hardening COTS software with generic software wrappers." In: *DISCEX*. 2000.

[89] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. "CFINSIGHT: A Comprehensive Metric for CFI Policies." In: *NDSS*. 2022.

[90] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. "IMIX: In-Process Memory Isolation EXtension." In: *USENIX Security Symposium.* 2018.

[91] Jacob Fustos, Michael Bechtel, and Heechul Yun. "SpectreRewind: Leaking Secrets to Past Instructions." In: *ASHES*. 2020.

[92] Cesar Pereida García and Billy Bob Brumley. "Constant-Time Callees with Variable-Time Callers." In: *USENIX Security Symposium.* 2017.

[93] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. "Ostia: A Delegating Architecture for Secure System Call Interposition." In: *NDSS*. 2004.

[94] Robert Gawlik and Thorsten Holz. "Towards automated integrity protection of C++ virtual function tables in binary programs." In: *ACSAC*. 2014.

[95] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. "Enabling client-side crash-resistance to overcome diversification and information hiding." In: *NDSS*. 2016.

[96] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2016).

[97] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2018).

[98] Ge, Xinyang and Talele, Nirupama and Payer, Mathias and Jaeger, Trent. "Fine-Grained Control-Flow Integrity for Kernel Software." In: *Euro S&P*. 2016.

[99] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. "Drive-by Key-Extraction Cache Attacks from Portable Code." In: *ACNS*. 2018.

[100] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. "Temporal System Call Specialization for Attack Surface Reduction." In: *USENIX Security Symposium*. 2020.

[101] Ghavamnia, Seyedhamed and Palit, Tapti and Mishra, Shachee and Polychronakis, Michalis. "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction." In: *RAID*. 2020.

[102] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. "SLIC: An Extensibility System for Commodity Operating Systems." In: *USENIX ATC*. 1998.

[103] Lukas Giner, Andreas Kogler, Claudio Canella, Michael Schwarz, and Daniel Gruss. "Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX." In: *USENIX Security Symposium*. 2022.

[104] Thomas Gleixner. *x86/kpti: Kernel Page Table Isolation (was KAISER)*. 2017. URL: `https://lkml.org/lkml/2017/12/4/709`.

[105] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era." In: *CCS*. 2020.

[106] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. "Out of Control: Overcoming Control-Flow Integrity." In: *S&P*. 2014.

[107] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard." In: *USENIX Security Symposium*. 2014.

[108] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. "Undermining information hiding (and what to do about it)." In: *USENIX Security Symposium*. 2016.

[109] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. "A secure environment for untrusted helper applications: Confining the wily hacker." In: *USENIX Security Symposium*. 1996.

[110] Daniel M. Gordon. "A Survey of Fast Exponentiation Methods." In: *J. Algorithms* (1998).

[111] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX." In: *EuroSec*. 2017.

[112] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks." In: *USENIX Security Symposium*. 2018.

[113] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. 2017.

[114] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme." In: *CHES*. 2016.

[115] Daniel Gruss. "Software-based Microarchitectural Attacks." PhD thesis. Graz University of Technology, 2017.

[116] Daniel Gruss. "Transient-Execution Attacks and Defenses." Habilitation Thesis. Graz University of Technology, 2020.

[117] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR." In: *ESSoS*. 2017.

[118] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *CCS*. 2016.

[119] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA*. 2016.

[120] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[121] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." In: *USENIX Security Symposium*. 2017.

[122] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[123] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled Detection of Speculative Information Flows." In: *S&P*. 2020.

[124] Serge Guelton and Siddhesh Poyarekar. *Use compiler flags for stack protection in GCC and Clang*. 2022.

[125] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *S&P*. 2011.

[126] Berk Gulmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross-VM cache attacks on AES." In: *IEEE TMSCS* (2016).

[127] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES." In: *COSADE*. 2015.

[128] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)." In: *Operating Systems Review* (1988).

[129] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. "Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries." In: *Usenix ATC*. 2019.

[130] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2017.

[131] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming." In: *WOOT*. 2012.

[132] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[133] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. "Enforcing Least Privilege Memory Views for Multithreaded Applications." In: *CCS*. 2016.

[134] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *S&P*. 2013.

[135] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX." In: *CHES*. 2020.

[136] Michael Hutter and Jörn-Marc Schmidt. “The temperature side channel and heating fault attacks.” In: *CARDIS*. 2013.

[137] Takashi Iiguni, Hitoshi Kamei, and Keizo Saisho. “Sprofiler: Automatic Generating System of Container-Native System Call Filtering Rules for Attack Surface Reduction.” In: *CSCI*. 2021.

[138] Google Inc. *Seccomp filter in Android O*. 2017. URL: `https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html`.

[139] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cache Attacks Enable Bulk Key Recovery on the Cloud.” In: *CHES*. 2016.

[140] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud.” In: *Cryptology ePrint Archive, Report 2015/898* (2015).

[141] Intel. *Control-flow Enforcement Technology Preview*. Revision 2.0. 2017.

[142] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[143] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.

[144] Intel. *Intel Analysis of Speculative Execution Side Channels*. Revision 4.0. 2018.

[145] Intel. *Q2 2018 Speculative Execution Side Channel Update*. 2018.

[146] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES.” In: *S&P*. 2015.

[147] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Systematic reverse engineering of cache slice selection in Intel processors.” In: *Euromicro Conference on Digital System Design*. 2015.

[148] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Know Thy Neighbor: Crypto Library Detection in Cloud.” In: *PETS* (2015).

[149] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Lucky 13 Strikes Back.” In: *AsiaCCS*. 2015.

[150] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Wait a minute! A fast, Cross-VM attack on AES.” In: *RAID*. 2014.

[151] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks.” In: *USENIX Security Symposium*. 2019.

[152] Himanshi Jain, D Anthony Balaraju, and Chester Rebeiro. “Spy Cartel: Parallelizing Evict+ Time-Based Cache Attacks on Last-Level Caches.” In: *Journal of Hardware and Systems Security* (2019).

[153] Kapil Jain and R Sekar. “User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement.” In: NDSS. 2000.

[154] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. “SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks.” In: *NDSS*. 2014.

[155] Yeongjin Jang, Sangho Lee, and Taesoo Kim. “Breaking Kernel Address Space Layout Randomization with Intel TSX.” In: *CCS*. 2016.

[156] Y. Jégou and O. Temam. “Speculative Prefetching.” In: *ICS*. 1993.

[157] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel.” In: *NDSS*. 2022.

[158] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. “A high-resolution side-channel attack on last-level cache.” In: *DAC*. 2016.

[159] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. “ret2dir: Rethinking kernel isolation.” In: *USENIX Security Symposium*. 2014.

[160] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. “kGuard: Lightweight Kernel Protection against Return-to-User Attacks.” In: *USENIX Security Symposium*. 2012.

[161] kernel.org. *Kernel Self-Protection*. 2022. URL: `https://www.kernel.org/doc/html/latest/security/self-protection.html`.

[162] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. “Adaptive call-site sensitive control flow integrity.” In: *EuroS&P*. 2019.

[163] Taehun Kim and Youngjoo Shin. “ThermalBleed: A Practical Thermal Side-Channel Attack.” In: *IEEE Access* (2022).

[164] Vladimir Kiriansky and Carl Waldspurger. “Speculative Buffer Overflows: Attacks and Defenses.” In: *arXiv:1807.03757* (2018).

[165] Paul Kocher. “Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems.” In: *CRYPTO*. 1996.

[166] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution.” In: *S&P*. 2019.

[167] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. “No Need to Hide: Protecting Safe Regions on Commodity Hardware.” In: *EuroSys*. 2017.

[168] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “Spectre Returns! Speculation Attacks using the Return Stack Buffer.” In: *WOOT*. 2018.

[169] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation.” In: *S&P*. 2020.

[170] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs.” In: *EuroS&P*. 2020.

[171] Sebastian Krahmer. *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*. 2005.

[172] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “NetCAT: Practical Cache Attacks from the Network.” In: *S&P*. 2020.

[173] Donghyun Kwon, Jiwon Seo, Sehyun Baek, Giyeol Kim, Sunwoo Ahn, and Yunheung Paek. “VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT.” In: *International Conference on Computational Science and Its Applications*. 2018.

[174] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. “Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses.” In: *2015 IEEE Trustcom/BigDataSE/ISPA*. 2015.

[175] Larabel, Michael. *Intel CET Indirect Branch Tracking Submitted For Linux 5.18*. 2022.

[176] Larabel, Michael. *Intel SMAP Comes To Try To Better Secure Linux*. 2012.

[177] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing.” In: *USENIX Security Symposium*. 2017.

[178] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. “Fine-cfi: fine-grained control-flow integrity for operating system kernels.” In: *TIFS* (2018).

[179] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. “Protecting Against Unexpected System Calls.” In: *USENIX Security Symposium*. 2005.

[180] Moritz Lipp. “Exploiting Microarchitectural Optimizations from Software.” PhD thesis. Graz University of Technology, 2021.

[181] Moritz Lipp, Daniel Gruss, and Michael Schwarz. “AMD Prefetch Attacks through Power and Time.” In: *USENIX Security Symposium*. 2022.

[182] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices.” In: *USENIX Security Symposium*. 2016.

[183] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take a Way: Exploring the Security Implications of AMD’s Cache Way Predictors.” In: *AsiaCCS*. 2020.

[184] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. “PLATYPUS: Software-based Power Side-Channel Attacks on x86.” In: *S&P*. 2021.

[185] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space.” In: *USENIX Security Symposium*. 2018.

[186] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. “Light-Weight Contexts: An OS Abstraction for Safety and Performance.” In: *OSDI*. 2016.

[187] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical.” In: *S&P*. 2015.

[188] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. “Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation.” In: *CCS*. 2015.

[189] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. “A Survey of Microarchitectural Side-Channel Vulnerabilities, Attacks, and Defenses in Cryptography.” In: *ACM Comput. Surv.* (2021).

[190] Kangjie Lu, Siyang Xiong, and Debin Gao. “Ropsteg: program steganography with return oriented programming.” In: *CODASPY*. 2014.

[191] Haoyu Ma, Kangjie Lu, Xinjie Ma, Haining Zhang, Chunfu Jia, and Debin Gao. “Software watermarking using return-oriented programming.” In: *AsiaCCS*. 2015.

[192] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers.” In: *CCS*. 2018.

[193] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008.

[194] Hector Marco-Gisbert and Ismael Ripoll. “Preventing brute force attacks against stack canary protection on networking servers.” In: *NCA*. 2013.

[195] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. “CCFI: Cryptographically enforced control flow integrity.” In: *CCS*. 2015.

[196] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “C5: Cross-Cores Cache Covert Channel.” In: *DIMVA*. 2015.

[197] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Complex Addressing Using Performance Counters.” In: *RAID*. 2015.

[198] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.” In: *NDSS*. 2017.

[199] Stephen McCamant and Greg Morrisett. “Evaluating SFI for a CISC Architecture.” In: *USENIX Security Symposium*. 2006.

[200] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture.” In: *USENIX Winter*. 1993.

[201] W. S. McPhee. “Operating system integrity in OS/VS2.” In: *IBM Systems Journal* (1974).

[202] Vahid Meraji and Hadi Soleimany. “Evict+Time Attack on Intel CPUs without Explicit Knowledge of Address Offsets.” In: *ISeCure* (2021).

[203] Microsoft. *Data Execution Prevention*. 2021. URL: `https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`.

[204] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX.” In: *CT-RSA*. 2018.

[205] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX amplifies the power of cache attacks.” In: *CHES*. 2017.

[206] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. “Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis.” In: *USENIX Security Symposium*. 2020.

[207] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. “MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation.” In: *RAID*. 2018.

[208] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. “Opaque Control-Flow Integrity.” In: *NDSS*. 2015.

[209] John Monaco. “SoK: Keylogging Side Channels.” In: *S&P*. 2018.

[210] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. “DROP THE ROP fine-grained control-flow integrity for the Linux kernel.” In: *Black Hat Asia* (2017).

[211] Mozilla. *Seccomp filter in Android O*. 2016. URL: `https://wiki.mozilla.org/Security/Sandbox/Seccomp`.

[212] Dongliang Mu, Jia Guo, Wenbiao Ding, Zhilong Wang, Bing Mao, and Lei Shi. “ROPOB: obfuscating binary code via return oriented programming.” In: *International Conference on Security and Privacy in Communication Systems.* 2017.

[213] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. “Analyzing Control Flow Integrity with LLVM-CFI.” In: *ACSAC.* 2019.

[214] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. “Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA.” In: *Information Systems* (2020).

[215] Amir Naseredini, Stefan Gast, Martin Schwarzl, Pedro Miguel Sousa Bernardo, Amel Smajic, Claudio Canella, Martin Berger, and Daniel Gruss. “Systematic Analysis of Programming Languages and Their Execution Environments for Spectre Attacks.” In: *ICISSP.* 2022.

[216] Nergal. *The advanced return-into-lib(c) explits: PaX case study.* 2001.

[217] Michael Neve and Jean-Pierre Seifert. “Advances on Access-Driven Cache Attacks on AES.” In: *Selected Areas in Cryptography (SAC).* 2006.

[218] Ben Niu and Gang Tan. “Modular control-flow integrity.” In: *PLDI.* 2014.

[219] Ben Niu and Gang Tan. “Monitor integrity protection with space efficiency and separate compilation.” In: *CCS.* 2013.

[220] Ben Niu and Gang Tan. “Per-input control-flow integrity.” In: *CCS.* 2015.

[221] Ben Niu and Gang Tan. “RockJIT: Securing just-in-time compilation using modular control-flow integrity.” In: *CCS.* 2014.

[222] O’Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. *Spectre attack against SGX enclave.* 2018.

[223] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. “Specfuzz: Bringing Spectre-type vulnerabilities to the surface.” In: *USENIX Security Symposium.* 2020.

[224] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications.” In: *CCS*. 2015.

[225] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: the Case of AES.” In: *CT-RSA*. 2006.

[226] Pailoor, Shankara and Wang, Xinyu and Shacham, Hovav and Dillig, Isil. “Automated Policy Synthesis for System Call Sandboxing.” In: *PACMPL* (2020).

[227] Meghna Pancholi, Andreas D Kellas, Vasileios P Kemerlis, and Simha Sethumadhavan. “Timeloops: System Call Policy Learning for Containerized Microservices.” In: *arXiv:2204.06131* (2022).

[228] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. “Transparent ROP Exploit Mitigation Using Indirect Branch Tracing.” In: *USENIX Security Symposium*. 2013.

[229] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. “libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK).” In: *USENIX ATC*. 2019.

[230] PaX Team. *Address space layout randomization (ASLR)*. 2003.

[231] PaX Team. *Design & Implementation of PAGEEXEC*. 2000.

[232] Mathias Payer, Antonio Barresi, and Thomas R. Gross. “Fine-Grained Control-Flow Integrity Through Binary Hardening.” In: *DIMVA*. 2015.

[233] Arthur Perais and André Seznec. “Practical data value speculation for future high-end processors.” In: *HPCA*. 2014.

[234] Colin Percival. “Cache Missing for Fun and Profit.” In: *BSDCan*. 2005.

[235] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “Make Sure DSA Signing Exponentiations Really Are Constant-Time.” In: *CCS*. 2016.

[236] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Syngress Publishing, 2010. ISBN: 1597494860.

[237] David S. Peterson, Matt Bishop, and Raju Pandey. “A Flexible Containment Mechanism for Executing Untrusted Code.” In: *USENIX Security Symposium*. 2002.

[238] Jannik Pewny and Thorsten Holz. “Control-flow restrictor: Compiler-based CFI for iOS.” In: *ACSAC*. 2013.

[239] Joop van de Pol, Nigel P Smart, and Yuval Yarom. “Just a little bit more.” In: *CT-RSA*. 2015.

[240] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. “Reducing set-associative cache energy via way-prediction and selective direct-mapping.” In: *MICRO*. 2001.

[241] Aravind Prakash, Xunchao Hu, and Heng Yin. “vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.” In: *NDSS*. 2015.

[242] Niels Provos. “Improving Host Security with System Call Policies.” In: *USENIX Security Symposium*. 2003.

[243] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. “Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks.” In: *CCS*. 2021.

[244] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. “SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets.” In: *NDSS*. 2021.

[245] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “CROSSTALK: Speculative Data Leaks Across Cores Are Real.” In: *S&P*. 2021.

[246] G. Ramalingam. “The Undecidability of Aliasing.” In: *ACM Trans. Program. Lang. Syst.* (1994).

[247] Josyula R Rao and Pankaj Rohatgi. “EMpowering Side-Channel Attacks.” In: *IACR Cryptology ePrint Archive* (2001).

[248] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. “Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack.” In: *SBCCI*. 2016.

[249] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. “I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches.” In: *ISCA*. 2021.

[250] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. “Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds.” In: *CCS*. 2009.

[251] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. “Reverse Engineering the Stream Prefetcher for Profit.” In: *SILM Workshop*. 2020.

[252] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. “Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions.” In: *Journal of Computer Virology and Hacking Techniques* (2018).

[253] Jonathan Salwan. *ROPgadget - Gadgets finder and auto-roper*. 2011. URL: `http://shell-storm.org/project/ROPgadget/`.

[254] Sercan Sari. *What are Rings in Operating Systems?* Retrieved on July 4, 2022. URL: `https://www.baeldung.com/cs/os-rings`.

[255] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. *Addendum 2 to RIDL: Rogue In-flight Data Load*. 2020.

[256] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. *Addendum to RIDL: Rogue In-flight Data Load*. 2019.

[257] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “RIDL: Rogue In-flight Data Load.” In: *S&P*. 2019.

[258] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. “CacheOut: Leaking Data on Intel CPUs via Cache Evictions.” In: *S&P*. 2021.

[259] Robert Schilling, Pascal Nasahl, and Stefan Mangard. “FIPAC: Thwarting Fault-and Software-Induced Control-Flow Attacks with ARM Pointer Authentication.” In: *COSADE*. 2022.

[260] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. “Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86.” In: *USENIX Security Symposium*. 2020.

[261] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. “Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications.” In: *S&P*. 2015.

[262] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. “Evaluating the effectiveness of current anti-ROP defenses.” In: *RAID*. 2014.

[263] Michael Schwarz. “Software-based Side-Channel Attacks and Defenses in Restricted Environments.” PhD thesis. Graz University of Technology, 2019.

[264] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. “Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs.” In: *arXiv:1905.05725* (2019).

[265] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. “Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features.” In: *AsiaCCS*. 2018.

[266] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks.” In: *DIMVA*. 2017.

[267] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. “ConTExT: A Generic Approach for Mitigating Spectre.” In: *NDSS*. 2020.

[268] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. “KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks.” In: *NDSS*. 2018.

[269] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling.” In: *CCS*. 2019.

[270] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript.” In: *FC*. 2017.

[271] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. “NetSpectre: Read Arbitrary Memory over Network.” In: *ESORICS*. 2019.

[272] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. “ Specfuscator: Evaluating Branch Removal as a Spectre Mitigation.” In: *FC*. 2021.

[273] scoding.de. *Ropper - rop gadget finder and binary information tool*. 2013. URL: `https://scoding.de/ropper/`.

[274] Hovav Shacham. “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).” In: *CCS*. 2007.

[275] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. “Secure In-VM Monitoring Using Hardware Virtualization.” In: *CCS*. 2009.

[276] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. “Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage.” In: *CCS*. 2018.

[277] Rajesh Kumar Shrivastava and Chittaranjan Hota. “Lightweight code self-verification using return-oriented programming in resilient IoT.” In: *Real-Time Data Analytics for Large Scale Sensor Data*. 2020.

[278] Rajesh Kumar Shrivastava and Chittaranjan Hota. “Profile-guided code identification and hardening using return oriented programming.” In: *Journal of Information Security and Applications* (2019).

[279] Anatoly Shusterman, Ayush Agarwal, Sioli O’Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. “Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses.” In: *USENIX Security Symposium*. 2021.

[280] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization.” In: *S&P*. 2013.

[281] Solar Designer. *Getting around non-executable stack (and fix)*. 1997. URL: `http://seclists.org/bugtraq/1997/Aug/63`.

[282] Wei Song and Peng Liu. “Dynamically Finding Minimal Eviction Sets Can be Quicker Than You Think for Side-Channel Attacks Against the LLC.” In: *RAID*. 2019.

[283] Spengler, Brad. *Recent ARM Security Improvements*. 2013.

[284] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. “Systematic classification of side-channel attacks: a case study for mobile devices.” In: *IEEE Communications Surveys & Tutorials* 20.1 (2017), pp. 465–488.

[285] Raphael Spreitzer and Thomas Plos. “Cache-Access Pattern Attack on Disaligned AES T-Tables.” In: *COSADE*. 2013.

[286] Julian Stecklina and Thomas Prescher. “LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels.” In: *arXiv:1806.07480* (2018).

[287] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. “Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds.” In: *NDSS*. 2018.

[288] Jakub Szefer. “Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses.” In: *Cryptology ePrint Archive, Report 2016/479* (2016).

[289] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory.” In: *S&P*. 2013.

[290] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2014.

[291] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. “TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering.” In: *USENIX Security Symposium*. 2022.

[292] PaX Team. *Homepage of The PaX Team*. 2012. URL: `https://pax.grsecurity.net/`.

[293] PaX Team. *UDEREF*. 2007. URL: `https://grsecurity.net/~spender/uderef.txt`.

[294] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.” In: *USENIX Security Symposium*. 2014.

[295] M Caner Tol, Berk Gulmezoglu, Koray Yurtseven, and Berk Sunar. “Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings.” In: *EuroS&P*. 2021.

[296] Robert M Tomasulo. “An efficient algorithm for exploiting multiple arithmetic units.” In: *IBM Journal of research and Development* (1967).

[297] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. “MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols.” In: *arXiv:1802.03802* (2018).

[298] Eran Tromer, Dag Arne Osvik, and Adi Shamir. “Efficient Cache Attacks on AES, and Countermeasures.” In: *Journal of Cryptology* (2010).

[299] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. "ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys." In: *USENIX Security Symposium.* 2019.

[300] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control." In: *Workshop on System Software for Trusted Execution.* 2017.

[301] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution." In: *USENIX Security Symposium.* 2017.

[302] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium.* 2018.

[303] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *S&P.* 2020.

[304] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think." In: *USENIX Security Symposium.* 2018.

[305] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. "Practical context-sensitive CFI." In: *CCS.* 2015.

[306] Victor van der Veen, Nitish dutt Sharma, Herbert Bos, and Lorenzo Cavallaro. "Memory Errors: the Past, the Present, and the Future." In: *RAID.* 2012.

[307] Vendicator. *Stack Shield: A stack smashing technique protection tool for Linux.* 2011.

[308] Pepe Vila, Boris Köpf, and Jose Morales. "Theory and Practice of Finding Eviction Sets." In: *S&P.* 2019.

[309] David A. Wagner. *Janus: An Approach for Confinement of Untrusted Applications.* Tech. rep. University of California at Berkeley, 1999.

[310] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. "Efficient software-based fault isolation." In: *ACM SOSP.* 1993.

[311] Jack Wampler, Ian Martiny, and Eric Wustrow. "ExSpectre: Hiding Malware in Speculative Execution." In: *NDSS*. 2019.

[312] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. "Volcano: Stateless Cache Side-channel Attack by Exploiting Mesh Interconnect." In: *arXiv:2103.04533* (2021).

[313] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. "Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries." In: *NDSS*. 2019.

[314] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. "PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack." In: *DAC*. 2019.

[315] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. "KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution." In: *TOSEM* (2020).

[316] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. "oo7: Low-overhead Defense against Spectre attacks via Program Analysis." In: *Transactions on Software Engineering* (2019).

[317] Zhi Wang and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." In: *S&P*. 2010.

[318] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. "Osiris: Automated Discovery Of Microarchitectural Side Channels." In: *USENIX Security Symposium*. 2021.

[319] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGXJail: Defeating Enclave Malware via Confinement." In: *RAID*. 2019.

[320] Samuel Weiser and Mario Werner. "Sgxio: Generic trusted i/o path for intel sgx." In: *CODASPY*. 2017.

[321] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf`.

[322] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud." In: *ACM Transactions on Networking* (2014).

[323] Wenjie Xiong and Jakub Szefer. “Leaking Information Through Cache LRU States.” In: *HPCA*. 2020.

[324] Wenjie Xiong and Jakub Szefer. “Survey of Transient Execution Attacks and Their Mitigations.” In: *ACM Computing Surveys* (2021).

[325] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. “An exploration of L2 cache covert channels in virtualized environments.” In: *CCSW*. 2011.

[326] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. “Attack directories, not caches: Side channel attacks in a non-inclusive world.” In: *S&P*. 2019.

[327] Yuval Yarom and Naomi Benger. “Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack.” In: *Cryptology ePrint Archive, Report 2014/140* (2014).

[328] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.” In: *USENIX Security Symposium*. 2014.

[329] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. “Mapping the Intel Last-Level Cache.” In: *Cryptology ePrint Archive, Report 2015/905* (2015).

[330] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: A Timing Attack on OpenSSL Constant Time RSA.” In: *JCEN* (2017).

[331] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. “In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication.” In: *USENIX Security Symposium*. 2022.

[332] Yves Younan, Wouter Joosen, and Frank Piessens. “Runtime Countermeasures for Code Injection Attacks against C and C++ Programs.” In: *ACM Computing Surveys* (2012).

[333] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. “Hardware-assisted fine-grained code-reuse attack detection.” In: *RAID*. 2015.

[334] Andreas Zankl, Hermann Seuschek, Gorka Irazoqui, and Berk Gulmezoglu. "Side-channel attacks in the Internet of Things: threats and challenges." In: *Research Anthology on Artificial Intelligence Applications in Security*. 2021.

[335] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. "VTint: Protecting Virtual Function Tables' Integrity." In: *NDSS*. 2015.

[336] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. "Practical control flow integrity and randomization for binary executables." In: *S&P*. 2013.

[337] Mingwei Zhang and R. Sekar. "Control Flow Integrity for COTS Binaries." In: *USENIX Security Symposium*. 2013.

[338] Tianning Zhang, Miao Cai, Diming Zhang, and Hao Huang. "Se-BROP: blind ROP attacks without returns." In: *Frontiers of Computer Science* (2022).

[339] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. "A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds." In: *Information and Communications Security*. 2016.

[340] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds." In: *CCS*. 2014.

[341] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM Side Channels and Their Use to Extract Private Keys." In: *CCS*. 2012.

[342] Tianhao Zheng, Haishan Zhu, and Mattan Erez. "SIPT: Speculatively Indexed, Physically Tagged Caches." In: *HPCA*. 2018.

Part II

Publications

5 A Systematic Evaluation of Transient Execution Attacks and Defenses

Publication Data

Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses.” In: *USENIX Security Symposium.* Extended classification tree and PoCs at https://transient.fail/. 2019

Contributions

Main author.

A Systematic Evaluation of Transient Execution Attacks and Defenses

Claudio Canella[1], Jo Van Bulck[2], Michael Schwarz[1], Moritz Lipp[1], Benjamin von Berg[1], Philipp Ortner[1], Frank Piessens[2], Dmitry Evtyushkin[3], Daniel Gruss[1]

[1] Graz University of Technology, Austria [2] imec-DistriNet, KU Leuven, Belgium [3] College of William and Mary, USA

Abstract

Research on *transient execution* attacks including Spectre and Meltdown showed that exception or branch misprediction events might leave secret-dependent traces in the CPU's microarchitectural state. This observation led to a proliferation of new Spectre and Meltdown attack variants and even more ad-hoc defenses (e.g., microcode and software patches). Both the industry and academia are now focusing on finding effective defenses for known issues. However, we only have limited insight on residual attack surface and the completeness of the proposed defenses.

In this paper, we present a systematization of transient execution attacks. Our systematization uncovers 6 (new) transient execution attacks that have been overlooked and not been investigated so far: 2 new exploitable Meltdown effects: Meltdown-PK (Protection Key Bypass) on Intel, and Meltdown-BND (Bounds Check Bypass) on Intel and AMD; and 4 new Spectre mistraining strategies. We evaluate the attacks in our classification tree through proof-of-concept implementations on 3 major CPU vendors (Intel, AMD, ARM). Our systematization yields a more complete picture of the attack surface and allows for a more systematic evaluation of defenses. Through this systematic evaluation, we discover that most defenses, including deployed ones, cannot fully mitigate all attack variants.

1 Introduction

CPU performance over the last decades was continuously improved by shrinking processing technology and increasing clock frequencies, but

The original publication is available at `https://www.usenix.org/conference/usenixsecurity19/presentation/canella`.

physical limitations are already hindering this approach. To still increase the performance, vendors shifted the focus to increasing the number of cores and optimizing the instruction pipeline. Modern CPU pipelines are massively parallelized allowing hardware logic in prior pipeline stages to perform operations for subsequent instructions ahead of time or even out-of-order. Intuitively, pipelines may stall when operations have a dependency on a previous instruction which has not been executed (and retired) yet. Hence, to keep the pipeline full at all times, it is essential to predict the control flow, data dependencies, and possibly even the actual data. Modern CPUs, therefore, rely on intricate microarchitectural optimizations to predict and sometimes even re-order the instruction stream. Crucially, however, as these predictions may turn out to be wrong, pipeline flushes may be necessary, and instruction results should always be committed according to the intended in-order instruction stream. Pipeline flushes may occur even without prediction mechanisms, as on modern CPUs virtually any instruction can raise a fault (e.g., page fault or general protection fault), requiring a roll-back of all operations following the faulting instruction. With prediction mechanisms, there are more situations when partial pipeline flushes are necessary, namely on every misprediction. The pipeline flush discards any architectural effects of pending instructions, ensuring functional correctness. Hence, the instructions are executed *transiently* (first they are, and then they vanish), *i.e.*, we call this *transient execution* [50, 56, 86].

While the architectural effects and results of transient instructions are discarded, microarchitectural side effects remain beyond the transient execution. This is the foundation of Spectre [50], Meltdown [56], and Foreshadow [86]. These attacks exploit transient execution to encode secrets through microarchitectural side effects (e.g., cache state) that can later be recovered by an attacker at the architectural level. The field of transient execution attacks emerged suddenly and proliferated, leading to a situation where people are not aware of all variants and their implications. This is apparent from the confusing naming scheme that already led to an arguably wrong classification of at least one attack [48]. Even more important, this confusion leads to misconceptions and wrong assumptions for defenses. Many defenses focus exclusively on hindering exploitation of a specific covert channel, instead of addressing the microarchitectural root cause of the leakage [45, 47, 50, 91]. Other defenses rely on recent CPU features that have not yet been evaluated from a transient security perspective [84]. We also debunk implicit assumptions including that AMD or the latest Intel CPUs are completely immune to Meltdown-type effects, or that serializing instructions mitigate Spectre Variant 1 on any CPU.

In this paper, we present a systematization of transient execution attacks, *i.e.*, Spectre, Meltdown, Foreshadow, and related attacks. Using our decision tree, transient execution attacks are accurately classified through an unambiguous naming scheme (cf. Figure 5.1). The hierarchical and extensible nature of our taxonomy allows to easily identify residual attack surface, leading to 6 previously overlooked transient execution attacks (Spectre and Meltdown variants) first described in this work. Two of the attacks are Meltdown-BND, exploiting a Meltdown-type effect on the x86 `bound` instruction on Intel and AMD, and Meltdown-PK, exploiting a Meltdown-type effect on memory protection keys on Intel. The other 4 attacks are previously overlooked mistraining strategies for Spectre-PHT and Spectre-BTB attacks. We demonstrate the attacks in our classification tree through practical proofs-of-concept with vulnerable code patterns evaluated on CPUs of Intel, ARM, and AMD.[1]

Next, we provide a classification of gadgets and their prevalence in real-world software based on an anaylsis of the Linux kernel. We also give a short overview on current tools for automatic gadget detection.

We then provide a systematization of the state-of-the-art defenses. Based on this, we systematically evaluate defenses with practical experiments and theoretical arguments to show which work and which do not or cannot suffice. This systematic evaluation revealed that we can still mount transient execution attacks that are supposed to be mitigated by rolled out patches. Finally, we discuss how defenses can be designed to mitigate entire types of transient execution attacks.

Contributions. The contributions of this work are:

1. We systematize Spectre- and Meltdown-type attacks, advancing attack surface understanding, highlighting misclassifications, and revealing new attacks.
2. We provide a clear distinction between Meltdown/Spectre, required for designing effective countermeasures.
3. We provide a classification of gadgets and discuss their prevalence in real-world software.
4. We categorize defenses and show that most, including deployed ones, cannot fully mitigate all attack variants.
5. We describe new branch mistraining strategies, highlighting the difficulty of eradicating Spectre-type attacks.

We responsibly disclosed the work to Intel, ARM, and AMD.

[1]`https://github.com/IAIK/transientfail`

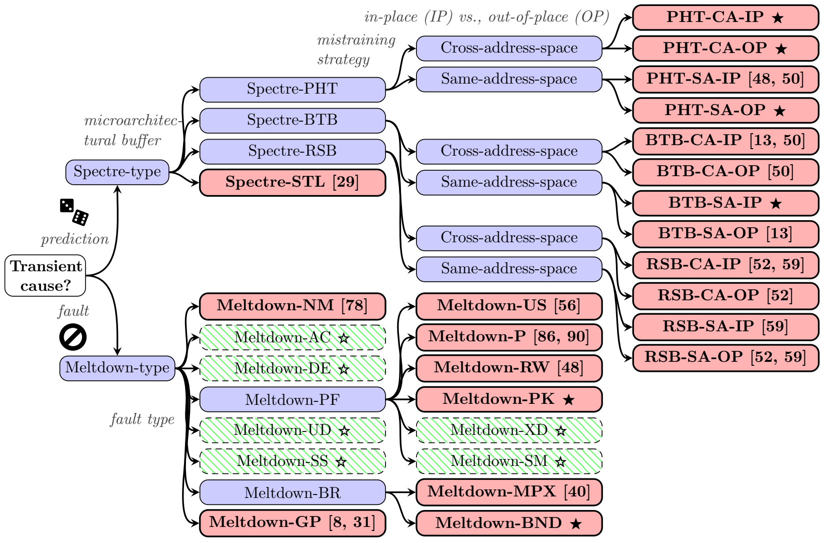


Figure 5.1: Transient execution attack classification tree with demonstrated attacks (red, bold), negative results (green, dashed), some first explored in this work (★ / ☆).[1]

Experimental Setup. Unless noted otherwise, the experimental results reported were performed on recent Intel Skylake i5-6200U, Coffee Lake i7-8700K, and Whiskey Lake i7-8565U CPUs. Our AMD test machines were a Ryzen 1950X and a Ryzen Threadripper 1920X. For experiments on ARM, an NVIDIA Jetson TX1 has been used.
Outline. Section 2 provides background. We systematize Spectre in Section 3 and Meltdown in Section 4. We analyze and classify gadgets in Section 5 and defenses in Section 6. We discuss future work and conclude in Section 7.

2 Transient Execution

Instruction Set Architecture and Microarchitecture. The instruction set architecture (ISA) provides an interface between hardware and software. It defines the instructions that a processor supports, the available registers, the addressing mode, and describes the execution model. Examples of different ISAs are x86 and ARMv8. The microarchitecture then describes how the ISA is implemented in a processor in the form of pipeline depth, interconnection of elements, execution units, cache, branch

[1]An up-to-date version of the tree is available at `http://transient.fail/`

prediction. The ISA and the microarchitecture are both stateful. In the ISA, this state includes, for instance, data in registers or main memory after a successful computation. Therefore, the architectural state can be observed by the developer. The microarchitectural state includes, for instance, entries in the cache and the translation lookaside buffer (TLB), or the usage of the execution units. Those microarchitectural elements are transparent to the programmer and can not be observed directly, only indirectly.

Out-of-Order Execution. On modern CPUs, individual instructions of a complex instruction set are first decoded and split-up into simpler micro-operations (μOPs) that are then processed. This design decision allows for superscalar optimizations and to extend or modify the implementation of specific instructions through so-called microcode updates. Furthermore, to increase performance, CPU's usually implement a so-called *out-of-order* design. This allows the CPU to execute μOPs not only in the sequential order provided by the instruction stream but to dispatch them in parallel, utilizing the CPU's execution units as much as possible and, thus, improving the overall performance. If the required operands of a μOP are available, and its corresponding execution unit is not busy, the CPU starts its execution even if μOPs earlier in the instruction stream have not finished yet. As immediate results are only made visible at the architectural level when all previous μOPs have finished, CPUs typically keep track of the status of μOPs in a so-called *Reorder Buffer* (ROB). The CPU takes care to *retire* μOPs in-order, deciding to either discard their results or commit them to the architectural state. For instance, exceptions and external interrupt requests are handled during retirement by flushing any outstanding μOP results from the ROB. Therefore, the CPU may have executed so-called *transient instructions* [56], whose results are never committed to the architectural state.

Speculative Execution. Software is mostly not linear but contains (conditional) branches or data dependencies between instructions. In theory, the CPU would have to stall until a branch or dependencies are resolved before it can continue the execution. As stalling decreases performance significantly, CPUs deploy various mechanisms to predict the outcome of a branch or a data dependency. Thus, CPUs continue executing along the predicted path, buffering the results in the ROB until the correctness of the prediction is verified as its dependencies are resolved. In the case of a correct prediction, the CPU can commit the pre-computed results from the reorder buffer, increasing the overall performance. However, if the prediction was incorrect, the CPU needs to

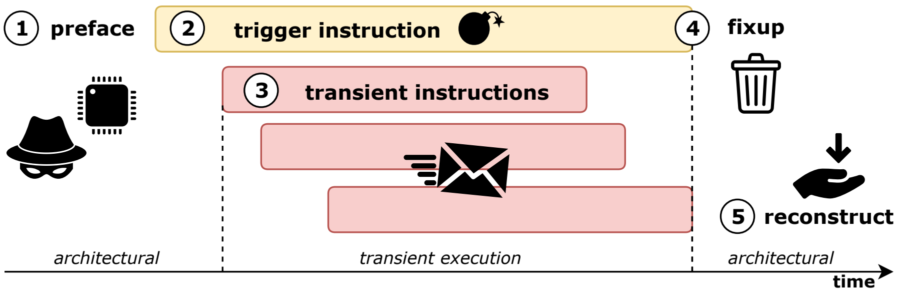


Figure 5.2: High-level overview of a transient execution attack in 5 phases: (1) prepare microarchitecture, (2) execute a *trigger instruction*, (3) *transient instructions* encode unauthorized data through a microarchitectural covert channel, (4) CPU retires trigger instruction and flushes transient instructions, (5) reconstruct secret from microarchitectural state.

perform a roll-back to the last correct state by squashing all pre-computed transient instruction results from the ROB.

Cache Covert Channels. Modern CPUs use caches to hide memory latency. However, these latency differences can be exploited in side-channels and covert channels [24, 51, 60, 67, 92]. In particular, Flush+Reload allows observations across cores at cache-line granularity, enabling attacks, e.g., on cryptographic algorithms [26, 43, 92], user input [24, 55, 72], and kernel addressing information [23]. For Flush+Reload, the attacker continuously flushes a shared memory address using the `clflush` instruction and afterward reloads the data. If the victim used the cache line, accessing it will be fast; otherwise, it will be slow.

Covert channels are a special use case of side-channel attacks, where the attacker controls both the sender and the receiver. This allows an attacker to bypass many restrictions that exist at the architectural level to leak information.

Transient Execution Attacks. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. For functional correctness, it is crucial that their results are never committed to the architectural state. However, transient instructions may still leave traces in the CPU's microarchitectural state, which can subsequently be exploited to partially recover unauthorized results [50, 56, 86]. This observation has led to a variety of transient execution attacks, which from a high-level always follow the same abstract flow, as shown in Figure 5.2. The attacker first brings the microarchitecture into the desired state, e.g., by flushing and/or populating internal branch predictors or data caches. Next is the execution of a so-called *trigger instruction*. This can be any instruction that causes subsequent operations to be eventually squashed, e.g., due to an exception or a mispredicted branch or data dependency. Before completion of the trigger instruction, the CPU proceeds with the execution of a *transient instruction sequence*. The attacker abuses the transient instructions to act as the sending end of a microarchitectural covert channel, e.g., by loading a secret-dependent memory location into the CPU cache. Ultimately, at the retirement of the trigger instruction, the CPU discovers the exception/misprediction and flushes the pipeline to discard any architectural effects of the transient instructions. However, in the final phase of the attack, unauthorized transient computation results are recovered at the receiving end of the covert channel, e.g., by timing memory accesses to deduce the secret-dependent loads from the transient instructions.

High-Level Classification: Spectre vs. Meltdown. Transient execution attacks have in common that they abuse transient instructions (which are never architecturally committed) to encode unauthorized data in the microarchitectural state. With different instantiations of the abstract phases in Figure 5.2, a wide spectrum of transient execution attack variants emerges. We deliberately based our classification on the root cause of the transient computation (phases 1, 2), abstracting away from the specific covert channel being used to transmit the unauthorized data (phases 3, 5). This leads to a first important split in our classification tree (cf. Figure 5.1). Attacks of the first type, dubbed Spectre [50], exploit transient execution following control or data flow misprediction. Attacks of the second type, dubbed Meltdown [56], exploit transient execution following a faulting instruction.

Importantly, Spectre and Meltdown exploit fundamentally different CPU properties and hence require orthogonal defenses. Where the former relies on dedicated control or data flow prediction machinery, the latter merely exploits that data from a faulting instruction is forwarded to instructions ahead in the pipeline. Note that, while Meltdown-type attacks so far exploit out-of-order execution, even elementary in-order pipelines may allow for similar effects [85]. Essentially, the different root cause of the trigger instruction (Spectre-type misprediction vs. Meltdown-type fault) determines the nature of the subsequent unauthorized transient computations and hence the scope of the attack.

That is, in the case of Spectre, transient instructions can only compute on data which the application is also allowed to access architecturally.

Spectre thus transiently bypasses *software-defined* security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths. Hence, much like in a "confused deputy" scenario, successful Spectre attacks come down to steering a victim into transiently computing on memory locations the victim is authorized to access but the attacker not. In practice, this implies that one or more phases of the transient execution attack flow in Figure 5.2 should be realized through so-called *code gadgets* executing within the victim application. We propose a novel taxonomy of gadgets based on these phases in Section 5.

For Meltdown-type attacks, on the other hand, transient execution allows to completely "melt down" architectural isolation barriers by computing on unauthorized results of faulting instructions. Meltdown thus transiently bypasses *hardware-enforced* security policies to leak data that should always remain architecturally inaccessible for the application. Where Spectre-type leakage remains largely an unintended side-effect of important speculative performance optimizations, Meltdown reflects a failure of the CPU to respect hardware-level protection boundaries for transient instructions. That is, the mere continuation of the transient execution after a fault itself is required, but not sufficient for a successful Meltdown attack. As further explored in Section 6, this has profound consequences for defenses. Overall, mitigating Spectre requires careful hardware-software co-design, whereas merely replacing the data of a faulting instruction with a dummy value suffices to block Meltdown-type leakage in silicon, e.g., as it is done in AMD processors, or with the Rogue Data Cache Load resistance (RDCL_NO) feature advertised in recent Intel CPUs from Whiskey Lake onwards [40].

3 Spectre-type Attacks

In this section, we provide an overview of Spectre-type attacks (cf. Figure 5.1). Given the versatility of Spectre variants in a variety of adversary models, we propose a novel two-level taxonomy based on the preparatory phases of the abstract transient execution attack flow in Figure 5.2. First, we distinguish the different microarchitectural buffers that can trigger a prediction (phase 2), and second, the mistraining strategies that can be used to steer the prediction (phase 1).

Systematization of Spectre Variants. To predict the outcome of various types of branches and data dependencies, modern CPUs accumulate an extensive microarchitectural state across various internal buffers

Table 5.1: Spectre-type attacks and the microarchitectural element they exploit (●), partially target (◐), or not affect (○).

Attack \ **Element**	BTB	BHB	PHT	RSB	STL
Spectre-PHT (Variant 1) [50]	○	◐	●	○	○
Spectre-PHT (Variant 1.1) [48]	○	◐	●	○	○
Spectre-BTB (Variant 2) [50]	●	◐	○	○	○
Spectre-RSB (ret2spec) [52, 59]	◐	○	○	●	○
Spectre-STL (Variant 4) [29]	○	○	○	○	●

Glossary: Branch Target Buffer (BTB), Branch History Buffer (BHB), Pattern History Table (PHT), Return Stack Buffer (RSB), Store To Load (STL).

and components [19]. Table 5.1 overviews Spectre-type attacks and the corresponding microarchitectural elements they exploit. As the first level of our classification tree, we categorize Spectre attacks based on the microarchitectural root cause that triggers the misprediction leading to the transient execution:

- Spectre-PHT [48, 50] exploits the *Pattern History Table* (PHT) that predicts the outcome of conditional branches.
- Spectre-BTB [50] exploits the *Branch Target Buffer* (BTB) for predicting branch destination addresses.
- Spectre-RSB [52, 59] primarily exploits the *Return Stack Buffer* (RSB) for predicting return addresses.
- Spectre-STL [29] exploits memory disambiguation for predicting *Store To Load* (STL) data dependencies.

Note that NetSpectre [74], SGXSpectre [63], and SGXPectre [13] focus on applying one of the above Spectre variants in a specific exploitation scenario. Hence, we do not consider them separate variants in our classification.

Systematization of Mistraining Strategies. We now propose a second-level classification scheme for Spectre variants that abuse history-based branch prediction (*i.e.*, all of the above except Spectre-STL). These Spectre variants first go through a preparatory phase (cf. Figure 5.2) where the microarchitectural branch predictor state is "poisoned" to cause intentional misspeculation of a particular victim branch. Since branch prediction buffers in modern CPUs [19, 50] are commonly indexed based on the virtual address of the branch instruction, mistraining can happen

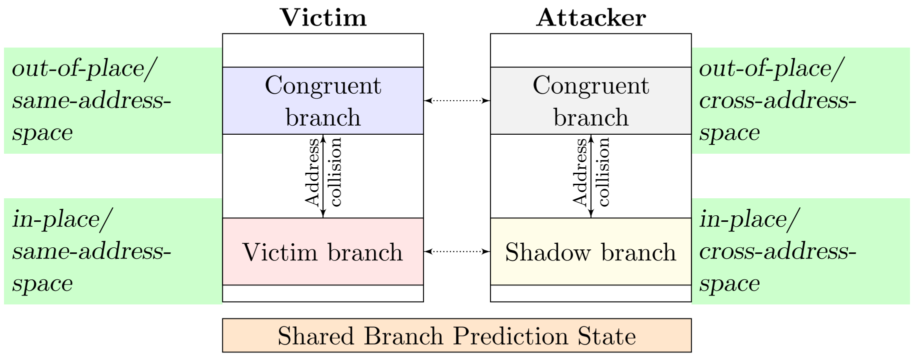

Victim
Attacker
out-of-place/ same-address-space
Congruent branch
Congruent branch
out-of-place/ cross-address-space
Address collision
Address collision
in-place/ same-address-space
Victim branch
Shadow branch
in-place/ cross-address-space
Shared Branch Prediction State

Figure 5.3: A branch can be mistrained either by the victim process (*same-address-space*) or by an attacker-controlled process (*cross-address-space*). Mistraining can be achieved either using the vulnerable branch itself (*in-place*) or a branch at a congruent virtual address (*out-of-place*).

either within the same address space or from a different attacker-controlled process. Furthermore, as illustrated in Figure 5.3, when only a subset of the virtual address is used in the prediction, mistraining can be achieved using a branch instruction at a congruent virtual address. We thus enhance the field of Spectre-type branch poisoning attacks with 4 distinct mistraining strategies:

1. Executing the victim branch in the victim process (*same-address-space in-place*).
2. Executing a congruent branch in the victim process (*same-address-space out-of-place*).
3. Executing a shadow branch in a different process (*cross-address-space in-place*).
4. Executing a congruent branch in a different process (*cross-address-space out-of-place*).

In current literature [6, 13, 48, 50], several of the above branch poisoning strategies have been overlooked for different Spectre variants. We summarize the results of an assessment of vulnerabilities under mistraining strategies in Table 5.2. Our systematization thus reveals clear blind spots that allow an attacker to mistrain branch predictors in previously unknown ways. As explained further, depending on the adversary's capabilities (e.g., in-process, sandboxed, remote, enclave, etc.) these previously unknown mistraining strategies may lead to new attacks and/or bypass existing defenses.

3.1 Spectre-PHT (Input Validation Bypass)

Microarchitectural Element. Kocher et al. [50] first introduced Spectre Variant 1, an attack that poisons the Pattern History Table (PHT) to mispredict the direction (taken or not-taken) of conditional branches. Depending on the underlying microarchitecture, the PHT is accessed based on a combination of virtual address bits of the branch instruction plus a hidden Branch History Buffer (BHB) that accumulates global behavior for the last N branches on the same physical core [18, 19]

Reading Out-of-Bounds. Conditional branches are commonly used by programmers and/or compilers to maintain memory safety invariants at runtime. For example, consider the following code snippet for bounds checking [50]:

```
    if (x < len(array1)) { y = array2[array1[x] * 4096]; }
```

At the architectural level, this program clearly ensures that the index variable `x` always lies within the bounds of the fixed-length buffer `array1`. However, after repeatedly supplying valid values of `x`, the PHT will reliably predict that this branch evaluates to true. When the adversary now supplies an invalid index `x`, the CPU continues along a mispredicted path and transiently performs an out-of-bounds memory access. The above code snippet features an explicit example of a “leak gadget” that may act as a microarchitectural covert channel: depending on the out-of-bounds value being read, the transient instructions load another memory page belonging to `array2` into the cache.

Writing Out-of-Bounds. Kiriansky and Waldspurger [48] showed that transient writes are also possible by following the same principle. Consider the following code line:

```
    if (x < len(array)) { array[x] = value; }
```

After mistraining the PHT component, attackers controlling the untrusted index `x` can transiently write to arbitrary out-of-bounds addresses. This creates a transient buffer overflow, allowing the attacker to bypass both type and memory safety. Ultimately, when repurposing traditional techniques from return-oriented programming [75] attacks, adversaries may even gain arbitrary code execution in the transient domain by overwriting return addresses or code pointers.

Overlooked Mistraining Strategies. Spectre-PHT attacks so far [48, 50, 63] rely on a same-address-space in-place branch poisoning strategy. However, our results (cf. Table 5.2) reveal that the Intel, ARM, and AMD CPUs we tested are vulnerable to all four PHT mistraining strategies. In this, we are the first to successfully demonstrate Spectre-PHT-style branch misprediction attacks *without prior execution of the victim branch.*

Table 5.2: Spectre-type attacks performed in-place, out-of-place, same-address-space (*i.e.*, intra-process), or cross-address-space (*i.e.*, cross-process).

Method		**Attack**	Spectre-PHT	Spectre-BTB	Spectre-RSB	Spectre-STL
Intel	intra-process	in-place	● [48, 50]	★	● [59]	● [29]
		out-of-place	★	● [13]	● [52, 59]	○
	cross-process	in-place	★	● [13, 50]	● [52, 59]	○
		out-of-place	★	● [50]	● [52]	○
ARM	intra-process	in-place	● [48, 50]	★	● [6]	● [6]
		out-of-place	★	☆	● [6]	○
	cross-process	in-place	★	● [6, 50]	☆	○
		out-of-place	★	☆	☆	○
AMD	intra-process	in-place	● [50]	★	★	● [29]
		out-of-place	★	☆	★	○
	cross-process	in-place	★	● [50]	★	○
		out-of-place	★	☆	★	○

Symbols indicate whether an attack is possible and known (●), not possible and known (○), possible and previously unknown or not shown (★), or tested and did not work and previously unknown or not shown (☆). All tests performed with no defenses enabled.

This is an important contribution as it may open up previously unknown attack avenues for restricted adversaries.

Cross-address-space PHT poisoning may, for instance, enable advanced attacks against a privileged daemon process that does not directly accept user input. Likewise, for Intel SGX technology, remote attestation schemes have been developed [76] to enforce that a victim enclave can only be run exactly once. This effectively rules out current state-of-the-art SGXSpectre [63] attacks that repeatedly execute the victim enclave to mistrain the PHT branch predictor. Our novel out-of-place PHT poisoning strategy, on the other hand, allows us to perform the training phase entirely *outside* the enclave on the same physical core by repeatedly executing a congruent branch in the untrusted enclave host process (cf. Figure 5.3).

3.2 Spectre-BTB (Branch Target Injection)

Microarchitectural Element. In Spectre Variant 2 [50], the attacker poisons the Branch Target Buffer (BTB) to steer the transient execution to a mispredicted branch target. For direct branches, the CPU indexes the BTB using a subset of the virtual address bits of the branch instruction to yield the predicted jump target. For indirect branches, CPUs use different mechanisms [28], which may take into account global branching history accumulated in the BHB when indexing the BTB. We refer to both types as Spectre-BTB.

Hijacking Control Flow. Contrary to Spectre-PHT, where transient instructions execute along a restricted mispredicted path, Spectre-BTB allows redirecting transient control flow to an arbitrary destination. Adopting established techniques from return-oriented programming (ROP) attacks [75], but abusing BTB poisoning instead of application-level vulnerabilities, selected code "gadgets" found in the victim address space may be chained together to construct arbitrary transient instruction sequences. Hence, where the success of Spectre-PHT critically relies on unintended leakage along the mispredicted code path, ROP-style gadget abuse in Spectre-BTB allows to more directly construct covert channels that expose secrets from the transient domain (cf. Figure 5.2). We discuss gadget types in more detail in Section 5.

Overlooked Mistraining Strategies. Spectre-BTB was initially demonstrated on Intel, AMD, and ARM CPUs using a cross-address-space in-place mistraining strategy [50]. With SGXPectre [13], Chen et al. extracted secrets from Intel SGX enclaves using either a cross-address-space in-place or same-address-space out-of-place BTB poisoning strategy. We experimentally reproduced these mistraining strategies through a systematic evaluation presented in Table 5.2. On AMD and ARM, we could not demonstrate out-of-place BTB poisoning. Possibly, these CPUs use an unknown (sub)set of virtual address bits or a function of bits which we were not able to reverse engineer. We encourage others to investigate whether a different (sub)set of virtual address bits is required to enable the attack.

To the best of our knowledge, we are the first to recognize that Spectre-BTB mistraining can also proceed by *repeatedly executing the vulnerable indirect branch with valid inputs*. Much like Spectre-PHT, such same-address-space in-place BTB (Spectre-BTB-SA-IP) poisoning abuses the victim's own execution to mistrain the underlying branch target predictor. Hence, as an important contribution to understanding attack surface and defenses, in-place mistraining *within* the victim domain may allow bypassing widely deployed mitigations [4, 40] that flush and/or partition the BTB before entering the victim. Since the branch destination address is now determined by the victim code and not under the direct control of the attacker, however, Spectre-BTB-SA-IP cannot offer the full power of arbitrary transient control flow redirection. Yet, in higher-level languages like C++ that commonly rely on indirect branches to implement polymorph abstractions, Spectre-BTB-SA-IP may lead to subtle "speculative type confusion" vulnerabilities. For example, a victim that repeatedly executes a virtual function call with an object of `TypeA` may inadvertently mistrain the branch target predictor to cause misspeculation when finally executing the virtual function call with an object of another `TypeB`.

3.3 Spectre-RSB (Return Address Injection)

Microarchitectural Element. Maisuradze and Rossow [59] and Koruyeh et al. [52] introduced a Spectre variant that exploits the Return Stack Buffer (RSB). The RSB is a small per-core microarchitectural buffer that stores the virtual addresses following the N most recent `call` instructions. When encountering a `ret` instruction, the CPU pops the topmost element from the RSB to predict the return flow.

Hijacking Return Flow. Misspeculation arises whenever the RSB layout diverges from the actual return addresses on the software stack. Such disparity for instance naturally occurs when restoring kernel/enclave/user stack pointers upon protection domain switches. Furthermore, same-address-space adversaries may explicitly overwrite return addresses on the software stack, or transiently execute `call` instructions which update the RSB without committing architectural effects [52]. This may allow untrusted code executing in a sandbox to transiently divert return control flow to interesting code gadgets outside of the sandboxed environment.

Due to the fixed-size nature of the RSB, a special case of misspeculation occurs for deeply nested function calls [52, 59]. Since the RSB can only store return addresses for the N most recent calls, an underfill occurs when the software stack is unrolled. In this case, the RSB can no longer provide accurate predictions. Starting from Skylake, Intel CPUs use the BTB as a fallback [19, 52], thus allowing Spectre-BTB-style attacks triggered by `ret` instructions.

Overlooked Mistraining Strategies. Spectre-RSB has been demonstrated with all four mistraining strategies, but only on Intel [52, 59]. Our experimental results presented in Table 5.2 generalize these strategies to AMD CPUs. Furthermore, in line with ARM's own analysis [6], we successfully poisoned RSB entries within the same-address-space but did not observe any cross-address-space leakage on ARM CPUs. We expect this

Table 5.3: Demonstrated Meltdown-type (MD) attacks.

Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
MD-GP (Variant 3a) [8]	●	○	○	○						
MD-NM (Lazy FP) [78]	○	●	○	○						
MD-BR	○	○	●	○						
MD-US (Meltdown) [56]	○	○	○	●	●	○	○	○	○	○
MD-P (Foreshadow) [86, 90]	○	○	○	●	○	●	○	●	○	○
MD-RW (Variant 1.2) [48]	○	○	○	●	○	○	●	○	○	○
MD-PK	○	○	○	●	○	○	○	○	○	●

Symbols (● or ○) indicate whether an exception type (left) or permission bit (right) is exploited. Systematic names are derived from what is exploited.

may be a limitation of our current proof-of-concept code and encourage others to investigate this further.

3.4 Spectre-STL (Speculative Store Bypass)

Microarchitectural Element. Speculation in modern CPUs is not restricted to control flow but also includes predicting dependencies in the data flow. A common type of Store To Load (STL) dependencies require that a memory load shall not be executed before all preceding stores that write to the same location have completed. However, even before the addresses of all prior stores in the pipeline are known, the CPUs' memory disambiguator [3, 34, 44] may predict which loads can already be executed speculatively.

When the disambiguator predicts that a load does not have a dependency on a prior store, the load reads data from the L1 data cache. When the addresses of all prior stores are known, the prediction is verified. If any overlap is found, the load and all following instructions are re-executed.

Reading Stale Values. Horn [29] showed how mispredictions by the memory disambiguator could be abused to speculatively bypass store instructions. Like previous attacks, Spectre-STL adversaries rely on an appropriate transient instruction sequence to leak unsanitized stale values via a microarchitectural covert channel. Furthermore, operating on stale pointer values may speculatively break type and memory safety guarantees in the transient execution domain [29].

Table 5.4: Secrets recoverable via Meltdown-type attacks and whether they cross the current privilege level (CPL).

Leaks / **Attack**	Memory	Cache	Register	Cross-CPL
Meltdown-US (Meltdown) [56]	●	●	○	✓
Meltdown-P (Foreshadow-NG) [90]	○	●	○	✓
Meltdown-P (Foreshadow-SGX) [86]	◐	●	◐	✓
Meltdown-GP (Variant 3a) [8]	○	○	●	✓
Meltdown-NM (Lazy FP) [78]	○	○	●	✓
Meltdown-RW (Variant 1.2) [48]	●	●	○	✘
Meltdown-PK	☆	★	☆	✘
Meltdown-BR	★	★	☆	✘

Symbols indicate whether an attack crosses a processor privilege level (✓) or not (✘), whether it can leak secrets from a buffer (●), only with additional steps (◐), or not at all (○). Respectively (★ vs. ☆) if first shown in this work.

4 Meltdown-type Attacks

This section overviews Meltdown-type attacks, and presents a classification scheme that led to the discovery of two previously overlooked Meltdown variants (cf. Figure 5.1). Importantly, where Spectre-type attacks exploit (branch) misprediction events to trigger transient execution, Meltdown-type attacks rely on transient instructions following a CPU exception. Essentially, Meltdown exploits that exceptions are only raised (*i.e.*, become architecturally visible) upon the retirement of the faulting instruction. In some microarchitectures, this property allows transient instructions ahead in the pipeline to compute on unauthorized results of the instruction that is about to suffer a fault. The CPU's in-order instruction retirement mechanism takes care to discard any architectural effects of such computations, but as with the Spectre-type attacks above, secrets may leak through microarchitectural covert channels.

Systematization of Meltdown Variants. We introduce a classification for Meltdown-type attacks in two dimensions. In the first level, we categorize attacks based on the exception that causes transient execution. Following Intel's [35] classification of exceptions as *faults*, *traps*, or *aborts*, we observed that Meltdown-type attacks so far have exploited faults, but not traps or aborts. The CPU generates faults if a correctable error has occurred, *i.e.*, they allow the program to continue after it has been resolved. Traps are reported immediately after the execution of the instruction, *i.e.*, when the instruction retires and becomes architecturally visible. Aborts report some unrecoverable error and do not allow a restart of the task that caused the abort.

In the second level, for page faults (`#PF`), we further categorize based on page-table entry protection bits (cf. Table 5.3). We also categorize attacks based on which storage locations can be reached, and whether it crosses a privilege boundary (cf. Table 5.4). Through this systematization, we discovered several previously unknown Meltdown variants that exploit different exception types as well as page-table protection bits, including two exploitable ones. Our systematic analysis furthermore resulted in the first demonstration of exploitable Meltdown-type delayed exception handling effects on AMD CPUs.

4.1 Meltdown-US (Supervisor-only Bypass)

Modern CPUs commonly feature a “user/supervisor” page-table attribute to denote a virtual memory page as belonging to the OS kernel. The original Meltdown attack [56] reads kernel memory from user space on CPUs that do *not* transiently enforce the user/supervisor flag. In the trigger phase (cf. Figure 5.2) an unauthorized kernel address is dereferenced, which eventually causes a page fault. Before the fault becomes architecturally visible, however, the attacker executes a transient instruction sequence that for instance accesses a cache line based on the privileged data read by the trigger instruction. In the final phase, after the exception has been raised, the privileged data is reconstructed at the receiving end of the covert channel (e.g., Flush+Reload).

The attacks bandwidth can be improved by suppressing exceptions through transaction memory CPU features such as Intel TSX [35], exception handling [56], or hiding it in another transient execution [28, 56]. By iterating byte-by-byte over the kernel space and suppressing or handling exceptions, an attacker can dump the entire kernel. This includes the entire physical memory if the operating system has a direct physical map in the kernel. While extraction rates are significantly higher when the kernel data resides in the CPU cache, Meltdown has even been shown to successfully extract uncached data from memory [56].

4.2 Meltdown-P (Virtual Translation Bypass)

Foreshadow. Van Bulck et al. [86] presented Foreshadow, a Meltdown-type attack targeting Intel SGX technology [32]. Unauthorized accesses to enclave memory usually do not raise a `#PF` exception but are instead silently replaced with abort page dummy values (cf. Section 6.2). In the absence of a fault, plain Meltdown cannot be mounted against SGX enclaves. To overcome this limitation, a Foreshadow attacker clears the "present" bit in the page-table entry mapping the enclave secret, ensuring that a `#PF` will be raised for subsequent accesses. Analogous to Meltdown-US, the adversary now proceeds with a transient instruction sequence to leak the secret (e.g., through a Flush+Reload covert channel).

Intel [30] named *L1 Terminal Fault* (L1TF) as the root cause behind Foreshadow. A terminal fault occurs when accessing a page-table entry with either the present bit cleared or a "reserved" bit set. In such cases, the CPU immediately aborts address translation. However, since the L1 data cache is indexed in parallel to address translation, the page table entry's physical address field (*i.e.*, frame number) may still be passed to the L1 cache. Any data present in L1 and tagged with that physical address will now be forwarded to the transient execution, regardless of access permissions.

Although Meltdown-P-type leakage is restricted to the L1 data cache, the original Foreshadow [86] attack showed how SGX's secure page swapping mechanism might first be abused to prefetch arbitrary enclave pages into the L1 cache, including even CPU registers stored on interrupt. This highlights that SGX's privileged adversary model considerably amplifies the transient execution attack surface.

Foreshadow-NG. Foreshadow-NG [90] generalizes Foreshadow from the attack on SGX enclaves to bypass operating system or hypervisor isolation. The generalization builds on the observation that the physical frame number in a page-table entry is sometimes under direct or indirect control of an adversary. For instance, when swapping pages to disk, the kernel is free to use all but the present bit to store metadata (e.g., the offset on the swap partition). However, if this offset is a valid physical address, any cached memory at that location leaks to an unprivileged Foreshadow-OS attacker.

Even worse is the Foreshadow-VMM variant, which allows an untrusted virtual machine, controlling guest-physical addresses, to extract the host machine's entire L1 data cache (including data belonging to the hypervisor or other virtual machines). The underlying problem is that a terminal fault in the guest page-tables early-outs the address translation process, such that guest-physical addresses are erroneously passed to the L1 data cache, without first being translated into a proper host physical address [30].

4.3 Meltdown-GP (System Register Bypass)

Meltdown-GP (named initially Variant 3a) [37] allows an attacker to read privileged system registers. It was first discovered and published by ARM [8] and subsequently Intel [31] determined that their CPUs are also susceptible to the attack. Unauthorized access to privileged system registers (e.g., via `rdmsr`) raises a *general protection* fault (`#GP`). Similar to previous Meltdown-type attacks, however, the attack exploits that the transient execution following the faulting instruction can still compute on the unauthorized data, and leak the system register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.4 Meltdown-NM (FPU Register Bypass)

During a context switch, the OS has to save all the registers, including the floating point unit (FPU) and SIMD registers. These latter registers are large and saving them would slow down context switches. Therefore, CPUs allow for a lazy state switch, meaning that instead of saving the registers, the FPU is simply marked as “not available”. The first FPU instruction issued after the FPU was marked as “not available” causes a *device-not-available* (`#NM`) exception, allowing the OS to save the FPU state of previous execution context before marking the FPU as available again.

Stecklina and Prescher [78] propose an attack on the above lazy state switch mechanism. The attack consists of three steps. In the first step, a victim performs operations loading data into the FPU registers. Then, in the second step, the CPU switches to the attacker and marks the FPU as “not available”. The attacker now issues an instruction that uses the FPU, which generates an `#NM` fault. Before the faulting instruction retires, however, the CPU has already transiently executed the following instructions using data from the previous context. As such, analogous to previous Meltdown-type attacks, a malicious transient instruction sequence following the faulting instruction can encode the unauthorized FPU register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.5 Meltdown-RW (Read-only Bypass)

Where the above attacks [8, 56, 78, 86] focussed on stealing information across privilege levels, Kiriansky and Waldspurger [48] presented the first Meltdown-type attack that bypasses page-table based access rights *within* the current privilege level. Specifically, they showed that transient execution does not respect the "read/write" page-table attribute. The ability to transiently overwrite read-only data within the current privilege level can bypass software-based sandboxes which rely on hardware enforcement of read-only memory.

Confusingly, the above Meltdown-RW attack was originally named "Spectre Variant 1.2" [48] as the authors followed a Spectre-centric naming scheme. Our systematization revealed, however, that the transient cause exploited above is a `#PF` exception. Hence, this attack is of Meltdown-type, but *not* a variant of Spectre.

4.6 Meltdown-PK (Protection Key Bypass)

Intel Skylake-SP server CPUs support memory-protection keys for user space (PKU) [33]. This feature allows processes to change the access permissions of a page directly from user space, *i.e.*, without requiring a syscall/hypercall. Thus, with PKU, user-space applications can implement efficient hardware-enforced isolation of trusted parts [27, 84].

We present a novel Meltdown-PK attack to bypass both read and write isolation provided by PKU. Meltdown-PK works if an attacker has code execution in the containing process, even if the attacker cannot execute the `wrpkru` instruction (e.g., blacklisting). Moreover, in contrast to cross-privilege level Meltdown attack variants, there is no software workaround. According to Intel [36], Meltdown-PK can be mitigated using address space isolation. Recent Meltdown-resistant Intel processors enumerating RDCL_NO plus PKU support furthermore mitigate Meltdown-PK in silicon. With those mitigations, the memory addresses that might be revealed by transient execution attacks can be limited.

Experimental Results. We tested Meltdown-PK on an Amazon EC2 C5 instance running Ubuntu 18.04 with PKU support. We created a memory mapping and used PKU to remove both read and write access. As expected, protected memory accesses produce a `#PF`. However, our proof-of-concept manages to leak the data via an adversarial transient instruction sequence with a Flush+Reload covert channel.

4.7 Meltdown-BR (Bounds Check Bypass)

To facilitate efficient software instrumentation, x86 CPUs come with dedicated hardware instructions that raise a *bound-range-exceeded* exception (`#BR`) when encountering out-of-bound array indices. The IA-32 ISA, for instance, defines a `bound` opcode for this purpose. While the `bound` instruction was omitted in the subsequent x86-64 ISA, modern Intel CPUs ship with Memory Protection eXtensions (MPX) for efficient array bounds checking.

Our systematic evaluation revealed that Meltdown-type effects of the `#BR` exception had not been thoroughly investigated yet. Specifically, Intel's analysis [40] only briefly mentions MPX-based bounds check bypass as a possibility, and recent defensive work by Dong et al. [16] highlights the need to introduce a memory `lfence` after MPX bounds check instructions. They classify this as a Spectre-type attack, implying that the `lfence` is needed to prevent the branch predictor from speculating on the outcome of the bounds check. According to Oleksenko et al. [64], neither `bndcl` nor `bndcu` exert pressure on the branch predictor, indicating that there is no prediction happening. Based on that, we argue that the classification as a Spectre-type attack is misleading as no prediction is involved. The observation by Dong et al. [16] indeed does not shed light on the `#BR` exception as the root cause for the MPX bounds check bypass, and they do not consider IA32 `bound` protection at all. Similar to Spectre-PHT, Meltdown-BR is a bounds check bypass, but instead of mistraining a predictor it exploits the lazy handling of the raised `#BR` exception.

Experimental Results. We introduce the Meltdown-BR attack which exploits transient execution following a `#BR` exception to encode out-of-bounds secrets that are never architecturally visible. As such, Meltdown-BR is an exception-driven alternative for Spectre-PHT. Our proofs-of-concept demonstrate out-of-bounds leakage through a Flush+Reload covert channel for an array index safeguarded by either IA32 `bound` (Intel, AMD), or state-of-the-art MPX protection (Intel-only). For Intel, we ran the attacks on a Skylake i5-6200U CPU with MPX support, and for AMD we evaluated both an E2-2000 and a Ryzen Threadripper 1920X. This is the first experiment demonstrating a Meltdown-type transient execution attack exploiting delayed exception handling on AMD CPUs [4, 56].

4.8 Residual Meltdown (Negative Results)

We systematically studied transient execution leakage for other, not yet tested exceptions. In our experiments, we consistently found no traces

Table 5.5: CPU vendors vulnerable to Meltdown (MD).

Vendor \ Attack	MD-US [56]	MD-P [86, 90]	MD-GP [8, 31]	MD-NM [78]	MD-RW [48]	MD-PK	MD-BR	MD-DE	MD-AC	MD-UD	MD-SS	MD-XD	MD-SM
Intel	●	●	●	●	●	★	★	☆	☆	☆	☆	☆	☆
ARM	●	○	●	❘	●	❘	❘	☆	☆	☆	❘	☆	☆
AMD	○	○	○	○	○	❘	★	☆	☆	☆	☆	☆	☆

Symbols indicate whether at least one CPU model is vulnerable (filled) vs. no CPU is known to be vulnerable (empty). Glossary: reproduced (● vs. ○), first shown in this paper (★ vs. ☆), not applicable (❘). All tests performed without defenses enabled.

of transient execution beyond traps or aborts, which leads us to the hypothesis that Meltdown is only possible with faults (as they can occur at any moment during instruction execution). Still, the possibility remains that our experiments failed and that they are possible. Table 5.5 and Figure 5.1 summarize experimental results for fault types tested on Intel, ARM, and AMD.

Division Errors. For the divide-by-zero experiment, we leveraged the signed division instruction (`idiv` on x86 and `sdiv` on ARM). On the ARMs we tested, there is no exception, but the division yields merely zero. On x86, the division raises a *divide-by-zero* exception (`#DE`). Both on the AMD and Intel we tested, the CPU continues with the transient execution after the exception. In both cases, the result register is set to '0', which is the same result as on the tested ARM. Thus, according to our experiments Meltdown-DE is not possible, as no real values are leaked.

Supervisor Access. Although supervisor mode access prevention (SMAP) raises a page fault (`#PF`) when accessing user-space memory from the kernel, it seems to be free of any Meltdown effect in our experiments. Thus, we were not able to leak any data using Meltdown-SM in our experiments.

Alignment Faults. Upon detecting an unaligned memory operand, the CPU may generate an *alignment check* exception (`#AC`). In our tests, the results of unaligned memory accesses never reach the transient execution. We suspect that this is because `#AC` is generated early-on, even before the operand's virtual address is translated to a physical one. Hence, our experiments with Meltdown-AC were unsuccessful in showing any leakage.

Segmentation Faults. We consistently found that out-of-limit segment accesses never reach transient execution in our experiments. We suspect that, due to the simplistic IA32 segmentation design, segment limits are validated early-on, and immediately raise a `#GP` or `#SS` (*stack-segment fault*) exception, without sending the offending instruction to the ROB. Therefore, we observed no leakage in our experiments with Meltdown-SS.
Instruction Fetch. To yield a complete picture, we investigated Meltdown-type effects during the instruction fetch and decode phases. On our test systems, we did not succeed in transiently executing instructions residing in non-executable memory (*i.e.*, Meltdown-XD), or following an *invalid opcode* (`#UD`) exception (*i.e.*, Meltdown-UD). We suspect that exceptions during instruction fetch or decode are immediately handled by the CPU, without first buffering the offending instruction in the ROB. Moreover, as invalid opcodes have an undefined length, the CPU does not even know where the next instruction starts. Hence, we suspect that invalid opcodes only leak if the microarchitectural effect is already an effect caused by the invalid opcode itself, not by subsequent transient instructions.

5 Gadget Analysis and Classification

We deliberately oriented our attack tree (cf. Figure 5.1) on the microarchitectural root causes of the transient computation, abstracting away from the underlying covert channel and/or code *gadgets* required to carry out the attack successfully. In this section, we further dissect transient execution attacks by categorizing gadget types in two tiers and overviewing current results on their exploitability in real-world software.

5.1 Gadget Classification

First-Tier: Execution Phase. We define a “gadget” as a series of instructions executed by either the attacker or the victim. Table 5.6 shows how gadget types discussed in literature can be unambiguously assigned to one of the abstract attack phases from Figure 5.2. New gadgets can be added straightforwardly after determining their execution phase and objective.

Table 5.6: Gadget classification according to the attack flow and whether executed by the attacker (●), victim (○), or either (◐).

Attack	**1. Preface**	**2. Trigger example**	**3. Transient**	**5. Reconstruction**
Covert channel [1, 74, 92]	◐ Flush/Prime/Evict	-	◐ Load/AVX/Port/...	◐ Reload/Probe/Time
Meltdown-US/RW/GP/NM/PK [8, 48, 56, 78]	● (Exception suppression)	● `mov`/`rdmsr`/FPU	● Controlled encode	● Exception handling
Meltdown-P [86, 90]	○ (L1 prefetch)	● `mov`	● Controlled encode	& controlled decode
Meltdown-BR	-	○ `bound`/`bndclu`	○ Inadvertent leak	*same as above*
Spectre-PHT [50]	◐ PHT poisoning	○ `jz`	○ Inadvertent leak	● Controlled decode
Spectre-BTB/RSB [13, 50, 52, 59]	◐ BTB/RSB poisoning	○ `call`/`jmp`/`ret`	○ ROP-style encode	● Controlled decode
Spectre-STL [29]	-	○ `mov`	○ Inadvertent leak	● Controlled decode
NetSpectre [74]	○ Thrash/reset	○ `jz`	○ Inadvertent leak	○ Inadvertent transmit

Importantly, our classification table highlights that gadget choice largely depends on the attacker's capabilities. By plugging in different gadget types to compose the required attack phases, an almost boundless spectrum of adversary models can be covered that is only limited by the attacker's capabilities. For local adversaries with arbitrary code execution (e.g., Meltdown-US [56]), the gadget functionality can be explicitly implemented by the attacker. For sandboxed adversaries (e.g., Spectre-PHT [50]), on the other hand, much of the gadget functionality has to be provided by "confused deputy" code executing in the victim domain. Ultimately, as claimed by Schwarz et al. [74], even fully remote attackers may be able to launch Spectre attacks given that sufficient gadgets would be available inside the victim code.

Second-Tier: Transient Leakage. During our analysis of the Linux kernel (see Section 5.2), we discovered that gadgets required for Spectre-PHT can be further classified in a second tier. A second tier is required in this case as those gadgets enable different types of attacks. The first type of gadget we found is called *Prefetch*. A Prefetch gadget consists of a single array access. As such it is not able to leak data, but can be used to load data that can then be leaked by another gadget as was demonstrated by Meltdown-P [86]. The second type of gadget, called *Compare*, loads a value like in the Prefetch gadget and then branches on it. Using a contention channel like execution unit contention [2, 9] or an AVX channel as claimed by Schwarz et al. [74], an attacker might be able to leak data. We refer to the third gadget as *Index* gadget and it is the double array access shown by Kocher et al. [50]. The final gadget type, called *Execute*, allows arbitrary code execution, similar to Spectre-BTB. In such a gadget, an array is indexed based on an attacker-controlled input and the resulting value is used as a function pointer, allowing an attacker to transiently execute code by accessing the array out-of-bounds. Table 5.7 gives examples for all four types.

5.2 Real-World Software Gadget Prevalence

While for Meltdown-type attacks, convincing real-world exploits have been developed to dump arbitrary process [56] and enclave [86] memory, most Spectre-type attacks have so far only been demonstrated in controlled environments. The most significant barrier to mounting a successful Spectre attack is to find exploitable gadgets in real-world software, which at present remains an important open research question in itself [59, 74].

Table 5.7: Spectre-PHT gadget classification and the number of occurrences per gadget type in Linux kernel v5.0.

Gadget	Example (Spectre-PHT)	#Occurrences
Prefetch	`if(i<LEN_A){a[i];}`	172
Compare	`if(i<LEN_A){if(a[i]==k){};}`	127
Index	`if(i<LEN_A){y = b[a[i]*x];}`	0
Execute	`if(i<LEN_A){a[i](void);}`	16

Automated Gadget Analysis. Since the discovery of transient execution attacks, researchers have tried to develop methods for the automatic analysis of gadgets. One proposed method is called oo7 [89] and uses taint tracking to detect Spectre-PHT Prefetch and Index gadgets. oo7 first marks all variables that come from an untrusted source as tainted. If a tainted variable is later on used in a branch, the branch is also tainted. The tool then reports a possible gadget if a tainted branch is followed by a memory access depending on the tainted variable. Guarnieri et al. [25] mention that oo7 would still flag code locations that were patched with Speculative Load Hardening [12] as it would still match the vulnerable pattern.

Another approach, called Spectector [25], uses symbolic execution to detect Spectre-PHT gadgets. It tries to formally prove that a program does not contain any gadgets by tracking all memory accesses and jump targets during execution along all different program paths. Additionally, it simulates the path of mispredicted branches for a number of steps. The program is run twice to determine whether it is free of gadgets or not. First, it records a trace of memory accesses when no misspeculation occurs (*i.e.*, runs the program in its intended way). Second, it records a trace of memory accesses with misspeculation of a certain number of instructions. Spectector then reports a gadget if it detects a mismatch between the two traces. One problem with the Spectector approach is scalability as it is currently not feasible to symbolically execute large programs.

The Linux kernel developers use a different approach. They extended the Smatch static analysis tool to automatically discover potential Spectre-PHT out-of-bounds access gadgets [10]. Specifically, Smatch finds all instances of user-supplied array indices that have not been explicitly hardened. Unfortunately, Smatch's false positive rate is quite high. According to Carpenter [10], the tool reported 736 gadget candidates in April 2018, whereas the kernel only featured about 15 Spectre-PHT-resistant array indices at that time. We further investigated this by analyzing the number of occurrences of the newly introduced `array_index_nospec` and `array_index_mask_nospec` macros in the Linux kernel per month. Figure 5.4 shows that the number of Spectre-PHT patches has been continuously increasing over the past year. This provides further evidence that patching Spectre-PHT gadgets in real-world software is an ongoing effort and that automated detection methods and gadget classification pose an important research challenge.

Academic Review. To date, only 5 academic papers have demonstrated Spectre-type gadget exploitation in real-world software [9, 13, 29, 50, 59]. Table 5.8 reveals that they either abuse ROP-style gadgets in larger code bases or more commonly rely on Just-In-Time (JIT) compilation to indirectly provide the vulnerable gadget code. JIT compilers as commonly used in e.g., JavaScript, WebAssembly, or the eBPF Linux kernel interface, create a software-defined sandbox by extending the untrusted attacker-provided code with runtime checks. However, the attacks in Table 5.8 demonstrate that such JIT checks can be transiently circumvented to leak memory contents outside of the sandbox. Furthermore, in the case of Spectre-BTB/RSB, even non-JIT compiled real-world code has been shown to be exploitable when the attacker controls sufficient inputs to the victim application. Kocher et al. [50] constructed a minimalist proof-of-concept that reads attacker-controlled inputs into registers before calling a function. Next, they rely on BTB poisoning to redirect transient control flow to a gadget they identified in the Windows `ntdll` library that allows leaking arbitrary memory from the victim process. Likewise, Chen et al. [13] analyzed various trusted enclave runtimes for Intel SGX and found several instances of vulnerable branches with attacker-controlled input registers, plus numerous exploitable gadgets to which transient control flow may be directed to leak unauthorized enclave memory. Bhattacharyya et al. [9] analyzed common software libraries that are likely to be linked against a victim program for gadgets. They were able to find numerous gadgets and were able to exploit one in OpenSSL to leak information.

Case Study: Linux Kernel. To further assess the prevalence of Spectre gadgets in real-world software, we selected the Linux kernel (Version 5.0) as a relevant case study of a major open-source project that underwent numerous Spectre-related security patches over the last year. We opted for an in-depth analysis of one specific piece of software instead of a breadth-first approach where we do a shallow analysis of multiple pieces of software. This allowed us to analyse historical data (*i.e.*, code locations

Table 5.8: Spectre-type attacks on real-world software.

Attack	Gadgets	JIT	Description
Spectre-PHT [50]	2	✓	Chrome Javascript, Linux eBPF
Spectre-BTB [50]	2	✓/✗	Linux eBPF, Windows `ntdll`
Spectre-BTB [13]	336	✗	SGX SDK Intel/Graphene/Rust
Spectre-BTB [9]	690	✗	OpenSSL, glibc, pthread, ...
Spectre-RSB [59]	1	✓	Firefox WebAssembly
Spectre-STL [29]	1	✓	Partial PoC on Linux eBPF

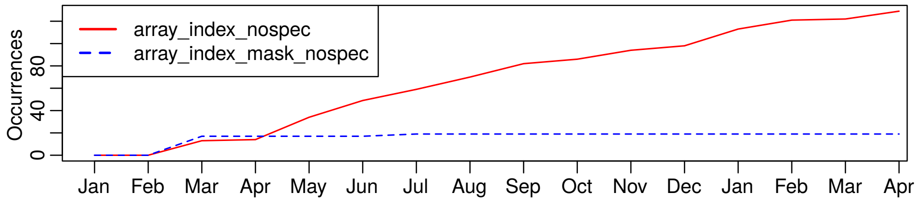


Figure 5.4: Evolution of Spectre-PHT patches in the Linux kernel over time (2018-2019).

the kernel developers deemed necessary to protect) that led to the second tier classification discussed in Section 5.1.

There are a couple of reasons that make analysis difficult. The first is that Linux supports many different platforms. Therefore, particular gadgets are only available in a specific configuration. The second point is that the number of instructions that can be transiently executed depends on the size of the ROB [89]. As we analyze high-level code, we can only estimate how far ahead the processor can transiently execute.

Table 5.7 shows the number of occurrences of each gadget type from our second tier classification. While Figure 5.4 shows around 120 occurrences of `array_index_nospec`, the number of gadgets in our analysis is higher. The reason behind that is that multiple arrays are indexed with the same masked index and that there are multiple branches on a value that was loaded with a potential malicious index. Our analysis also shows that more dangerous gadgets that either allow more than 1-bit leakage or even arbitrary code execution are not frequently occurring. Even if one is found, it might still be hard to exploit. During our analysis, we also discovered that the patch had been reverted in 13 locations, indicating that there is also some confusion among the kernel developers what needs to be fixed.

Table 5.9: Categorization of Spectre defenses and systematic overview of their microarchitectural target.

Microarchitectural Element \ Defense	InvisiSpec	SafeSpec	DAWG	Taint Tracking	Timer Reduction	RSB Stuffing	Retpoline	SLH	YSNB	IBRS	STIBP	IBPB	Serialization	Sloth	SSBD/SSBB	Poison Value	Index Masking	Site Isolation
Cache	●	●	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○
TLB	◐	●	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
BTB	○	○	○	○	○	○	●	○	○	●	●	●	○	○	○	○	○	○
BHB	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
PHT	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
RSB	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○
AVX	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
FPU	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Ports	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	C1					**C2**										**C3**		

A defense considers the microarchitectural element (●), partially considers it or same technique possible for it (◐) or does not consider it at all (○).

6 Defenses

In this section, we discuss proposed defenses in software and hardware for Spectre and Meltdown variants. We propose a classification scheme for defenses based on their attempt to stop leakage, similar to Miller [62]. Our work differs from Miller in three points. First, ours extends to newer transient execution attacks. Second, we consider Meltdown and Spectre as two problems with different root causes, leading to a different classification. Third, it helped uncover problems that were not clear with the previous classification.

We categorize Spectre-type defenses into three categories:

- **C1**: Mitigating or reducing the accuracy of covert channels used to extract the secret data.
- **C2**: Mitigating or aborting speculation if data is potentially accessible during transient execution.
- **C3**: Ensuring that secret data cannot be reached.

Table 5.9 lists proposed defenses against Spectre-type attacks and assigns them to the category they belong.

We categorize Meltdown-type defenses into two categories:

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

D2: Preventing the occurrence of faults.

6.1 Defenses for Spectre

C1: Mitigating or reducing accuracy of covert channels. Transient execution attacks use a covert channel to transfer a microarchitectural state change induced by the transient instruction sequence to the architectural level. One approach in mitigating Spectre-type attacks is reducing the accuracy of covert channels or preventing them.

Hardware. One enabler of transient execution attacks is that the transient execution sequence introduces a microarchitectural state change the receiving end of the covert channel observes. To secure CPUs, SafeSpec [45] introduces shadow hardware structures used during transient execution. Thereby, any microarchitectural state change can be squashed if the prediction of the CPU was incorrect. While their prototype implementation protects only caches (and the TLB), other channels, e.g., DRAM buffers [69], or execution unit congestion [1, 9, 56], remain open.

Yan et al. [91] proposed InvisiSpec, a method to make transient loads invisible in the cache hierarchy. By using a *speculative buffer*, all transiently executed loads are stored in this buffer instead of the cache. Similar to SafeSpec, the buffer is invalidated if the prediction was incorrect. However, if the prediction was correct, the content of the buffer is loaded into the cache. For data coherency, InvisiSpec compares the loaded value during this process with the most recent, up-to-date value from the cache. If a mismatch occurs, the transient load and all successive instructions are reverted. Since InvisSpec only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels.

Kiriansky et al. [47] securely partition the cache across its ways. With protection domains that isolate on a cache hit, cache miss and metadata level, cache-based covert channels are mitigated. This does not only require changes to the cache and adaptions to the coherence protocol but also enforces the correct management of these domains in software.

Kocher et al. [50] proposed to limit data from entering covert channels through a variation of taint tracking. The idea is that the CPU tracks data loaded during transient execution and prevents their use in subsequent operations.

Software. Many covert channels require an accurate timer to distinguish microarchitectural states, e.g., measuring the memory access latency to distinguish between a cache hit and cache miss. With reduced timer accuracy an attacker cannot distinguish between microarchitectural states any longer, the receiver of the covert channel cannot deduce the sent information. To mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript by adding jitter [61, 70, 80, 88]. However, Schwarz et al. [73] demonstrated that timers can be constructed in many different ways and, thus, further mitigations are required [71]. While Chrome initially disabled `SharedArrayBuffers` in response to Meltdown and Spectre [80], this timer source has been re-enabled with the introduction of site-isolation [77].

NetSpectre requires different strategies due to its remote nature. Schwarz et al. [74] propose to detect the attack using DDoS detection mechanisms or adding noise to the network latency. By adding noise, an attacker needs to record more traces. Adding enough noise makes the attack infeasible in practice as the amount of traces as well as the time required for averaging it out becomes too large [87].

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution. Since Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [50, 79]. As the loss of performance for commodity computers and servers would be too drastic, another proposal is to disable speculation only while processing secret data.

Hardware. A building blocks for some variants of Spectre is branch poisoning (an attacker mistrains a prediction mechanism, cf. Section 3). To deal with mistraining, both Intel and AMD extended the instruction set architecture (ISA) with a mechanism for controlling indirect branches [4, 40]. The proposed addition to the ISA consists of three controls:

- Indirect Branch Restricted Speculation (IBRS) prevents indirect branches executed in privileged code from being influenced by those in less privileged code. To enforce this, the CPU enters the IBRS mode which cannot be influenced by any operations outside of it.
- Single Thread Indirect Branch Prediction (STIBP) restricts sharing of branch prediction mechanisms among code executing across hyperthreads.
- The Indirect Branch Predictor Barrier (IBPB) prevents code that executes before it from affecting the prediction of code following it by flushing the BTB.

For existing ARM implementations, there are no generic mitigation techniques available. However, some CPUs implement specific controls that allow invalidating the branch predictor which should be used during context switches [6]. On Linux, those mechanisms are enabled by default [46]. With the ARMv8.5-A instruction set [7], ARM introduces a new barrier (`sb`) to limit speculative execution on following instructions. Furthermore, new system registers allow to restrict speculative execution and new prediction control instructions prevent control flow predictions (`cfp`), data value prediction (`dvp`) or cache prefetch prediction (`cpp`) [7].

To mitigate Spectre-STL, ARM introduced a new barrier called `SSBB` that prevents a load following the barrier from bypassing a store using the same virtual address before it [6]. For upcoming CPUs, ARM introduced Speculative Store Bypass Safe (SSBS); a configuration control register to prevent the re-ordering of loads and stores [6]. Likewise, Intel [40] and AMD [3] provide Speculative Store Bypass Disable (SSBD) microcode updates that mitigate Spectre-STL.

As an academic contribution, plausible hardware mitigations have furthermore been proposed [48] to prevent transient computations on out-of-bounds writes (Spectre-PHT).

Software. Intel and AMD proposed to use serializing instructions like `lfence` on both outcomes of a branch [4, 31]. ARM introduced a full data synchronization barrier (`DSB SY`) and an instruction synchronization barrier (ISB) that can be used to prevent speculation [6]. Unfortunately, serializing every branch would amount to completely disabling branch prediction, severely reducing performance [31]. Hence, Intel further proposed to use static analysis [31] to minimize the number of serializing instructions introduced. Microsoft uses the static analyzer of their C Compiler MSVC [68] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [66] use a similar approach using static analysis. Kocher [49] showed that this approach misses many gadgets that can be exploited.

Serializing instructions can also reduce the effect of indirect branch poisoning. By inserting it before the branch, the pipeline prior to it is cleared, and the branch is resolved quickly [4]. This, in turn, reduces the size of the speculation window in case that misspeculation occurs.

While `lfence` instructions stop speculative execution, Schwarz et al. [74] showed they do not stop microarchitectural behaviors happening before execution. This, for instance, includes powering up the AVX functional units, instruction cache fills, and iTLB fills which still leak data.

Evtyushkin et al. [18] propose a similar method to serializing instructions, where a developer annotates potentially leaking branches. When indicated, the CPU should not predict the outcome of these branches and thus stop speculation.

Additionally to the serializing instructions, ARM also introduced a new barrier (`CSDB`) that in combination with conditional selects or moves controls speculative execution [6].

Speculative Load Hardening (SLH) is an approach used by LLVM and was proposed by Carruth [12]. Using this idea, loads are checked using branchless code to ensure that they are executing along a valid control flow path. To do this, they transform the code at the compiler level and introduce a data dependency on the condition. In the case of misspeculation, the pointer is zeroed out, preventing it from leaking data through speculative execution. One prerequisite for this approach is hardware that allows the implementation of a branchless and unpredicted conditional update of a register's value. As of now, the feature is only available in LLVM for x86 as the patch for ARM is still under review. GCC adopted the idea of SLH for their implementation, supporting both x86 and ARM. They provide a builtin function to either emit a speculation barrier or return a safe value if it determines that the instruction is transient [17].

Oleksenko et al. [65] propose an approach similar to Carruth [12]. They exploit that CPUs have a mechanism to detect data dependencies between instructions and introduce such a dependency on the comparison arguments. This ensures that the load only starts when the comparison is either in registers or the L1 cache, reducing the speculation window to a non-exploitable size. They already note that their approach is highly dependent on the ordering of instructions as the CPU might perform the load before the comparison. In that case, the attack would still be possible.

Google proposes a method called *retpoline* [83], a code sequence that replaces indirect branches with return instructions, to prevent branch poisoning. This method ensures that return instructions always speculate into an endless loop through the RSB. The actual target destination is pushed on the stack and returned to using the `ret` instruction. For retpoline, Intel [39] notes that in future CPUs that have Control-flow Enforcement Technology (CET) capabilities to defend against ROP attacks, retpoline might trigger false positives in the CET defenses. To mitigate this possibility, future CPUs also implement hardware defenses for Spectre-BTB called *enhanced IBRS* [39].

On Skylake and newer architectures, Intel [39] proposes RSB stuffing to prevent an RSB underfill and the ensuing fallback to the BTB. Hence, on

every context switch into the kernel, the RSB is filled with the address of a benign gadget. This behavior is similar to retpoline. For Broadwell and older architectures, Intel [39] provided a microcode update to make the `ret` instruction predictable, enabling retpoline to be a robust defense against Spectre-BTB. Windows has also enabled retpoline on their systems [14].
C3: Ensuring that secret data cannot be reached. Different projects use different techniques to mitigate the problem of Spectre. WebKit employs two such techniques to limit the access to secret data [70]. WebKit first replaces array bound checks with index masking. By applying a bit mask, WebKit cannot ensure that the access is always in bounds, but introduces a maximum range for the out-of-bounds violation. In the second strategy, WebKit uses a pseudo-random *poison value* to protect pointers from misuse. Using this approach, an attacker would first have to learn the poison value before he can use it. The more significant impact of this approach is that mispredictions on the branch instruction used for type checks results in the wrong type being used for the pointer.

Google proposes another defense called *site isolation* [81], which is now enabled in Chrome by default. Site isolation executes each site in its own process and therefore limits the amount of data that is exposed to side-channel attacks. Even in the case where the attacker has arbitrary memory reads, he can only read data from its own process.

Kiriansky and Waldspurger [48] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [35]. They note that by using Spectre-PHT an attacker can first disable the protection before reading the data. To prevent this, they propose to include an `lfence` instruction in `wrpkru`, an instruction used to modify protection keys.

6.2 Defenses for Meltdown

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

The fundamental problem of Meltdown-type attacks is that the CPU allows the transient instruction stream to compute on architecturally inaccessible values, and hence, leak them. By assuring that execution does not continue with unauthorized data after a fault, such attacks can be mitigated directly in silicon. This design is enforced in AMD processors [4], and more recently also in Intel processors from Whiskey Lake onwards that enumerate RDCL_NO support [40]. However, mitigations for existing microarchitectures are necessary, either through microcode updates, or operating-system-level software workarounds. These approaches aim to keep architecturally inaccessible data also inaccessible at the microarchitectural level.

Gruss et al. originally proposed KAISER [22, 23] to mitigate side-channel attacks defeating KASLR. However, it also defends against Meltdown-US attacks by preventing kernel secrets from being mapped in user space. Besides its performance impact, KAISER has one practical limitation [22, 56]. For x86, some privileged memory locations must always be mapped in user space. KAISER is implemented in Linux as kernel page-table isolation (KPTI) [58] and has also been backported to older versions. Microsoft provides a similar patch as of Windows 10 Build 17035 [42] and Mac OS X and iOS have similar patches [41].

For Meltdown-GP, where the attacker leaks the contents of system registers that are architecturally not accessible in its current privilege level, Intel released microcode updates [31]. While AMD is not susceptible [5], ARM incorporated mitigations in future CPU designs and suggests to substitute the register values with dummy values on context switches for CPUs where mitigations are not available [6].

Preventing the access-control race condition exploited by Foreshadow and Meltdown may not be feasible with microcode updates [86]. Thus, Intel proposes a multi-stage approach to mitigate Foreshadow (L1TF) attacks on current CPUs [30, 90]. First, to maintain process isolation, the operating system has to sanitize the physical address field of unmapped page-table entries. The kernel either clears the physical address field, or sets it to non-existent physical memory. In the case of the former, Intel suggests placing 4 KB of dummy data at the start of the physical address space, and clearing the PS bit in page tables to prevent attackers from exploiting huge pages.

For SGX enclaves or hypervisors, which cannot trust the address translation performed by an untrusted OS, Intel proposes to either store secrets in uncacheable memory (as specified in the PAT or the MTRRs), or flush the L1 data cache when switching protection domains. With recent microcode updates, L1 is automatically flushed upon enclave exit, and hypervisors can additionally flush L1 before handing over control to an untrusted virtual machine. Flushing the cache is also done upon exiting System Management Mode (SMM) to mitigate Foreshadow-NG attacks on SMM.

To mitigate attacks across logical cores, Intel supplied a microcode update to ensure that different SGX attestation keys are derived when hyperthreading is enabled or disabled. To ensure that no non-SMM software runs while data belonging to SMM are in the L1 data cache, SMM software must rendezvous all logical cores upon entry and exit. According to Intel, this is expected to be the default behavior for most SMM software [30]. To protect against Foreshadow-NG attacks when hyperthreading is enabled, the hypervisor must ensure that no hypervisor thread runs on a sibling core with an untrusted VM.

D2: Preventing the occurrence of faults. Since Meltdown-type attacks exploit delayed exception handling in the CPU, another mitigation approach is to prevent the occurrence of a fault in the first place. Thus, accesses which would normally fault, become (both architecturally and microarchitecturally) valid accesses but do not leak secret data.

One example of such behavior are SGX's abort page semantics, where accessing enclave memory from the outside returns -1 instead of faulting. Thus, SGX has inadvertent protection against Meltdown-US. However, the Foreshadow [86] attack showed that it is possible to actively provoke another fault by unmapping the enclave page, making SGX enclaves susceptible to the Meltdown-P variant.

Preventing the fault is also the countermeasure for Meltdown-NM [78] that is deployed since Linux 4.6 [57]. By replacing lazy switching with eager switching, the FPU is always available, and access to the FPU can never fault. Here, the countermeasure is effective, as there is no other way to provoke a fault when accessing the FPU.

6.3 Evaluation of Defenses

Spectre Defenses. We evaluate defenses based on their capabilities of mitigating Spectre attacks. Defenses that require hardware modifications are only evaluated theoretically. In addition, we discuss which vendors have CPUs vulnerable to what type of Spectre- and Meltdown-type attack. The results of our evaluation are shown in Table 5.10.

Several defenses only consider a specific covert channel (see Table 5.9), *i.e.*, they only try to prevent an attacker from recovering the data using a specific covert channel instead of targeting the root cause of the vulnerability. Therefore, they can be subverted by using a different one. As such, they can not be considered a reliable defense. Other defenses only limit the amount of data that can be leaked [70, 81] or simply require more repetitions on the attacker side [74, 87]. Therefore, they are only partial solutions. RSB stuffing only protects a cross-process attack but does not mitigate a same-process attack. Many of the defenses are not enabled by default or depend on the underlying hardware and operating system [3, 4, 6, 40]. With serializing instructions [4, 6, 31] after a bounds check, we were still able to leak data on Intel and ARM (only with `DSB SY+ISH` instruction) through a single memory access and the TLB. On ARM, we observed no leakage following a `CSDB` barrier in combination with conditional selects or moves. We also observed no leakage with SLH, although the possibility remains that our experiment failed to bypass the mitigation. Taint tracking theoretically mitigates all forms of Spectre-type attacks as data that has been tainted cannot be used in a transient execution. Therefore, the data does not enter a covert channel and can subsequently not be leaked.

Meltdown Defenses. We verified whether we can still execute Meltdown-type attacks on a fully-patched system. On a Ryzen Threadripper 1920X, we were still able to execute Meltdown-BND. On an i5-6200U (Skylake), an i7-8700K (Coffee Lake), and an i7-8565U (Whiskey Lake), we were able to successfully run a Meltdown-MPX, Meltdown-BND, and Meltdown-RW attack. Additionally to those, we were also able to run a Meltdown-PK attack on an Amazon EC2 C5 instance (Skylake-SP). Our results indicate that current mitigations only prevent Meltdown-type attacks that cross the current privilege level. We also tested whether we can still successfully execute a Meltdown-US attack on a recent Intel Whiskey Lake CPU without KPTI enabled, as Intel claims these processors are no longer vulnerable. In our experiments, we were indeed not able to leak any data on such CPUs but encourage other researchers to further investigate newer processor generations.

6.4 Performance Impact of Countermeasures

There have been several reports on performance impacts of selected countermeasures. Some report the performance impact based on real-world scenarios (top of Table 5.11) while others use a specific benchmark that might not resemble real-world usage (lower part of Table 5.11). Based on the different testing scenarios, the results are hard to compare. To further complicate matters, some countermeasures require hardware modifications that are not available, and it is therefore hard to verify the performance loss.

One countermeasure that stands out with a huge decrease in performance is serialization and highlights the importance of speculative execution to improve CPU performance. Another interesting countermeasure is KPTI. While it was initially reported to have a huge impact on performance, recent work shows that the decrease is almost negligible

Table 5.10: Spectre defenses and which attacks they mitigate.

Attack	Defense	`InvisiSpec`	`SafeSpec`	`DAWG`	*RSB Stuffing*	*Retpoline*	*Poison Value*	*Index Masking*	*Site Isolation*	*SLH*	`YSNB`	*IBRS*	*STIBP*	*IBPB*	*Serialization*	`Taint Tracking`	*Timer Reduction*	`Sloth`	*SSBD/SSBB*
Intel	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	●	◐	◐	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
ARM	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
AMD	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	■	◧	◧	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◧	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇). Defenses in *italics* are production-ready, while `typeset` defenses are academic proposals.

on systems that support PCID [20]. To mitigate Spectre and Meltdown, current systems rely on a combination of countermeasures. To show the overall decrease on a Linux 4.19 kernel with the default mitigations enabled, Larabel [54] performed multiple benchmarks to determine the impact. On Intel, the slowdown was 7-16% compared to a non-mitigated kernel, on AMD it was 3-4%.

Naturally, the question arises which countermeasures to enable. For most users, the risk of exploitation is low, and default software mitigations as provided by Linux, Microsoft, or Apple likely are sufficient. This is likely the optimum between potential attacks and reasonable performance. For data centers, it is harder as it depends on the needs of their customers and one has to evaluate this on an individual basis.

Table 5.11: Reported performance impacts of countermeasures. Top shows performance impact in real-world scenarios while the bottom shows it on a specific benchmark.

Defense Evaluation	**Penalty**	**Benchmark**
KAISER/KPTI [21]	0–2.6 %	System call rates
Retpoline [11]	5–10 %	Real-world workload servers
Site Isolation [81]	10–13 %	Memory overhead
InvisiSpec [91]	22 %	SPEC
SafeSpec [45]	-3 %	SPEC on MARSSx86
DAWG [47]	1–15 %	PARSEC , GAPBS
SLH [12]	29–36.4 %	Google microbenchmark suite
YSNB [65]	60 %	Phoenix
IBRS [82]	20–30 %	Sysbench 1.0.11
STIBP [53]	30–50 %	Rodinia OpenMP, DaCapo
Serialization [12]	62–74.8 %	Google microbenchmark suite
SSBD/SSBB [15]	2–8 %	SYSmark 2018, SPEC integer
L1TF Mitigations [38]	-3–31 %	SPEC

7 Future Work and Conclusion

Future Work. For Meltdown-type attacks, it is important to determine where data is actually leaked from. For instance, Lipp et al. [56] demonstrated that Meltdown-US can not only leak data from the L1 data cache and main memory but even from memory locations that are explicitly marked as "uncacheable" and are hence served from the Line Fill Buffer (LFB). [1] In future work, other Meltdown-type attacks should be tested to determine whether they can also leak data from different microarchitectural buffers. In this paper, we presented a small evaluation of the prevalence of gadgets in real-world software. Future work should develop methods for automating the detection of gadgets and extend the analysis on a larger amount of real-world software. We have also discussed

[1]The initial Meltdown-US disclosure (December 2017) and subsequent paper [56] already made clear that Meltdown-type leakage is *not* limited to the L1 data cache. We sent Intel a PoC leaking uncacheable-typed memory locations from a concurrent hyperthread on March 28, 2018. We clarified to Intel on May 30, 2018, that we attribute the source of this leakage to the LFB. In our experiments, this works identically for Meltdown-P (Foreshadow). This issue was acknowledged by Intel, tracked under CVE-2019-11091, and remained under embargo until May 14, 2019.

mitigations and shown that some of them can be bypassed or do not target the root cause of the problem. We encourage both offensive and defensive research that may use our taxonomy as a guiding principle to discover new attack variants and develop mitigations that target the root cause of transient information leakage.

Conclusion. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. We presented a systematization of transient execution attacks. Our systematization uncovered 6 (new) transient execution attacks (Spectre and Meltdown variants) which have been overlooked and have not been investigated so far. We demonstrated these variants in practical proof-of-concept attacks and evaluated their applicability to Intel, AMD, and ARM CPUs. We also presented a short analysis and classification of gadgets as well as their prevalence in real-world software. We also systematically evaluated defenses, discovering that some transient execution attacks are not successfully mitigated by the rolled out patches and others are not mitigated because they have been overlooked. Hence, we need to think about future defenses carefully and plan to mitigate attacks and variants that are yet unknown.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Jonathan McCune, for their helpful comments and suggestions that substantially helped in improving the paper.

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This research received funding from the Research Fund KU Leuven, and Jo Van Bulck is supported by the Research Foundation – Flanders (FWO). Evtyushkin acknowledges the start-up grant from the College of William and Mary. Additional funding was provided by generous gifts from ARM and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. *Port Contention for Fun and Profit.* 2018.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit." In: *ePrint 2018/1060* (2018).

[3] AMD. *AMD64 Technology: Speculative Store Bypass Disable.* Revision 5.21.18. 2018.

[4] AMD. *Software Techniques for Managing Speculation on AMD Processors.* Revison 7.10.18. 2018.

[5] AMD. *Spectre Mitigation Update.* July 2018.

[6] ARM. *Cache Speculation Side-channels.* Version 2.4. 2018.

[7] ARM Limited. *ARM A64 Instruction Set Architecture.* 2018.

[8] ARM Limited. *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism.* 2018.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention." In: *arXiv:1903.01843* (2019).

[10] Dan Carpenter. *Smatch check for Spectre stuff.* Apr. 2018.

[11] Chandler Carruth. Jan. 2018. URL: `https://reviews.llvm.org/D41723`.

[12] Chandler Carruth. *RFC: Speculative Load Hardening (a Spectre variant #1 mitigation).* Mar. 2018.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. "SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution." In: *arXiv:1802.09085* (2018).

[14] Microsoft Corp. Mar. 2019. URL: `https://support.microsoft.com/en-us/help/4482887/windows-10-update-kb4482887`.

[15] Leslie Culbertson. "Addressing New Research for Side-Channel Analysis." In: Intel. May 2018.

[16] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. "Spectres, virtual ghosts, and hardware support." In: *Workshop on Hardware and Architectural Support for Security and Privacy*. 2018.

[17] Richard Earnshaw. *Mitigation against unsafe data speculation (CVE-2017-5753)*. July 2018.

[18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor." In: *ASPLOS'18*. 2018.

[19] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2016.

[20] Brendan Gregg. *KPTI/KAISER Meltdown Initial Performance Regressions*. 2018.

[21] Daniel Gruss, Dave Hansen, and Brendan Gregg. "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer." In: *USENIX ;login* (2018).

[22] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR." In: *ESSoS*. 2017.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *CCS*. 2016.

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[25] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled Detection of Speculative Information Flows." In: *arXiv:1812.08639* (2018).

[26] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES." In: *Constructive Side-Channel Analysis and Secure Design*. 2015.

[27] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. *Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries*. 2018.

[28] Jann Horn. *Reading privileged memory with a side-channel.* Jan. 2018.

[29] Jann Horn. *speculative execution, variant 4: speculative store bypass.* 2018.

[30] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault.* Aug. 2018.

[31] Intel. *Intel Analysis of Speculative Execution Side Channels.* Revision 4.0. July 2018.

[32] Intel. *Intel Software Guard Extensions (Intel SGX).* 2016.

[33] Intel. *Intel Xeon Processor Scalable Family Technical Overview.* Sept. 2017.

[34] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* 2017.

[35] Intel. "Intel 64 and IA-32 Architectures Software Developer′s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide." In: 325384 (2016).

[36] Intel. *More Information on Transient Execution Findings.* 2018. URL: https://software.intel.com/security-software-guidance/insights/more-information-transient-execution-findings.

[37] Intel. *Q2 2018 Speculative Execution Side Channel Update.* May 2018.

[38] Intel. *Resources and Response to Side Channel L1 Terminal Fault.* Aug. 2018.

[39] Intel. *Retpoline: A Branch Target Injection Mitigation.* Revision 003. June 2018.

[40] Intel. *Speculative Execution Side Channel Mitigations.* Revision 3.0. May 2018.

[41] Alex Ionescu. *Twitter: Apple Double Map.* 2017. URL: https://twitter.com/aionescu/status/948609809540046849.

[42] Alex Ionescu. *Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER).* 2017. URL: https://twitter.com/aionescu/status/930412525111296000.

[43] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES." In: *RAID'14.* 2014.

[44] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks." In: *arXiv:1903.00446* (2019).

[45] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation." In: *arXiv:1806.05179* (2018).

[46] Russel King. *ARM: spectre-v2: harden branch predictor on context switches.* 2018.

[47] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors." In: *ePrint 2018/418* (May 2018).

[48] Vladimir Kiriansky and Carl Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses." In: *arXiv:1807.03757* (2018).

[49] Paul Kocher. *Spectre Mitigations in Microsoft's C/C++ Compiler.* 2018.

[50] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *S&P.* 2019.

[51] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems." In: *CRYPTO.* 1996.

[52] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer." In: *WOOT.* 2018.

[53] Michael Larabel. *Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower.* Nov. 2018.

[54] Michael Larabel. *The Performance Cost Of Spectre / Meltdown / Foreshadow Mitigations On Linux 4.19.* Aug. 2018.

[55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium.* 2016.

[56] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium.* 2018.

[57] Andy Lutomirski. *x86/fpu: Hard-disable lazy FPU mode.* June 2018.

[58] LWN. *The current state of kernel page-table isolation.* Dec. 2017. URL: `https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/`.

[59] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers.” In: *CCS*. 2018.

[60] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.” In: *NDSS*. 2017.

[61] Microsoft. *Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer*. Jan. 2018.

[62] Matt Miller. *Mitigating speculative execution side channel hardware vulnerabilities*. Mar. 2018.

[63] Dan O’Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. *Spectre attack against SGX enclave*. Jan. 2018.

[64] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. “Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches.” In: *arXiv:1702.00719* (2017).

[65] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. “You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass.” In: *arXiv:1805.08506* (2018).

[66] Open Source Security Inc. *Respectre: The State of the Art in Spectre Defenses*. Oct. 2018.

[67] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: the Case of AES.” In: *CT-RSA*. 2006.

[68] Andrew Pardoe. *Spectre mitigations in MSVC*. 2018.

[69] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.” In: *USENIX Security Symposium*. 2016.

[70] Filip Pizlo. *What Spectre and Meltdown Mean For WebKit*. Jan. 2018.

[71] Michael Schwarz, Moritz Lipp, and Daniel Gruss. “JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks.” In: *NDSS*. 2018.

[72] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In: *NDSS*. 2018.

[73] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC*. 2017.

[74] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *arXiv:1807.10535* (2018).

[75] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In: *CCS*. 2007.

[76] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating controlled-channel attacks against enclave programs." In: *NDSS*. 2017.

[77] Ben Smith. *Enable SharedArrayBuffer by default on non-android*. Aug. 2018.

[78] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels." In: *arXiv:1806.07480* (2018).

[79] SUSE. *Security update for kernel-firmware*. 2018. URL: `https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1/`.

[80] The Chromium Projects. *Actions required to mitigate Speculative Side-Channel Attack techniques*. 2018.

[81] The Chromium Projects. *Site Isolation*. 2018.

[82] Vadim Tkachenko. *20-30% Performance Hit from the Spectre Bug Fix on Ubuntu*. Jan. 2018.

[83] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018.

[84] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. "ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys." In: *arXiv:1801.06822* (2018).

[85] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic." In: *CCS*. 2018.

[86] Jo Van Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.” In: *USENIX Security Symposium*. 2018.

[87] Kenton Varda. *WebAssembly’s post-MVP future*. 2018. URL: `https://news.ycombinator.com/item?id=18279791`.

[88] Luke Wagner. *Mitigations landing for new class of timing attack*. Jan. 2018.

[89] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. “oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis.” In: *arXiv:1807.05843* (2018).

[90] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. 2018.

[91] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy.” In: *MICRO*. 2018.

[92] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.” In: *USENIX Security Symposium*. 2014.

6
KASLR: Break It, Fix It, Repeat

Publication Data

Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat.” In: *AsiaCCS*. 2020

Contributions

Main author.

KASLR: Break It, Fix It, Repeat

Claudio Canella, Michael Schwarz, Martin Haubenwallner
Martin Schwarzl, Daniel Gruss

Graz University of Technology, Austria

Abstract

In this paper, we analyze the hardware-based Meltdown mitigations in recent Intel microarchitectures, revealing that illegally accessed data is only zeroed out. Hence, while non-present loads stall the CPU, illegal loads are still executed. We present EchoLoad, a novel technique to distinguish load stalls from transiently executed loads. EchoLoad allows detecting physically-backed addresses from unprivileged applications, breaking KASLR in 40 µs on the newest Meltdown- and MDS-resistant Cascade Lake microarchitecture. As EchoLoad only relies on memory loads, it runs in highly-restricted environments, e.g., SGX or JavaScript, making it the first JavaScript-based KASLR break. Based on EchoLoad, we demonstrate the first proof-of-concept Meltdown attack from JavaScript on systems that are still broadly not patched against Meltdown, *i.e.*, 32-bit x86 OSs.

We propose FLARE, a generic mitigation against known microarchitectural KASLR breaks with negligible overhead. By mapping unused kernel addresses to a reserved page and mirroring neighboring permission bits, we make used and unused kernel memory indistinguishable, *i.e.*, a uniform behavior across the entire kernel address space, mitigating the root cause behind microarchitectural KASLR breaks. With incomplete hardware mitigations, we propose to deploy FLARE even on recent CPUs.

1 Introduction

CPUs are optimized for performance and efficiency. Some optimizations are exposed to the user via the instruction-set architecture (ISA), the hardware-software interface, but most are transparent to the developer. CPU vendors can implement any optimizations while still adhering to the ISA, without taking security into account.

The original publication is available at `https://dl.acm.org/doi/abs/10.1145/3320269.3384747`.

As a consequence, many attacks on the microarchitectural CPU state have been published [24]. Most of these are side-channel attacks, including attacks on cryptographic algorithms [4, 45, 46, 60, 69, 71, 96], on user interactions [33, 58, 79], but also covert data transmission [60, 62, 94, 95]. Meltdown [59], Foreshadow [88, 93], RIDL [74], ZombieLoad [80], and Fallout [10] are recent microarchitectural attacks that go beyond side-channel attacks and directly leak (arbitrary) data instead of metadata. These attacks, commonly referred to as Meltdown-type transient-execution attacks [9], exploit the lazy fault handling property of some CPUs. With lazy fault handling, the CPU continues using data in the out-of-order execution even if the loading of the data resulted in a fault, e.g., failed the privilege-level check. While the data never becomes visible on the architectural level, it is encoded in the microarchitectural state, *i.e.*, in the cache. From there, it is made visible on the architectural level using microarchitectural side-channel attacks.

Meltdown-type transient-execution attacks [9] break the hardware enforced-isolation between the trusted kernel and untrusted user programs. These attacks, which are present in most Intel CPUs, showed that it is not always possible to protect an application against side-channel attacks. This is contrary to the belief that side-channel attacks have to be prevented by the application itself [5, 46].

Deeply rooted in the CPU, either close to or in the critical path, most transient-execution attacks cannot be fixed with microcode updates. However, on recent CPUs, Intel introduced hardware mitigations for the first Meltdown-type attacks [17, 40, 42]. On the latest microarchitecture (Cascade Lake), all known Meltdown-type attacks are mitigated in hardware [37]. However, due to their severity and the ease of exploitation, operating systems (OSs) rolled out software-based mitigations to prevent Meltdown [29]. These software mitigations introduce a stricter separation of user and kernel space [30, 32]. This stricter separation does not only prevent Meltdown but also prevents other microarchitectural attacks on the kernel [30], e.g., KASLR (kernel address space layout randomization) breaks [32, 36, 49] which allow an attacker to de-randomize the location of the kernel in the address space. As a drawback, these software mitigations may introduce significant performance overhead. This is especially true for workloads that need frequent switching between user and kernel space [29]. Consequently, hardware manufacturers solved the underlying root cause directly in recent CPUs, making the software mitigations obsolete.

Even though new CPUs are not vulnerable anymore to the original Meltdown attack, we show that they still show signs of Meltdown-type effects. In this paper, we investigate CPUs that have hardware-based Meltdown mitigations. We analyze these fixes and develop the hypothesis that they only prevent the data from being used in subsequent operations, not the actual load. We confirm this hypothesis by showing that the fixes introduce new side effects, namely that on illegal accesses to kernel addresses, the CPU zeroes out the data but still performs the load. In contrast, loads from non-present pages stall the CPU. We present a method based on Flush+Reload [96] to distinguish the stalling behavior of loads. With this method, we can exploit the side effects of the Meltdown mitigations to break KASLR reliably. By probing the kernel space for load stalls, we detect whether the probed virtual address is physically backed, revealing the location of the kernel. We demonstrate that these effects can also be exploited on older CPUs, which are affected by Meltdown but protected by software mitigations.

Our KASLR break, EchoLoad, works on all major OSs (Linux, Windows, macOS, and Android x86_64). We tested the KASLR break on Intel microarchitectures from Arrandale (2010) to Cascade Lake (2019) on Atom, Core, and Xeon CPUs. Even on Cascade Lake with fixes for Meltdown and MDS [37], we de-randomize the kernel in 40 µs (F-score 1, $n = 10^9$). Hence, our KASLR break is the fastest and most reliable one published. Moreover, EchoLoad is the only KASLR break that only relies on memory loads and works on Intel microarchitectures since at least 2010. EchoLoad even works on KPTI, the Linux software mitigation for Meltdown.

As EchoLoad does not require anything but memory loads, it works in restricted environments such as SGX and JavaScript. We highlight that EchoLoad can aid kernel exploitation from within SGX enclaves, facilitating SGX malware [82, 83]. In contrast to previous ASLR breaks from JavaScript [7, 28, 77], we are the first to demonstrate a microarchitectural KASLR break from JavaScript on x86 OSs. We also show that on older unpatched x86 OSs, Meltdown can even be exploited from JavaScript. This is particularly dangerous for any Windows XP machines (1–3% of Desktop computers [68]), for which no software patches are available, but which are still running in official, commercial, industrial, or personal environments. Our attack will also soon be possible on unprotected 64-bit systems as WebAssembly plans to extend the size of linear memory indices to 64 b [91]. We pinpoint the remaining challenges for widely deployable JavaScript-based Meltdown exploits.

To mitigate all microarchitectural attacks on KASLR, including EchoLoad, we present FLARE (Fake Load Address REsponse). The basic idea is to back the entire kernel address space with physical pages. FLARE prevents previous attacks [10, 32, 36, 49, 77] by hiding the kernel within a virtual-address range appearing to be valid. Our proof-of-concept implementation has a memory overhead of only 12 kB, and no measurable runtime overhead.

FLARE is compatible with KPTI on Meltdown-affected CPUs, and forms a low-cost mitigation on CPUs with hardware fixes. We evaluated our open-source proof-of-concept implementation of FLARE[1] for Linux for both cases. Our results show that FLARE indeed prevents all known microarchitectural attacks on KASLR.

We conclude that while the hardware mitigation for Meltdown fixes the problem of Meltdown-US [9], they introduce a new side effect by merely zeroing out data that is illegally accessed, enabling EchoLoad. Based on our analysis of the behavior of AMD and ARM CPUs, we believe that the only complete solution for Meltdown is to treat inaccessible pages the same way as unmapped pages. Furthermore, the software-based isolation of user and kernel space [30] is not sufficient, and we thus suggest to deploy FLARE to prevent microarchitectural attacks on the kernel.

Contributions. The contributions of this work are:

1. We analyze Meltdown hardware fixes on Intel CPUs and discover a Meltdown-related effect on Meltdown-fixed Intel CPUs.
2. We present KASLR and ASLR breaks, even from SGX and including the first KASLR break from JavaScript.
3. We show a JavaScript Meltdown attack on 32-bit x86.
4. We propose FLARE, a mitigation preventing currently known microarchitectural attacks on KASLR with negligible overhead.

Outline. Section 2 provides background on ASLR and attacks on ASLR. We analyze Meltdown hardware fixes in Section 3. Section 4 presents our new mitigation for microarchitectural KASLR breaks. Section 5 evaluates FLARE's performance and efficacy against attacks. Section 6 discusses related work. Section 7 concludes.

Responsible Disclosure. We responsibly disclosed our findings to Intel on August 5, 2019, and Intel acknowledged them.

[1] `https://github.com/IAIK/FLARE`

2 Background

In this section, we provide the background on caches, transient execution and transient-execution attacks, virtual memory, Intel SGX, and address space layout randomization (ASLR).

2.1 Cache Attacks

Caches were designed to hide the latency of memory accesses, creating a timing side channel. Over the past two decades, many different attack techniques have been proposed [5, 31, 54, 69, 96]. Two of these attacks are Prime+Probe [69, 71] and Flush+Reload [96]. In a Prime+Probe attack, an attacker constantly measures how long it takes to fill a cache set with the same set of data. Whenever a victim accesses a cache line mapping to the same cache set, the attacker will measure a higher runtime for the filling. In a Flush+Reload attack, an attacker constantly flushes a cache line and reloads the data. By measuring how long the reload takes, the attacker can infer whether a victim has accessed the data in the meantime. As Flush+Reload exhibits low noise and has a fine granularity, it has been used for attacks on user input [33, 58, 79], cryptographic algorithms [4, 46, 96], and web server function calls [97].

Side channels can also be used to build covert channels. In a covert channel, the attacker controls both the sender and receiver. The goal is then to leak information from one security domain to another, bypassing isolation on both the functional and system level. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [31, 60, 62].

2.2 Transient-execution Attacks

Another optimization is out-of-order execution, avoiding CPU stalls when in-order instructions wait for operands. Instructions are decoded into micro-operations (μOPs) [21] and placed in the Re-Order Buffer (ROB), along with their operands. While waiting for operands, μOPs whose operands are already available are scheduled in the meantime. Results of the out-of-order executed instructions are stored until they can be retired.

Modern software is rarely linear but contains branches. To avoid pipeline stalls upon unresolved branch conditions, modern CPUs implement speculative execution, predicting the most likely outcome of the branch and starting execution along the predicted path. The results are again placed in the ROB until retirement, *i.e.*, the prediction has been verified. If the prediction was correct, a significant speedup is achieved. Otherwise, the CPU has to revert all results and needs to flush the pipeline and the ROB. Unfortunately, microarchitectural state changes, such as loading data into the cache or TLB, are not reverted. This allows an attacker to use microarchitectural covert channels to exfiltrate the secret data. Speculative or out-of-order executed instructions that were never committed to the architectural state are also referred to as *transient instructions* [9, 53, 59]. Spectre-type attacks exploit transient execution before a misprediction is discovered [9, 35, 51, 53, 55, 61]. Meltdown-type attacks exploit transient execution before a fault or interrupt is handled [3, 9, 10, 40, 42, 51, 59, 74, 80, 86, 88, 93].

Meltdown. Meltdown exploited lazy exception handling in modern CPUs [59]. The attacker triggers a page fault but suppresses it via fault handling, TSX transactions, or misspeculation. While the CPU knows that the access is not allowed, the exception is only raised at the retirement stage. Hence, dependent instructions receive the data and can then, e.g., encode the value in the cache which the attacker can leak using a technique like Flush+Reload.

2.3 Intel SGX

In recent years, software vendors discovered that specific security properties, e.g., for DRM, in theory, are much easier to achieve with trusted-execution mechanisms. Consequently, hardware vendors reacted and developed different trusted-execution environments [1, 39, 50]. Intel developed an instruction-set extension called Software Guard Extension (SGX) [39]. With SGX, an application is split into a trusted and an untrusted part. To protect the former, it is executed within a hardware-backed enclave. In the SGX threat model, neither the OS nor any other application is trusted. Therefore, the CPU guarantees that any memory belonging to the enclave cannot be accessed by anyone else than the enclave. The SGX threat model also allows the remaining hardware to be malicious or compromised. Consequently, the SGX memory is encrypted, protecting it from being directly read from the DRAM module. Additional threats like memory-safety violations [56], side channels [8, 83], or race conditions [78, 92] are considered out of scope and remain an enclave developer's responsibility.

The SGX interface to let the untrusted part enter an enclave conceptually resembles system calls. Once the trusted execution is finished, the result of its computation as well as the control flow is returned to the callee. However, SGX protection mechanisms are one-sided: SGX allows data sharing between the trusted and the untrusted part by giving enclaves full access to the entire host application's address space. Recently, it has been shown that this asymmetric protection gives rise to enclave malware [82].

2.4 Address Translation

Modern OSs rely on memory isolation for security purposes. Hence, CPUs support virtual memory for abstraction and memory isolation. Processes work on virtual addresses and cannot architecturally interfere with each other as the virtual address spaces are non-overlapping and overlapping areas are protected according to the processes' requirements. These virtual addresses have to be translated to physical addresses using multi-level page tables. A dedicated translation-table register indicates the location of the first-level table, e.g., CR3 on Intel architectures. Upon a context switch, the OS updates the translation-table register with the physical address of the first-level page table of the process scheduled next. Page-table entries do not only provide translations but also define properties of memory regions, e.g., executable or not.

2.5 Address Space Layout Randomization

Since the introduction of non-executable (NX) bits, memory corruption attacks have to rely on existing code in the victim process instead of code injection [87]. Shacham et al. [84] generalized the concept of code-reuse attacks, which is now widely known as return-oriented programming (ROP). Subsequently, a variety of code-reuse attack techniques have been described [6, 12, 13, 27, 76].

Code-reuse attacks require knowing addresses of specific code snippets. Similarly, data-only attacks [11, 47] require knowledge of addresses, e.g., of specific data structures. Over the years, many different mitigation techniques have been developed [87], e.g., NX stacks, stack canaries, and ASLR. The idea behind ASLR is to make the addresses of code and data unknown to an attacker by randomizing them. Typically, ASLR randomizes the base address of the executable, stack, heap, and shared libraries. Hence, even if an attacker hijacks the control flow, it is significantly harder to exploit bugs in an application as the location of code snippets usable for code-reuse attacks is unknown. By brute-forcing the location, the chances are high that the process will crash, and any ongoing attack is unsuccessful.

Furthermore, an application is re-randomized on every startup, reducing the chances of a successful attack.

General Idea of KASLR. While ASLR initially only protected user-space applications, the kernel space was later on also protected by KASLR [19, 49], e.g., introduced in Windows in 2007 [49], macOS in 2012 [2], and Linux in 2014 [19]. The kernel consists of multiple segments that are individually mapped into the kernel address space. These segments include the code (*i.e.*, text segment), drivers or modules, and data (e.g., stack, heap). The KASLR implementations of the three major OSs (Linux, Windows, macOS) only use coarse-grained randomization, *i.e.*, randomized base address. Fine-grained KASLR implementations using code diversification have been proposed [26, 72] but are not used in practice.

Another property of KASLR implementations is that the kernel is mapped using either 4 kB or 2 MB pages. The mapping is 2 MB-aligned [77], reducing the number of possible offsets. Moreover, the order of the randomized segments is not changed, e.g., in Linux, the text segment always has a lower address than the modules [57]. Consequently, KASLR provides a lower entropy than typical user-space ASLR implementations [19]. However, if an exploit attempt fails, it likely crashes the kernel. Hence, an attacker only has one shot, and exploitation techniques relying on a large number of retries cannot be used against the kernel if KASLR is active.

Linux. In Linux 5.x, most sections are independently randomized at boot, including the direct-physical map, vmalloc and ioremap space (vmalloc area), virtual-memory map (vmemmap), text segment, and modules [23]. The text segment is mapped between `0xffff ffff 8000 0000` and `0xffff ffff c000 0000` with a maximum size of 1 GB [77]. As the kernel has to be aligned to a 2 MB boundary, the randomization has 9 bits of entropy. Therefore, the kernel is placed at one of 512 possible offsets. Modules are mapped using 4 kB pages in a 1 GB range following the text segment. Unmapped pages follow each module before a new module starts [49].

Start and end addresses for the direct-physical map, the vmalloc area, and the vmemmap are documented [57], but analyzing the start addresses on repeated restarts shows that they are only correct if KASLR is disabled. Therefore, we analyzed the KASLR implementation of Linux kernel version 5.2.9. This analysis showed that the possible start address is indeed

`0xffff 8880 0000 0000` for the direct-physical map. It is then placed at a random offset from the start address, aligned to a 1 GB boundary. The vmalloc space is placed at a random offset relative to the end of the direct-physical map with at least 1 GB between them. The vmemmap area is then randomized starting from the end of the vmalloc area, again with at least 1 GB between them. The range of possible addresses is, therefore, from `0xffff 8880 0000 0000` to `0xffff fdff ffff ffff`, always with a 1 GB alignment and the preserved order.

Windows. Windows randomizes almost everything except the HAL heap once at boot [44]. Windows first introduced KASLR with Vista [49] and improved it over time [32]. Windows 7 maps the kernel, followed by the drivers in the same range with the same randomization. The address range of the kernel and drivers is `0xffff f800 0000 0000` to `0xffff f803 ffff ffff` [49]. KASLR on Windows 10 differs from Windows 7 as there is a separate area for the kernel and drivers. The kernel is still mapped in the same virtual address range, but drivers are now mapped in the range of `0xffff f800 0000 0000` to `0xffff f80f ffff ffff` [22]. The kernel is also 2 MB-aligned, resulting in 8192 possible offsets. Drivers are mapped with 4 kB pages with a 16 kB alignment.

macOS. Starting with macOS 10.8 (Mountain Lion), the kernel, `kexts` (kernel modules), and zones are randomized [70]. For instance, the kernel is mapped in the range of `0xffff ff80 0000 0000` to `0xffff ff80 2000 0000` with a 2 MB alignment, resulting in 256 possible offsets. The offset at which the kernel is placed relative to the start of the address range is called *kslide*. According to Chen and He [14], kernel and `kexts` share the same `kslide`.

3 A Novel (K)ASLR Break

In this section, we first analyze the Meltdown hardware mitigation on new Intel CPUs. We then introduce EchoLoad, an attack primitive that exploits incomplete Meltdown countermeasures to break KASLR. We detail how we can use it to break KASLR from an unprivileged user-space application, JavaScript, and SGX.

3.1 Analyzing the Meltdown Mitigation

The Meltdown vulnerability allowed unprivileged users to leak kernel memory (cf. Section 2.2). The immediate workaround was KAISER [30], a software-only solution to unmap the kernel when running in user space. With the Whiskey Lake microarchitecture, Intel fixed the vulnerability in hardware without providing further details on how their fix works. CPUs with the hardware mitigation indicate that they are not vulnerable by having the `RDCL_NO` bit set in the `IA32_ARCH_CAPABILITIES` model-specific register [39].

Lipp et al. [59] argued that stalling the CPU until the permission check is done might be too costly. We suspect that such a change also requires redesigning a significant part of the CPU's pipeline. As the first CPUs with hardware mitigations already shipped approximately one and a half years after Meltdown was disclosed to Intel, we expect only minor hardware changes as mitigation.

Hypothesis. We hypothesize that instead of stalling on an illegal memory load, the CPU zeroes out the result. Hence, the CPU still loads inaccessible memory locations, but instead of providing the real value to dependent instructions, it always provides '0'.

Verification. We get the first indication that our hypothesis is correct by simply mounting a Meltdown attack. When running the Meltdown attack on a Xeon Silver 4208 CPU which has the `RDCL_NO` bit set, we always get '0's. To verify our hypothesis, we further analyzed performance counters on three different systems: a Meltdown-vulnerable Intel CPU (i7-8650U), an Intel CPU with hardware mitigations (Xeon Silver 4208), and a non-affected AMD CPU (Ryzen Threadripper 1920X). For all systems, we evaluate performance counters when executing the following code 10^4 times: `if (transient_begin()) { *(volatile char*)0; oracle[*address]; }`. The function `transient_begin` either starts a TSX transaction if available, or sets up a signal handler for segmentation faults [59]. The null-pointer access is required to always cause an exception.

The first performance counter of interest is the number of CPU stalls when executing the above code. On Intel CPUs, we use `CYCLE_ACTIVITY.STALLS_MEM_ANY`, and on AMD CPUs the "Dispatch Stalls" counter. We set `address` to a valid kernel address. As baselines, we choose a mapped user address as well as a non-present address for `address`. Figure 6.1 shows the results of the performance counters for all 3 systems. For

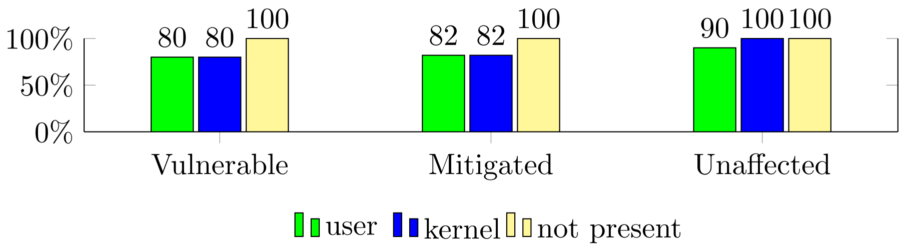


Figure 6.1: Loads from non-present pages always stall, loads to kernel addresses stall on unaffected AMD CPUs.

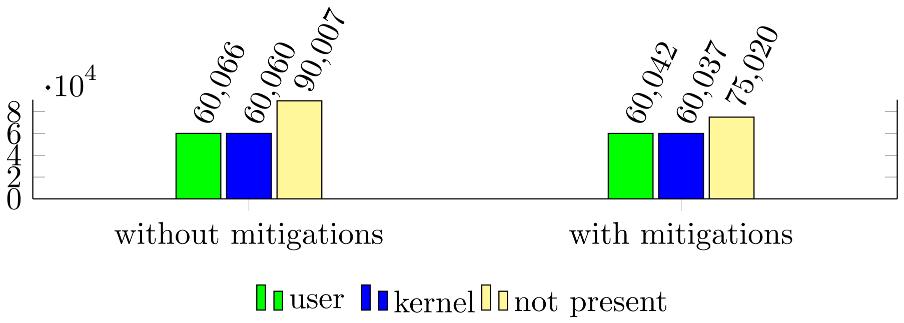


Figure 6.2: Issued load μOPs for user and kernel addresses (Intel). Only invalid loads from non-present pages are reissued.

comparison, we normalized the values such that the highest value on each system represents 100 %.

All CPUs stall when accessing a non-present virtual address. The AMD CPU also stalls when accessing a kernel address. Both Intel CPUs with and without mitigations show the same stall behavior. Hence, even the Intel CPUs with Meltdown mitigations do not stall when accessing a kernel address. This indicates that the memory load for the kernel address is actually issued.

We substantiate this observation by analyzing another performance counter. With the counters `UOPS_DISPATCHED_PORT.PORT_2` and `UOPS_DISPATCHED_PORT.PORT_3`, we can track the number of μOPs issued on the load ports. The sum of these two counters is the number of all memory loads. Figure 6.2 shows the number of memory loads when running the code mentioned above with a user-space, kernel-space, and non-present address both on an Intel CPU with and without hardware mitigation. When trying to load from a non-present page, the load faults and the

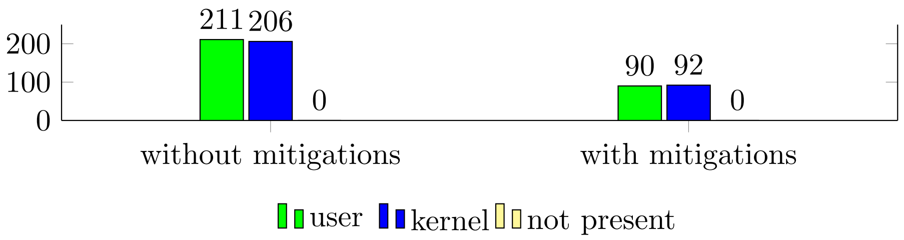


Figure 6.3: Number of cycles L1D cache misses are pending. User and kernel addresses reach the memory hierarchy, non-present pages do not.

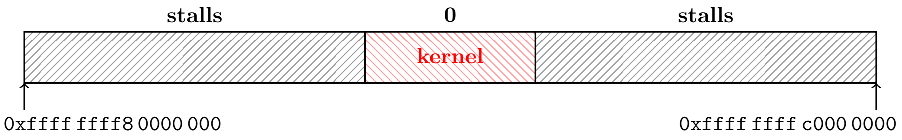


Figure 6.4: Reading addresses not physically backed stalls the CPU, while kernel addresses return ‘0’ (or the actual data).

load instruction is re-issued [80]. The number of issued loads for kernel addresses is the same as for user-space addresses on both CPUs. This indicates that these loads succeed and do not have to be re-issued.

Finally, we show that the issued loads for kernel addresses indeed load data from the memory hierarchy, and not, e.g., from an internal buffer containing ‘0’. Thus, we monitor the number of cycles that L1 data-cache misses are waiting to be retrieved. Figure 6.3 shows the values of the performance counter `L1D_PEND_MISS.PENDING_CYCLES` for the previously shown code. While non-present pages do not cause an L1 miss, both user-space and kernel addresses cause L1 misses. This is even the case for CPUs with hardware mitigations against Meltdown, showing that loads to kernel addresses retrieve the actual value, and only later on zero it out.

3.2 Breaking KASLR

EchoLoad is a new microarchitectural KASLR attack exploiting Meltdown-related side effects. EchoLoad reliably breaks KASLR, regardless of OS, software mitigations, and microcode updates. EchoLoad works on all Intel CPUs since 2010, even if they are not affected by Meltdown, e.g., CPUs with the RDCL_NO bit. In contrast to the KASLR break by Schwarz et al. [77], EchoLoad also works on the new Cascade Lake, which is not affected by Meltdown or MDS.

```
if (transient_begin()) {
  *(volatile char*)(mem + *address);
}
if (flush_reload(mem)) return ADDRESS_MAPPED;
else return ADDRESS_NOT_MAPPED;
```

Listing 3.1: The main part of EchoLoad. The address `mem` is only cached if the access to `address` does not stall.

General Idea. The general idea is to distinguish whether accessing a kernel address in the transient-execution domain leads to a stall. We exploit the fact that instructions can only be executed out of order if their data dependencies are fulfilled. Hence, we dereference a user-space memory location where the address is computed based on the value of the kernel address that is being tested.

Listing 3.1 shows this central part of EchoLoad. First, an attacker induces transient execution by provoking a fault or a misspeculation in Section 3.2. If the access to `address` in Section 3.2 stalls, the user address cannot be computed before the transient execution aborts. Otherwise, the user address is dereferenced and, thus, cached before the transient execution aborts. After the transient execution, the attacker probes the user- address in Section 3.2, e.g., using Flush+Reload. If the user address is cached, `address` is valid, *i.e.*, physically backed. Otherwise, `address` is not valid, *i.e.*, not physically backed. Figure 6.4 illustrates the general idea of EchoLoad.

CPUs with Meltdown Fixes. To break KASLR on CPUs with Meltdown fixes, we run EchoLoad on all 512 possible kernel offsets (cf. Section 2.5). Only where a physical page backs the tested address, we read 0, on other addresses the CPU stalls.

As the CPU stalls on all reads from addresses that the kernel is not mapped to, we observe no false positives. This makes EchoLoad a very reliable attack that even works on Cascade Lake CPUs.

CPUs without Meltdown Fixes. On CPUs without Meltdown fixes, we cannot rely on the CPU returning 0 for reads on kernel pages. Instead, if KPTI is disabled, we read the actual content of the page. As the content of the page is code, there are 256 possible addresses which could be dereferenced.

As the 256 possible addresses are contiguous, and the cache line size is typically 64 byte, they fall into one out of 4 possible cache lines. Testing 4 adjacent cache lines with Flush+Reload triggers the stride prefetcher [38] on Intel CPUs. Instead, we can exploit the L2 adjacent cache line prefetcher (spatial prefetcher) [38], which fetches the sibling cache line whenever a cache miss is handled. Hence, we only have to check 2 cache lines using Flush+Reload, which works without triggering the stride prefetcher. Consider a case with 4 adjacent cache lines. If the data we read falls into line 0 and we check line 1, we observe a hit on line 1 because the prefetcher also loads it into the cache. The same is true if the data falls into line 3, and we check line 4. By merely checking cache lines 1 and 3, we detect all possible accesses.

We can even further increase the performance by only checking one cache line. By using a kernel module, we investigated the beginning of the kernel text segment and determined that it is always the same across kernel versions (*i.e.*, `0x48`).

EchoLoad also works with KPTI [29] as the pages still mapped with KPTI use the same randomization offset as the rest of the kernel code. While the value differs with KPTI (*i.e.*, `0xf`), it is still the same across kernel versions that use it. As we only look for the beginning of the kernel and we know that the value remains constant, we can reduce the number of cache lines we need to check to 1. This further improves the performance of our KASLR break.

EchoLoad and LVI-NULL. On CPUs that have already received fixes for Meltdown, EchoLoad is the inverse of the LVI-NULL attack [89]. While LVI-NULL abuses the fixes to inject a dummy value of zero to dependent transient instructions in a victim, EchoLoad exploits the inverse effect, *i.e.*, the retrieving of a zero value, to break KASLR.

Table 6.1: Environments where we evaluated EchoLoad and Data Bounce (KPTI disabled).

CPU	**μarch.**	**EchoLoad**	**Data Bounce**
Intel Atom x5-Z8300	Cherry Trail	✓	✓
Intel Core i5-450M	Arrandale	✓	✓
Intel Core i5-3230M	Ivy Bridge	✓	✓
Intel Core i5-8250U	Kaby Lake R	✓	✓
Intel Core i7-4790	Haswell	✓	✓
Intel Core i7-6700K	Skylake	✓	✓
Intel Core i7-8650U	Kaby Lake R	✓	✓
Intel Core i7-8565U	Whiskey Lake	✓	✓
Intel Core i9-9900K	Coffee Lake	✓	✓
Intel Xeon E5-1630 v4	Broadwell	✓	✓
Intel Xeon Silver 4208	Cascade Lake	✓	✗
Intel Cascade Lake (Google Cloud)	Cascade Lake	✓	✗
AMD Ryzen Threadripper 1920X	Zen	✗	✗
AMD Ryzen 7 3700	Zen 2	✗	✗
ARM Cortex-A57	A57	✗	✗

Evaluation. We evaluated EchoLoad on different Intel microarchitectures running Linux (cf. Table 6.1). On all CPUs, we evaluated our attack with both KPTI enabled and disabled. These experiments show that KPTI does not prevent EchoLoad. If KPTI is disabled, EchoLoad detects the symbol `startup_64`. With KPTI, it detects the symbol `__entry_text_start`, which is the trampoline required to enter the kernel. As Android is based on the Linux kernel, the behavior on Android is the same. While we evaluate the ability and performance of EchoLoad to leak the kernel code offset, it can equally leak the offsets of all other randomized parts of the kernel.

For the performance evaluation, we used the same setup that Schwarz et al. [77] describe in their paper. We tested 10 different randomizations (*i.e.*, 10 reboots), each 100 times. Using this approach, we have a sample size of 10^3. We evaluated the performance on a selected number of architectures in all three cases, namely misspeculation, TSX, and segfault handling. Table 6.2 shows the result of this evaluation. In all tested cases, we achieve almost perfect accuracy. On the i9-9900K, we outperform Data Bounce in terms of time required while matching the accuracy.

Similar to Schwarz et al. [77], we tested EchoLoad with TSX on a larger scale. For that, we tested the same offset 100 million times and repeated the experiment 10 times for a total of 1 billion tries. On all three

Table 6.2: Performance of EchoLoad in terms of runtime and F-score. Each possible offset is tested a single time.

CPU		Speculation		TSX		Segfault	
i7-6700K	Time (F-Score)	63 µs	(0.999)	48 µs	(1.000)	133 µs	(1.000)
i9-9900K	Time (F-Score)	33 µs	(1.000)	29 µs	(1.000)	86 µs	(1.000)
Xeon Silver 4208	Time (F-Score)	51 µs	(0.994)	40 µs	(1.000)	127 µs	(1.000)

CPUs (cf. Table 6.2, we achieved an average F-score of 1, giving us perfect accuracy in detecting the KASLR offset.

On Windows 10, we also tested 10 different randomizations (*i.e.*, 10 reboots), each 100 times. In all cases, we successfully found the location of the kernel image. On macOS 10.11.6, instead of 10 randomizations, we repeated the experiment 100 times to verify that the given kernel range is still correct [14]. We then successfully recovered the kernel location in all 100 randomizations.

3.3 Breaking (K)ASLR from SGX

As EchoLoad only requires memory accesses, it also works in restricted environments. We demonstrate EchoLoad in SGX enclaves breaking host ASLR, victim-enclave ASLR, and KASLR.

While it is also possible to use EchoLoad for detecting the location of SGX enclaves from the host application, this is an artificial scenario. First, the host maps the enclave to its location and, thus, knows where the enclave is. Second, on Linux, the host can access this information from the pseudo file `/proc/self/maps`, containing all virtual-address mappings of the current process. Finally, the host can also probe the virtual memory for the enclave, e.g., using a signal handler to catch segmentation faults. If a region returns `0xff`, it is likely to be an EPC page of an enclave.

EchoLoad from Enclave to Host. *TAP* is a method to break host ASLR from an enclave using Intel TSX [82]. It allows scanning the host address space for mapped pages to mount a ROP attack from inside the enclave, impersonating the host application.

While *TAP* only worked for CPUs with TSX, it does not work on CPUs with MDS fixes in microcode at all. With the microcode update, all TSX transactions abort immediately when started inside an SGX enclave [41]. We further analyzed whether the transaction aborts immediately, or is only rolled back in all cases.

```
if(xbegin() == (~0u)) { *(volatile char*)mem;
    xend(); }
if(flush_reload(mem)) return ROLL_BACK;
else return IMMEDIATE_ABORT;
```

Listing 3.2: Analyzing the behavior of the TSX abort. If the transaction is aborted on `xbegin`, `mem` cannot be cached. If the transaction is just rolled back on `xend`, `mem` is cached.

Listing 3.2 shows the code we use to analyze the TSX-transaction aborts. If the transaction aborts already at the `xbegin` instruction, the memory dereference is never executed. If the transaction executes but then rolls back the executed instructions, the dereference of the address still causes the memory location to be in the cache.

Our results show that the transaction is never started as the address `mem` is never cached after the transaction. Hence, we cannot even use TSX to access memory locations transiently. We observe the same behavior outside an SGX enclave when setting the `TSX_FORCE_ABORT` MSR to 1. While this MSR is documented to abort every TSX transaction on *commit* [98], we verified with our test (Listing 3.2) that the transaction is not even started.

Consequently, even if TSX is re-enabled in SGX via a microcode update, it can be manually disabled with the `TSX_FORCE_ABORT` MSR to protect against attacks such as *TAP*. This is the case on the Amazon EC2 cloud [80]. In contrast to Data Bounce [77], EchoLoad works on the newest CPU generation, as it does not require TSX. Thus, EchoLoad can be used to mount SGX ROP attacks [82] even if TSX is disabled, once more enabling such attacks.

Due to the unavailability of syscalls and the `rdtsc` instruction inside SGX, we mount EchoLoad behind a misspeculated branch and use a counting thread [83] as a timer. We achieve a speed of 388 Mb/s for scanning the host address space with EchoLoad. Hence, EchoLoad is a viable alternative to *TAP* to de-randomize the host application from an SGX enclave.

EchoLoad from Enclave to Enclave. Enclaves might not only want to de-randomize the host application but also learn information about other enclaves. While enclaves are mutually untrusted and, thus, cannot access each other, EchoLoad can be used to learn the address-space layout of other enclaves. Moreover, assuming that enclaves have unique sizes, an enclave can even detect which other enclaves are used by the host by detecting their size.

We evaluated EchoLoad in the cross-enclave scenario by loading two enclaves in our test application. One enclave is malicious and leverages EchoLoad to learn which other enclaves are used by the host application. We use the same experiment as for de-randomizing the host to scan the address space for other enclaves. We successfully detect the location and the size of the second enclave used by the host application. The speed for scanning the address space is the same as for de-randomizing the host application.

EchoLoad from Enclave to Kernel. Enclaves may foster stealthy exploits [48, 65, 82, 83]. In this work, we add another primitive to malware hidden inside SGX. With EchoLoad, an enclave can de-randomize KASLR, which is a prerequisite for many kernel exploits.

The same code which is used to de-randomize the host application can be used to de-randomize the kernel. We evaluated EchoLoad inside an SGX enclave to find the KASLR offset. Due to the use of misspeculation and a timing thread, the performance is worse than in native code. However, we still detect the KASLR offset with an F-score of 1 ($n = 10^3$).

3.4 Meltdown and KASLR Break in JavaScript

EchoLoad can even be mounted from a JavaScript sandbox. We demonstrate EchoLoad, and as an extension Meltdown, from the Spidermonkey JavaScript engine 60.1.3 used in Firefox.

There are two challenges for mounting EchoLoad in JavaScript. First, both JavaScript and WebAssembly currently only support a 32-bit linear memory index, restricting arrays to 4 GB [43]. While this prevents EchoLoad on a 64-bit OS, it does not prevent it on 32-bit OSs, which only support a 32-bit virtual address space. Hence, we evaluate this attack on Ubuntu 16.04 (Kernel 4.15.0-60) i686 on an Intel i7-4790. While we are currently limited to 32-bit systems, the WebAssembly developers are planning to increase the size of linear memory indices from 32-bit to 64-bit, allowing the attack on all commodity systems that are not patched against Meltdown [91]. Second, the Spectre mitigations do not only reduce the resolution of the high-resolution timer [67], but also harden the bounds check for arrays, preventing speculative out-of-bounds accesses by default [66]. As our focus is not demonstrating a Spectre attack but a Meltdown-related effect, we use a version of the engine that allows speculative out-of-bounds accesses, as in previous work [53]. To develop widely deployable Meltdown and EchoLoad exploits, further research is necessary to investigate whether other misprediction mechanisms may provide a suitable workaround to the hardened out-of-bounds checks. Note that previous work has already shown that some of these mitigations can be circumvented [34].

Building Blocks. An alternative to the high-resolution timer is a counting thread which is commonly used for microarchitectural attacks in JavaScript [28, 53, 81]. Furthermore, as the `clflush` instruction is not available in JavaScript, we resort to Evict+Reload as described in related work [28, 77, 90]. Instead of measuring only one address in our Evict+Reload, we use amplification on multiple cache lines [63]. With amplification, we encode the out-of-bounds access into multiple different cache lines to achieve more reliable results. To access a kernel address during transient execution, we hide an out-of-bounds array access behind a misspeculated branch.

EchoLoad from JavaScript. By combining the building blocks, we can implement EchoLoad in JavaScript. On average, it takes 25.09 ms ($n = 10^3$, $\sigma_{\bar{x}} = 5.92$) to find the start of the kernel image. The detected offset is relative to the base of the array, which is used for the out-of-bounds accesses. However, an attacker can leverage any JavaScript ASLR break [28] to recover the array base address, and from that compute the absolute address of the kernel image.

Meltdown from JavaScript. Contrary to Linux, many 32-bit OSs still in use do not have Meltdown patches (e.g., Windows XP). Hence, we show that with the building blocks, we can mount a Meltdown attack from JavaScript on such systems. Relying on EchoLoad for the KASLR break, we can even target specific locations in the kernel.

To evaluate the attack performance of the proof of concept, we disable KPTI and leak a known value from the kernel. Our JavaScript attack leaks 2 B/s, with an error rate of 0.3 % ($n = 10^3$).

3.5 Other Side-Channel Attacks on KASLR

Microarchitectural attacks on KASLR so far relied on either branch-predictor states [20], address-translation caches [32, 36, 49], or store-buffer optimizations [10, 77]. We compare EchoLoad to previous attacks on KASLR [20, 32, 36, 49, 77].

Our attack outperforms all state-of-the-art KASLR breaks on Intel x86 CPUs (cf. Table 6.3). We outperform Data Bounce [77] in terms of speed

Table 6.3: We compare microarchitectural attacks on KASLR. EchoLoad outperforms all previous microarchitectural attacks on KASLR while having no requirements.

Attack	Time	Accuracy	Requirements
Hund et al. [36]	17 s	96 %	-
Gruss et al. [32]	500 s	N/A	cache eviction
Jang et al. [49]	5 ms	100 %	Intel TSX
Evtyushkin et al. [20]	60 ms	N/A	BTB reverse engineering
Canella et al. [10]	0.27 s	100 %	MDS vulnerable CPU
Schwarz et al. [77]	42 µs	100 %	Intel CPU before Cascade Lake
EchoLoad (our attack)	29 µs	100 %	-

and match it in accuracy while having lower requirements. Similar to Data Bounce, EchoLoad also has the advantage over previous microarchitectural attacks that it does not require Intel TSX, or knowledge of internal data structures like the branch-target buffer (BTB) or the store buffer.

For instance, Evtyushkin et al. [20] assume an attacker knows how the BTB works internally, which has not yet been reverse-engineered for microarchitectures after Haswell. Moreover, with the widely-deployed Spectre mitigations [9, 40], the BTB is either cleared on context switch or not shared between privilege levels. Hence, this attack does not work on state-of-the-art CPUs anymore.

The double page-fault attack [36] was the first microarchitectural attack on KASLR. By accessing a kernel memory location, an attacker first triggers a page fault. This triggers an interrupt which is handled by the OS. After handling the interrupt, the OS returns control to a pre-installed error handler in the user-space program. In the error handler, the attacker measures the time it took to handle the fault. The attacker then repeats the attack step, again measuring the time it took to handle the fault. If the kernel address is valid, the first illegal access has created a TLB entry. This speeds up the handling of the second fault, creating a timing side channel. Consequently, a user-space attacker can infer whether a kernel address is valid or not. The requirement for this attack is that the user can install a signal handler to handle segmentation faults. Hence, native code execution is required.

Jang et al. [49] retrofitted the attack by Hund et al. [36] with Intel TSX. TSX is an x86 instruction-set extension introducing hardware transactional memory. If a page fault occurs within a transaction, it is aborted without architecturally raising a fault and, hence, without any OS interaction. This allows the attack to skip the page fault handling of the OS, significantly speeding up the attack and reducing its noise. The approach by Jang et al. [49] only works on CPUs starting from Haswell as it relies on Intel TSX. This extension is not present on low-end CPUs or any CPUs built before 2013 and can be disabled on newer CPUs as well. Intel TSX is, for example, disabled on the Amazon EC2 cloud [80].

Gruss et al. [32] use the software prefetch instruction as a side channel. This side channel exploits that the execution time of the prefetch instruction depends on whether the translation cache holds the correct entry. As the TLB can only hold addresses for which a valid translation, *i.e.*, a physical page is mapped to it, the location of the kernel is revealed due to it consisting of the only valid address mapping within the predefined region. With this attack, the attacker additionally learns the page size that is used for the mapping.

Fallout [10] demonstrates a KASLR break on MDS-vulnerable Intel CPUs. First, they ensure that a user-controlled value is in the store buffer. Then, they attempt to access an address with the same *page offset* ,which is inaccessible. On MDS-vulnerable CPUs, the store-buffer content is transiently forwarded to faulting loads on valid kernel addresses, revealing the location of the kernel. Fallout [10] relies on the opportunistic store-buffer behavior that virtual addresses are likely equivalent if the least-significant 12 bits match. However, this is only the case on MDS-vulnerable Intel CPUs that are not patched. Hence, this KASLR break does not work on CPUs indicating that they are MDS-resistant via the `MDS_NO` flag in the `IA32_ARCH_CAPABILITIES` model-specific register, e.g., on the newest Cascade Lake CPUs.

Data Bounce [77] breaks KASLR by exploiting that the CPU only performs store-to-load forwarding if a physical page backs a virtual address, *i.e.*, the virtual address can be resolved to a physical address. Using this approach, they can break KASLR on all Intel CPUs going back to 2004. They claim that the attack has perfect accuracy and only requires 42 µs to detect the correct kernel location. One of the advantages of this approach over Jang et al. [49] is that it does not require TSX and, hence, is applicable to a broader range of CPUs. However, the behavior of store-to-load forwarding was changed in Cascade Lake CPUs to prevent this attack (cf. Table 6.1). Hence, while their approach works on microarchitectures starting from the Pentium 4 Prescott to Whiskey Lake and Coffee Lake R, it does not work on the recent Cascade Lake.

EchoLoad relies on the load stalling behavior of the CPU, an effect which has not been exploited so far. As this effect is deeply rooted in the design of the microarchitecture, it cannot easily be fixed (cf. Section 3.1), neither in software nor hardware. Moreover, the attack does not have any requirements as it solely relies on memory loads. As a consequence, even the most recent Cascade Lake is affected by EchoLoad.

4 FLARE: Mitigating KASLR Breaks

In this section, we propose FLARE, a defense against KASLR attacks rooted in a CPU's microarchitecture.

FLARE has a negligible memory overhead of only a few kilobytes and next to no runtime overhead. FLARE tackles the root causes of all the microarchitectural KASLR breaks discussed in Section 3.5. It builds on ideas from KAISER [30] and LAZARUS [25] to fix remaining weaknesses efficiently and securely.

The challenge is to fully eliminate differences in:

C1: timing and behavior for mapped and unmapped pages,
C2: timing for different page sizes, and
C3: timing between executable and NX pages.

As we show in this section, FLARE successfully tackles these challenges. However, before we justify these challenges, we briefly introduce a threat model. We then discuss implementation details, corner cases, and pitfalls in Section 4.1.

Threat Model. Our attacker can run unprivileged native code on an up-to-date OS. Furthermore, the attacker knows the exact version of the Linux kernel that the victim uses and, hence, knows the exact structure of the kernel image in memory.

C1: Differences for Mapped and Unmapped Pages. In Section 3.5, we discuss that recent attacks, including EchoLoad, can distinguish mapped from unmapped pages [10, 32, 36, 49, 77]. Therefore, the first challenge is to prevent an attacker from detecting the KASLR offset based on that information.

To tackle this challenge, we map all unmapped virtual addresses in the randomization range to a dummy physical page. Therefore, none of the known attacks that rely on distinguishing mapped from unmapped addresses can de-randomize the kernel anymore.

C2: Timing Differences for Page Sizes. In Section 3.5, we discuss that the attack by Gruss et al. [32] can distinguish different page sizes:

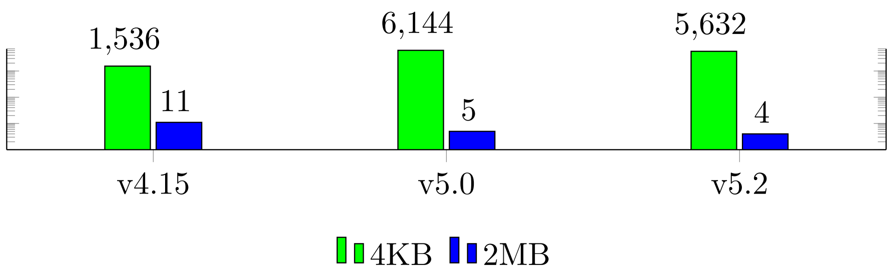


Figure 6.5: On recent Linux, several 2 MB pages in the kernel text segment have been replaced by 4 kB pages.

Even if the entire kernel space has a valid mapping, different page sizes can create a unique pattern which de-randomizes the kernel. This is especially a problem as the kernel uses different page sizes for its mapping (cf. Figure 6.5), possibly creating such a unique pattern. We tackle this challenge by avoiding different page sizes in the kernel altogether.

C3: Timing Difference between Executable and NX Pages. Jang et al. [49] showed that there is a timing difference between executable and NX pages. We analyzed the kernel and discovered that executable and NX pages are strictly separated. That is, after the first NX page in the address space there is not a single executable page in the remaining address space. To prevent this straightforward KASLR break, we randomize the executable and the NX range separately and pad them each with executable and NX pages respectively to the full randomization range.

4.1 Implementation Details

The different Linux kernel regions (cf. Section 2.5) are mapped with different properties, *i.e.*, different page sizes and permissions (e.g., executable and NX). Note that we only need to protect the trampoline code if KPTI is active, while we have to protect the following regions without KPTI.

Text Segment. Figure 6.5 shows that the text segment is mapped using both 4 kB and 2 MB pages. To address C1, we map the entire range where the text segment can be mapped using 4 kB pages, preventing the attacker from seeing the actual text-segment range. To address C2, we map the text segment only with 4 kB pages, preventing attacks that distinguish page sizes [32]. This is not a large kernel change as this is already an ongoing

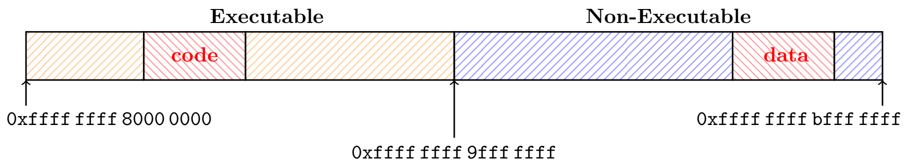


Figure 6.6: With FLARE, all possible kernel offsets are physically backed, *i.e.*, any potentially read value from this range will be zero. Code and data is independently randomized in 512 MB ranges. This setup allows preventing all currently known microarchitectural attacks on KASLR.

development (cf. Figure 6.5). The kernel already supports disabling the use of non-4 kB pages by clearing the CPU capability `X86_FEATURE_PSE`.

Furthermore, to tackle C3, we use the solution shown in Figure 6.6. We split the randomization range of the text segment in half. We then use one half for the randomization of executable pages, *i.e.*, the kernel code, and the other for NX pages, *i.e.*, the kernel data. Both regions are then randomized independently to not leak their corresponding start and end addresses. This split does not introduce any compatibility issues, even with relative addressing, as we stay within the maximum addressable range of 4 GB.

Modules. Modules already use 4 kB pages only, solving C2. In our proposal, we pad the code and data sections of every module to a multiple of 1 MB, depending on the size of the largest currently loaded module. We then map the remaining offsets in the address range with dummy modules using 4 kB pages that look exactly the same as the actual modules, *i.e.*, same size for code and data sections. Consequently, using the technique by Jang et al. [49] in the memory range for kernel modules, we only see executable and NX regions of all the same size. This mitigates the templating attack by Jang et al. [49] as the attacker cannot infer anymore which module is real and which one is not. With this approach, we solve all three challenges.

Naturally, the privileged user can dynamically load modules which takes the place of a previous dummy module. Likewise, for the unload, a dummy module replaces the kernel module mapping. However, the implementation should be careful not to leave a small time window open for an attack. In our FLARE proof-of-concept, we enable the loading of modules by first removing the dummy mapping by hooking the function *load_module*. Then the module is loaded, and afterward, the mitigation is re-applied. However, a proper implementation should exchange the page-table entries directly instead. This way, it is guaranteed that no time window is left for the attacker to observe the short unmapping from a concurrent microarchitectural attack, as there simply is no short unmapping. Furthermore, the loading and unloading of modules typically does not happen for an average user.

Direct-Physical Map, Vmalloc, Vmemmap. We analyzed how the direct-physical map, vmalloc, and vmemmap are mapped. None of the pages mapped in this region is executable. Thus, we tackle challenges C1 and C3 by mapping all pages in the corresponding randomization regions in our dummy mapping as NX.

Currently, the kernel does not use an explicit randomization range for each of the three regions. Instead, the kernel uses one large range and only guarantees to preserve their order. To mitigate the attack by Gruss et al. [32], all three must use the same page size. We verified that this is already the case when clearing the `X86_FEATURE_PSE` CPU capability at boot. As this causes significant pressure on the TLB, we propose a different approach.

We propose that the kernel uses an explicit randomization range for each of the three regions. Hence, to tackle C2, we can enforce that the kernel consistently uses one page size per region. This mitigates the attack by Gruss et al. [32]. In the analysis for our defense, we empirically determined the page sizes used for each region. On our test machine running Linux kernel 4.15, the kernel indicates during the boot process that 1 GB pages are used for mapping the direct-physical map. However, our analysis revealed that it is mapped using all three page sizes, *i.e.*, 4 kB, 2 MB, and 1 GB. Similarly, the vmalloc area uses both 4 kB and 2 MB pages. The vmemmap area consisted of 2 MB pages only.

Based on this analysis, we propose to consistently use 2 MB pages for the vmemmap region and 4 kB pages for the vmalloc region. For the direct-physical map, we use 1 GB pages. Unfortunately, we cannot use such a huge dummy page for our mapping as we would reduce the available physical memory by 1 GB. Instead, we pick 1 GB of RAM, which is already mapped in the direct-physical map, and map it using a 1 GB page in our dummy mapping. Hence, we avoid the additional memory overhead without increasing the risk for exploitation as we map the page as NX.

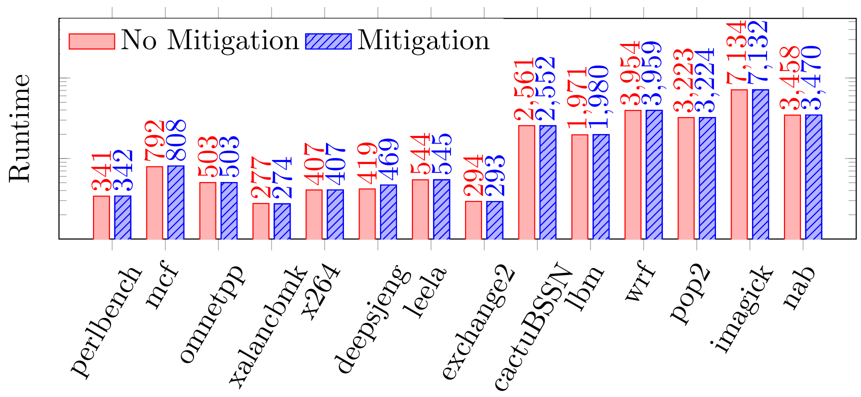


Figure 6.7: Runtime overhead of FLARE on SPEC CPU 2017.

5 Evaluation

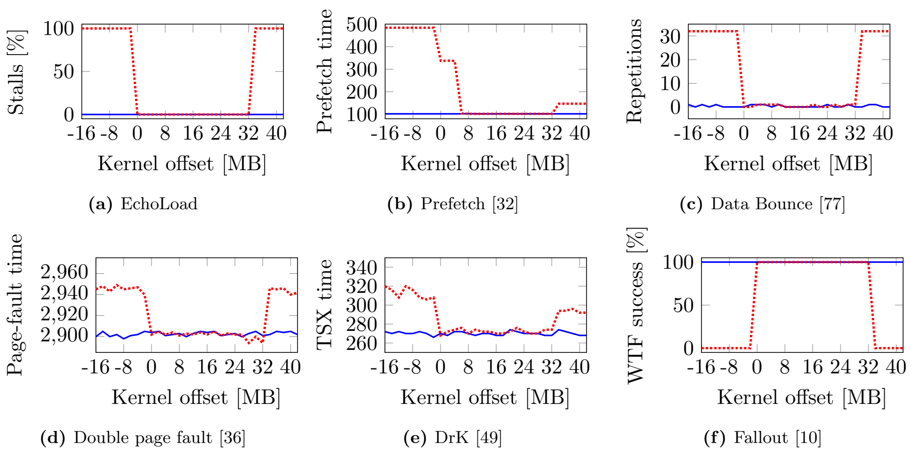
In this section, we evaluate the overhead of FLARE in three aspects, namely runtime overhead using the SPEC CPU 2017 benchmarks [16], module loading overhead, as well as the memory overhead. We also evaluate the efficacy of FLARE by analyzing how successful it is in preventing microarchitectural attacks on KASLR.

5.1 Overhead Analysis

Runtime. We create our dummy mapping directly in the *init_mm* struct which is copied into every newly created process. We only have to apply our mapping once, and every new process has the mitigation enabled. Hence, we expect no runtime overhead.

We confirmed this using the LMbench microbenchmark suite [64]. We evaluated process-creation time (fork and exec) and context switches on an Intel i7-8650U (Linux kernel 5.0.0-15). This involves a larger number of TLB invalidations and address resolutions, *i.e.*, the situations that may see a performance penalty. For process creation, we do not encounter any overhead. Both with and without FLARE, the process creation takes on average 61.14 µs ($n = 10^5$, $\sigma_{\bar{x}} = 0.27$). Similarly, there is no difference in the syscall latency. In both cases, the latency is on average 1.03 µs ($n = 10^5$, $\sigma_{\bar{x}} = 0.006$).

For a real-world workload, we evaluated the runtime overhead using the SPEC CPU 2017 benchmark. We ran the benchmark once with our mitigation and once without it on an Intel Xeon Silver 4208. We excluded



(a) EchoLoad **(b)** Prefetch [32] **(c)** Data Bounce [77]

(d) Double page fault [36] **(e)** DrK [49] **(f)** Fallout [10]

Figure 6.8: Detecting the kernel (offset 0 to 32 MB) with all known microarchitectural attacks on KASLR without FLARE (⋯) and with FLARE (—). For all attacks, FLARE successfully prevents the detection of the kernel.

some benchmarks in both the *intspeed* and *fpspeed* benchmark as they already crashed or did not compile on our vanilla Linux system. Figure 6.7 shows the results. As expected by the design of FLARE, we exhibit next to no runtime overhead.

Module Loading. Next, we evaluated the increase in module loading time. We first establish a baseline by loading and unloading a simple test module 10^4 times. We then load the FLARE proof of concept, which requires removing and re-applying the dummy mapping for every module load, thus overapproximating the overhead of our mitigation. We again load and measure the required time 10^4 times. We only observe a 4 % increase from 2.39 ms to 2.48 ms per module load. When implemented in the Linux kernel, the module memory allocation logic is made aware of the dummy mappings so that they are treated like free memory. Thus, overheads are avoided entirely except in cases where the modules have to be re-padded, where we observe the overheads to be negligible.

Memory. Finally, we analyzed the memory overhead of FLARE, which is minimal in our proof of concept. We always map the same dummy page in the paging hierarchy and re-use the same page directory and page table. We do not need a new PDPT, as we are working on existing 1 GB ranges. Therefore, we only require one page each for the new page directory, page table, and one page to point to. As all these pages are 4 kB, the maximum overhead is 12 kB. To map huge dummy pages, the maximum overhead is only increased by 2 MB. The direct-physical map padding with 1 GB pages does not consume additional memory (cf. Section 4).

5.2 Mitigate Microarchitectural KASLR Breaks

In a first step, we evaluated the effectiveness of FLARE in preventing breaking the randomization of the kernel text segment. Using a vanilla Linux 5.0 kernel, we test microarchitectural attacks on KASLR that are not mitigated through orthogonal countermeasures (cf. Section 3.5) with KPTI disabled (cf. Figure 6.8). In all cases, we first establish a baseline of the attack without FLARE in place, which shows the exact position of the kernel with all attacks.

We then load FLARE and re-evaluate all attacks. We see for each attack that the kernel can no longer be distinguished from other positions. With EchoLoad (Figure 6.8a), all offsets are backed by a physical page, the load succeeds, but the CPU returns zero for the illegal access. The stall percentage is based on cache hits and misses on the probe array, not performance counter values. With the prefetch side channel (Figure 6.8b), we see that the prefetch instruction can now also prefetch all other possible locations, mitigating the KASLR break. Data Bounce (Figure 6.8c) also no longer distinguishes kernel locations from dummy mappings as store-to-load forwarding works for all possible offsets. The double-page fault (Figure 6.8d) as well as the DrK attack (Figure 6.8e) also do not work anymore, exhibiting the same timing across the whole address range. With Fallout (Figure 6.8f), we also see no difference anymore as every page allows to trigger the WTF effect. An attack that tries to detect our dummy mapping based on timing the page-table walk is also not possible. Even though the physical page is shared across all dummy mappings, a TLB entry for one mapping is not shared with another. Hence, each access to a new page requires a full page-table walk. Our dummy mapping can also not be uncoverd via the cache as an access to a privileged address does not load the data into it [39, 77, 80]. Based on the results shown in Figure 6.8, none of the currently known microarchitectural attacks that are not mitigated through orthogonal countermeasures (cf. Section 3.5) can de-randomize the kernel location despite FLARE. This empirically confirms that we solve challenge C1.

Next, we de-randomize the kernel based on the timing difference between executable and NX pages [49]. We confirm that tackling only C1

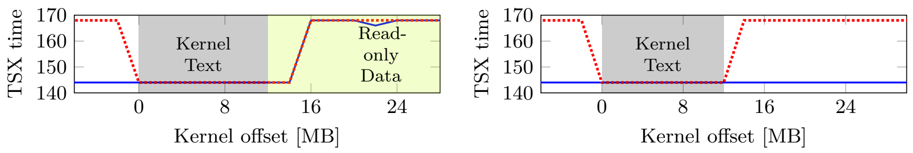


Figure 6.9: DrK [49] distinguishes executable from NX pages. An attacker can observe the switch to NX pages directly after the executable pages (left), when tackling only C1 and C2 (—) and entirely without (⋯) FLARE. With full FLARE (right), this attack is also fully mitigated.

and C2 is insufficient (cf. Figure 6.9). However, full FLARE (cf. Figure 6.9) separates the regions and the switch from executable to NX is not visible in this region anymore but at the pre-defined start of the randomization range (cf. Figure 6.6).

Next, we used the prefetch side-channel attack to try to break KASLR based on different page sizes. The different levels visible in the default case of Figure 6.8b show the different paging levels for the address we test. If nothing is mapped in the PML4, we observe the highest time. There is a drop in the access time for addresses with no PDPT entry, and another drop for addresses that map to an entry in the page table, *i.e.*, a 4 kB page. Thus, the prefetch side channel shows the different paging levels [32]. With FLARE in place, we can no longer see the difference in page sizes as all possible locations as well as the kernel are mapped using 4 kB pages. Thus, we empirically confirmed that our strategy for C3 works, defeating microarchitectural attacks on KASLR based on different page sizes.

6 Related Work

With the advent of KASLR, many different attacks have been proposed to break KASLR. One problem is that the kernels of the major OSs cannot change the randomization at runtime. Hence, if an attacker knows the KASLR offset, it is valid until the next time the OS is rebooted. So far, most of the attacks on KASLR relied either on software vulnerabilities or side-channel attacks on the microarchitecture as discussed in Section 3.5.

6.1 Software-based KASLR Breaks

On Linux, parts of kernel pointers are often disclosed inadvertently through kernel interfaces, e.g., due to uninitialized structure fields or structure padding [85]. There have been many such software vulnerabilities in kernels (e.g., CVE-2012-6138, CVE-2013-1825, 1826, 1827, 1873, 2634, 2635, 2636) that revealed parts of kernel addresses. Similarly, for Windows, multiple methods leak kernel pointers, e.g., using the `Win32ThreadInfo` or the Desktop heap [75]. Other attacks on KASLR relied on the fact that kernel addresses were used as unique identifiers [85], or as seed for pseudo-random numbers [52]. For debugging reasons, kernel addresses were often visible in log files or debugging interfaces such as the perf subsystem [18].

6.2 Mitigating Software-based KASLR Attacks

While software bugs causing KASLR breaks can be easily fixed, there are also general concepts for preventing address leakage from the kernel. Linux introduced a setting to mask kernel pointers in log files with a random mask [73]. Thus, a developer sees which pointers are the same, but an attacker cannot learn the actual pointer value and, thus, the KASLR offset. This mitigation reduces the risk of leaking the KASLR offset without impairing the debugging capabilities. The PaX Team proposed STACKLEAK [15], a mechanism to clear kernel-stack memory which is no longer in use. This reduces accidental address leakage from uninitialized stack values.

6.3 Mitigate Microarchitectural KASLR Breaks

While microarchitectural attacks on KASLR cannot be simply fixed in software, there are software-based workarounds. Gruss et al. [30, 32] proposed stronger kernel isolation to prevent microarchitectural attacks on the kernel by unmapping the kernel address space when running in user space. Thus, in theory, there is no valid kernel address in user space, preventing all microarchitectural attacks on the kernel. However, while their proposal is deployed on all major OSs to prevent Meltdown [59], it cannot prevent our KASLR break (cf. Section 3.2). The reason is that x86 requires some kernel pages always to be mapped, even when running in user space [30].

Lazarus [25] proposed a similar approach to KAISER [30]. It is based on fencing the kernel paging entries off from those of the user space by separating user and kernel page tables. Therefore, the Memory Management Unit can no longer use entries pointing to kernel space memory from user space. Contrary to KAISER, Lazarus uses dummy mappings to hide the context switching code while KAISER separates it from the rest of the kernel code section. However, it does not tackle the challenges we identified and does not defeat all known microarchitectural attacks on KASLR.

7 Conclusion

In this paper, we analyzed Intel's recent hardware fixes for Meltdown. Our analysis led to the understanding that illegal memory accesses do not lead to a CPU stall, but instead, the illegally loaded data is zeroed-out. With EchoLoad, we presented a novel technique based on Flush+ Reload to distinguish stalling loads from transiently executed ones. Hence, EchoLoad enables an attacker to detect physically-backed kernel addresses and break KASLR. Our KASLR break is the fastest and most reliable microarchitectural KASLR break presented so far, taking only 40 µs to de-randomize the kernel. The only requirement for EchoLoad are memory loads, allowing it to be mounted from restricted environments such as SGX and JavaScript. We presented the first JavaScript-based Meltdown attack and KASLR break on the systems that do not receive Meltdown patches, *i.e.*, x86 32-bit OSs.

With FLARE, we proposed a generic approach for protecting the kernel against microarchitectural KASLR breaks. We verified that FLARE mitigates the root cause behind current microarchitectural KASLR breaks and yields a uniform behavior across the kernel address space. Thus, considering the state of the hardware mitigations, we propose to deploy FLARE even on the most recent CPU generations.

Acknowledgments

We thank our anonymous reviewers for their comments and suggestions that helped improving the paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from Intel, ARM, and Cloudflare. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Tiago Alves. *TrustZone: Integrated Hardware and Software Security.* 2004.

[2] Apple Inc. *OS X Mountain Lion Core Technologies Overview.* 2012. URL: `http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf`.

[3] ARM Limited. *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism.* 2018.

[4] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way." In: *CHES.* 2014.

[5] Daniel J. Bernstein. *Cache-Timing Attacks on AES.* 2004.

[6] Erik Bosman and Herbert Bos. "Framing Signals - A Return to Portable Shellcode." In: *S&P.* 2014.

[7] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector." In: *S&P.* 2016.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *WOOT.* 2017.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium.* 2019.

[10] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-resistant CPUs." In: *CCS.* 2019.

[11] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity." In: *USENIX Security Symposium.* 2015.

[12] Nicholas Carlini and David A. Wagner. "ROP is Still Dangerous: Breaking Modern Defenses." In: *USENIX Security.* 2014.

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns." In: *CCS.* 2010.

[14] Liang Chen and Qidan He. *Shooting the OS X El Capitan Kernel Like a Sniper*. 2016.

[15] Jonathan Corbet. *Preventing kernel-stack leaks*. Mar. 2018. URL: https://lwn.net/Articles/748642/.

[16] Standard Performance Evaluation Corporation. *SPEC CPU 2017*. 2017. URL: https://www.spec.org/cpu2017/.

[17] Ian Cutress. *Spectre and Meltdown in Hardware: Intel Clarifies Whiskey Lake and Amber Lake*. Aug. 2018. URL: https://www.anandtech.com/show/13301/spectre-and-meltdown-in-hardware-intel-clarifies-whiskey-lake-and-amber-lake.

[18] Lizzie Dixon. *Breaking KASLR with perf*. 2017. URL: https://blog.lizzie.io/kaslr-and-perf.html.

[19] Jake Edge. *Kernel address space layout randomization*. 2013. URL: https://lwn.net/Articles/569635/.

[20] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR." In: *MICRO*. 2016.

[21] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2016.

[22] Ulf Frisk. *Windows 10 KASLR Recovery with TSX*. 2016. URL: http://blog.frizk.net/2016/11/windows-10-kaslr-recovery-with-tsx.html.

[23] Thomas Garnier. *Kernel memory randomization and trampoline page tables*. Aug. 2016. URL: https://medium.com/@mxatone/kernel-memory-randomization-and-trampoline-page-tables-9f73827270ab.

[24] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2016).

[25] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. "LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization." In: *RAID*. 2017.

[26] Jason Gionta, William Enck, and Per Larsen. "Preventing kernel code-reuse attacks through disclosure resistant code diversification." In: *Communications and Network Security (CNS)*. 2016.

[27] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. "Out of Control: Overcoming Control-Flow Integrity." In: *S&P*. 2014.

[28] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. 2017.

[29] Daniel Gruss, Dave Hansen, and Brendan Gregg. "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer." In: *USENIX ;login* (2018).

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR." In: *ESSoS*. 2017.

[31] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[32] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *CCS*. 2016.

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[34] Noam Hadad and Jonathan Afek. *Overcoming (some) Spectre browser mitigations*. 2018. URL: `https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/`.

[35] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[36] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *S&P*. 2013.

[37] Intel. *Deep Dive: Intel Analysis of Microarchitectural Data Sampling*. May 2019. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling`.

[38] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[39] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.

[40] Intel. *Intel Analysis of Speculative Execution Side Channels*. July 2018. URL: `https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf`.

[41] Intel. *Performance Monitoring Impact of Intel Transactional Synchronization Extension Memory*. 2019. URL: `https://cdrdv2.intel.com/v1/dl/getContent/604224`.

[42] Intel. *Speculative Execution Side Channel Mitigations*. Revision 3.0. 2018.

[43] ecma international. *ECMAScript 2018 Language Specification*. 2018. URL: `https://www.ecma-international.org/ecma-262/9.0/index.html`.

[44] Alex Ionescu. *Twitter: Windows KASLR*. 2016. URL: `https://twitter.com/aionescu/status/725399988306644992`.

[45] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES." In: *S&P*. 2015.

[46] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES." In: *RAID'14*. 2014.

[47] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks." In: *CCS*. 2018.

[48] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack." In: *SysTEX*. 2017.

[49] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX." In: *CCS*. 2016.

[50] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*. 2016.

[51] Vladimir Kiriansky and Carl Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses." In: *arXiv:1807.03757* (2018).

[52] Amit Klein and Benny Pinkas. "From IP ID to Device ID and KASLR Bypass." In: *USENIX Security*. 2019.

[53] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution.” In: *S&P*. 2019.

[54] Paul C. Kocher. “Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems.” In: *CRYPTO*. 1996.

[55] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “Spectre Returns! Speculation Attacks using the Return Stack Buffer.” In: *WOOT*. 2018.

[56] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. “Hacking in Darkness: Return-oriented Programming against Secure Enclaves.” In: *USENIX Security Symposium*. 2017.

[57] Linux. *Complete virtual memory map with 4-level page tables*. 2019. URL: `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt`.

[58] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices.” In: *USENIX Security Symposium*. 2016.

[59] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space.” In: *USENIX Security Symposium*. 2018.

[60] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical.” In: *S&P*. 2015.

[61] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers.” In: *CCS*. 2018.

[62] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.” In: *NDSS*. 2017.

[63] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. “Spectre is here to stay: An analysis of side-channels and speculative execution.” In: *arXiv:1902.05178* (2019).

[64] Larry McVoy and Carl Staelin. “Lmbench: Portable Tools for Performance Analysis.” In: *USENIX ATC*. 1996.

[65] Andrei Mogage, Rafael Pires, Vlad Crăciun, Pascal Felber, and Emanuel Onica. “Supply chain malware targets SGX: Take care of what you sign (Practical Experience Report).” In: *SRDS*. 2019.

[66] Mozilla. *Index Masking in Firefox.* 2019. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1430051.

[67] Mozilla. *performance.now resolution.* 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[68] Net Applications.com. *Desktop Operating System Market Share.* Aug. 2019. URL: http://www.netmarketshare.com/operating-system-market-share.aspx.

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA.* 2006.

[70] Matthew Panzarino. *Apple releases OS X 10.8 Mountain Lion Developer Preview 2, lists known issues.* Mar. 2012. URL: https://thenextweb.com/apple/2012/03/16/apple-releases-os-x-10-8-mountain-lion-developer-preview-2-to-mac-developers/.

[71] Colin Percival. "Cache missing for fun and profit." In: *BSDCan.* 2005.

[72] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. "kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse." In: *EuroSys.* 2017.

[73] Dan Rosenberg. *kptr_restrict for hiding kernel pointers.* Dec. 2010. URL: https://lwn.net/Articles/420403/.

[74] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load." In: *S&P.* 2019.

[75] Morten Schenk. *Development of a new Windows 10 KASLR Bypass (in One WinDBG Command).* 2019. URL: https://www.offensive-security.com/vulndev/development-of-a-new-windows-10-kaslr-bypass-in-one-windbg-command/.

[76] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *S&P.* 2015.

[77] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs." In: *arXiv:1905.05725* (2019).

[78] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In: *AsiaCCS*. 2018.

[79] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In: *NDSS*. 2018.

[80] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *CCS*. 2019.

[81] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC*. 2017.

[82] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *DIMVA*. 2019.

[83] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *DIMVA*. 2017.

[84] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In: *CCS*. 2007.

[85] Brad Spengler. *KASLR: An Exercise in Cargo Cult Security*. 2013. URL: `https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security.php`.

[86] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels." In: *arXiv:1806.07480* (2018).

[87] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *S&P*. 2013.

[88] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium*. 2018.

[89] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[90] Pepe Vila, Boris Köpf, and Jose Morales. “Theory and Practice of Finding Eviction Sets.” In: *S&P*. 2019.

[91] WebAssembly. *Features to add after the MVP*. 2019. URL: `https://github.com/WebAssembly/design/blob/master/FutureFeatures.md`.

[92] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. “AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves.” In: *ESORICS*. 2016.

[93] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. `https://foreshadowattack.eu/foreshadow-NG.pdf`. 2018.

[94] Zhenyu Wu, Zhang Xu, and Haining Wang. “Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud.” In: *IEEE/ACM Transactions on Networking* (2014).

[95] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. “An exploration of L2 cache covert channels in virtualized environments.” In: *CCSW’11*. 2011.

[96] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.” In: *USENIX Security Symposium*. 2014.

[97] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-Tenant Side-Channel Attacks in PaaS Clouds.” In: *CCS*. 2014.

[98] Peter Zijlstra. *Implement support for TSX Force Abort*. 2019. URL: `https://lkml.org/lkml/2019/3/12/1352`.

7 The Evolution of Transient-Execution Attacks

Publication Data

Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. “The Evolution of Transient-Execution Attacks.” In: *GLSVLSI*. 2020

Contributions

Main author.

The Evolution of Transient-Execution Attacks

Claudio Canella[1], Khaled N. Khasawneh[2], Daniel Gruss[1]

[1] Graz University of Technology, Austria [2] George Mason University, USA

Abstract

Historically, non-architectural state was considered non-observable. Side-channel attacks, in particular on caches, already showed that this is not entirely correct and meta-information, such as the cache state, can be extracted. Transient-execution attacks emerged when multiple groups discovered the exploitability of speculative execution and, simultaneously, the exploitability of deferred permission checks in modern out-of-order processors. These attacks are called transient as they exploit that the processor first executes operations that are then reverted as if they were never executed. However, on the microarchitectural level, these operations and their effects can be observed. While side-channel attacks enable and exploit direct access to meta-data from other security domains, transient-execution attacks enable and exploit direct access to actual data from other security domains. In this paper, we show how the transient-execution landscape evolved since the initial discoveries. We show that the understanding and systematic view of the field has advanced and now facilitate the discovery of new attack variants.

1 Introduction

Modern processors are designed to comply with a defined interface, the instruction-set architecture, allowing to run the same binaries on different processors. Hence, processors can optimize performance and efficiency as long as the behavior is architecturally defined. One optimization that improves performance substantially is caching. Caching is entirely transparent to software, but it introduces a significant speed up for memory accesses and execution by storing data likely to be accessed in the near future in small internal CPU memories. Another optimization is branch prediction. The processor guesses which direction of a branch is taken and thus does not have to wait with the execution of further instructions

The original publication is available at `https://dl.acm.org/doi/abs/10.1145/3386263.3407583`.

until the branch is resolved. If the branch predictor correctly predicted the outcome, a significant performance gain is possible.

From a security perspective, non-architectural state was long considered non-observable as there is no attacker-accessible interface to check what is in the cache. Kocher [23] brought up the idea that caches could be used in side-channel attacks. Such attacks measure the latency of memory accesses to infer what is in the cache, yielding an interface that makes non-architectural state visible to an attacker. However, the information extracted here is only meta-information, *i.e.*, whether the data was retrieved from the cache or main memory. While this enables powerful cryptographic [28] and non-cryptographic [12] attacks, both the security community and the architecture community deemed it most reasonable to design code in a way such that secrets do not influence meta-information, e.g., no secret-dependent memory accesses [3].

In 2018, Spectre [22] and Meltdown [25] were disclosed to the public. They exploit transient execution, the execution of operations which should not be executed, to leave microarchitectural traces. The processor runs into transient execution when a branch is mispredicted, due to lazy exception handling, or other microarchitectural constellations requiring at least a partial pipeline flush or stall. Initially, it was expected that more Spectre variants would be found, but Meltdown was perceived as a one-off vulnerability.

With Foreshadow [39, 41], it became clear that further Meltdown-like effects exist. Foreshadow extended the user-to-kernel attack, to a generic attack able to leak data from any physical memory location. In some sense, it foreshadowed the further development in this field, with many different Meltdown-type attacks discovered by now [7, 31, 33].

While the field of transient-execution attacks has grown ever since its initial discovery, so has the field of potential countermeasures. Concurrent work has analyzed and highlighted different approaches from academia and industry in mitigating Spectre-, Meltdown-, and LVI-type attacks [4].

In this paper, we show how the transient-execution landscape evolved since these initial discoveries. We introduce a 6-phase generalization of transient-execution attacks. We describe the currently known types of Spectre attacks, exploiting microarchitectural mispredictions. In contrast to previous work [6], we then provide a novel systematic view on Meltdown attacks that emphasizes similarities between Meltdown-type attacks rather than differences. We show that the understanding and systematic view of the field has advanced since the 2018 disclosure. We conclude that the discovery of new attack variants increasingly builds on this systematic view.

Outline. Section 2 provides technical background on speculative and out-of-order execution, cache attacks, and store-to-load forwarding. Section 3 introduces transient-execution attacks and structures them in 6 attack phases. Section 4 describes state of the art on Spectre-type attacks, Meltdown-type attacks based on the similarities we identified, and LVI attacks as inverted Meltdown-type attacks. Section 5 concludes our paper.

2 Background

In this section, we provide background on speculative and out-of-order execution, cache attacks in general as well as on store-to-load forwarding.

2.1 Speculative and Out-of-Order Execution

As CPUs need to outperform the previous generation of CPUs, vendors include many different forms of optimizations. Two such optimizations are speculative and out-of-order execution.

Out-of-Order Execution. CPUs parallelize more and more parts of the execution of instructions. In the pipeline, the fetch, decode, execute, and write-back stages handle a multitude of operations simultaneously. This parallelization allows the CPU to execute instructions out-of-order as soon as their operands are available, while instructions that precede them in the instruction stream are still waiting for their operands. Even though the instructions are executed out-of-order, they must still retire (commit) in-order as in the instruction stream to ensure a correct architectural state and enable precise interrupts. This design goes back to Tomasulo's algorithm [38].

Speculative Execution. Speculative execution is an optimization that tries to improve the performance of the non-linear instruction stream of an application. The CPU contains a branch prediction unit (BPU) that tries to predict, based on past behavior, what the most likely direction of control flow is in the instruction stream. The BPU typically consists of several structures, each designed to predict the behavior of different control-flow mechanisms. We refer to Canella et al. [6] for a more detailed description of the individual structures that comprise the BPU.

2.2 Cache Attacks

Cache attacks have become a widely used building block in microarchitectural attacks, especially for transient-execution attacks. Cache attacks exploit the timing difference between accessing data that resides in the cache, e.g., data was used recently, and data in main memory. Hence, due to the cache being closer to the CPU, its latency in responding to a memory request is far lower than a main memory access. While there are a variety of cache attacks such as Flush+Reload [13, 44], Prime+Probe [28, 29], Flush+Flush [10], or Evict+Time [3, 28], all of them share the same three stages.

1. The attacker brings the microarchitecture into a pre-defined state.
2. The attacker lets the victim perform an operation.
3. The attacker infers information on victim behavior based on microarchitectural state changes.

Covert channels are a special case of cache attacks. In such a scenario, the attacker controls both the sender and the receiver, enabling them to bypass restrictions on the architectural level.

2.3 Store-to-Load Forwarding

For store-to-load forwarding, two components are used, namely the memory disambiguation predictor and the store buffer. The idea for store-to-load forwarding is to forward the data from a store that is in the store buffer and has not yet been written back to the cache to a load that uses the same address. There are four different cases that need to be considered for store-to-load forwarding with respect to the load address:

- **True positive match.** A store buffer entry is matched with the same full physical address. In this case, the data is forwarded, which is the correct behavior.
- **True negative match.** No store buffer entry matches, and there is no store buffer entry with the same full physical address. No data is forwarded, which is the correct behavior.
- **False negative match.** No store buffer entry matches, although there is one with the exact same full physical address. The load retrieves stale data, e.g., from the L1 cache, and has to be reissued later on.
- **False positive match.** A store buffer entry is matched, but the full physical address check at the end fails. In this case, data is first transiently forwarded, but the load has to be reissued later on.

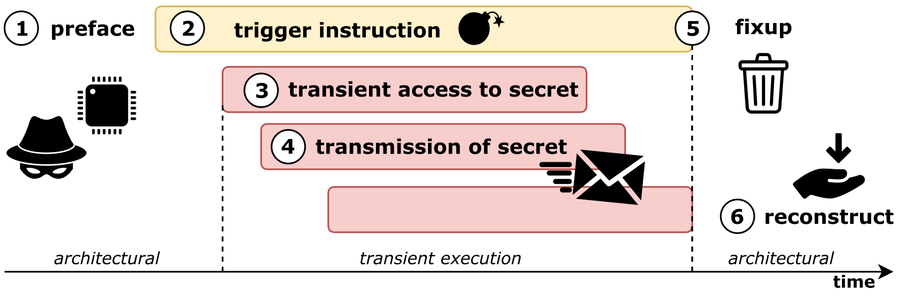


Figure 7.1: High-level overview of a transient-execution attack in 6 phases: (1) preparation, (2) transient-execution trigger, (3) access secret, (4) encode secret, (5) architectural revert, (6) decode secret.

3 Basic Idea of Transient-Execution Attacks

With transient execution, we describe the execution of instructions that are executed but whose results are never committed to the architectural level. We refer to these instructions as transient instructions [6, 22, 25]. Nevertheless, these transient instructions leave traces behind in the microarchitecture by changing its state, e.g., loading data into the cache. The literature has so far identified two different causes of transient execution, namely speculative and out-of-order execution [6, 22, 25]. With speculative execution, it is possible to execute transient instructions due to a wrong prediction of a branch or data dependency. Out-of-order execution can lead to transient execution due to a previous instruction in the pipeline triggering an exception. In that case, the next instructions in the pipeline may still be executed. Both cases have one thing in common: the operations and the architectural state have to be reverted, including flushing the pipeline.

While transient-execution attacks so far have relied on side channels, *i.e.*, predominantly the cache, for the transmission, we note that they themselves are not side channels, contradicting previous literature [14].

Canella et al. [6] proposed 5 distinct phases for a transient-execution attack while Xiong and Szefer [43] propose 3 phases. The phases by Xion and Szefer [43] are similar to the ones by Canella [6] with the three middle phases merged together. We extend the proposal by Canella et al. [6] with one additional distinct Phase 3 that we will describe in the following. Our extended phases are shown in Figure 7.1. Note that each phase may be either performed indirectly by the victim, *i.e.*, attacker provides the necessary trigger to the victim, or directly by the attacker.

Phase 1. This phase is the preparation phase of the attack and consists of two parts. In the first, the attacker prepares the microarchitecture in a way that allows entering transient execution during which the secret data is accessed. This includes ensuring that the transient window, *i.e.*, the time between the execution of the first transient instruction and the last one, is large enough. In the second, the attacker prepares the microarchitectural transmission channel so that the data can be recovered.

Phase 2. The attacker triggers transient execution using a trigger instruction. For Spectre-type attacks, this could be any branch in the victim domain. For Meltdown-type or LVI-type attacks, this could be any form of an aborting instruction, *i.e.*, an instruction that triggers a fault, an assist, or an interrupt. In the case of Spectre-type and LVI-type attacks, the trigger instruction is in the victim domain. For Meltdown-type attacks, it is in the attacker domain.

Phase 3. We split Phase 3 from the proposal by Canella et al. [6] into two phases, Phase 3 and Phase 4. In Phase 3, the transient instructions are executed either in the victim or the attacker domain. Typically, these instructions access data of interest to the attacker, *i.e.*, a secret such as a password. As a final step in this phase, the attacker prepares the data for transmission through a microarchitectural covert channel such as the cache.

Phase 4. This phase is the last in the transient domain. In this phase, the attacker encodes the previously prepared data in the microarchitecture. So far, the covert channel that has been used predominantly in the published literature is the cache [6].

Phase 5. In this phase, the CPU realizes the mistake, *i.e.*, the misprediction or that an exception has been raised. The architectural changes are reverted and the pipeline is flushed. The CPU then continues with the execution of the correct instructions, *i.e.*, a number of instructions may have to be reissued. In the case of Spectre, the CPU executes the correct side of the branch; in Meltdown, it executes the exception handler. Nevertheless, in this phase, the information the attacker wants to obtain is already encoded in the microarchitecture where not all changes are reverted.

Phase 6. In the recovery phase, the attacker extracts the encoded data from the microarchitectural level to the architectural level. As the data is usually encoded in the cache, the attacker uses cache side channels, such as Flush+Reload, to perform this task.

4 Transient-Execution Attacks

In this section, we discuss the known transient-execution attacks in more detail. We first focus on attacks that belong to the Spectre class of attacks, then move on to the Meltdown class of attacks, and finally discuss LVI-type attacks.

4.1 Spectre-type Attacks

Spectre-type attacks so far exclusively rely on exploiting transient execution following a misprediction. Misprediction occurs whenever the CPU predicts to take the wrong path based on past behavior. To make this prediction, the BPU consists of multiple predictors, each designed to predict a different type of branch. The Pattern History Table (PHT) is used to predict the outcome of a direct branch, such as an `if`-statement. Typically, the PHT is indexed by some bits of a virtual address and contains a saturating counter per entry [9]. The Branch Target Buffer (BTB) is used to predict the target of a branch following an indirect branch. For that, the BTB stores the previous jump target for the current address. The Return Stack Buffer (RSB) is used to improve the performance of return statements. Every call stores the return address at the top of the RSB, and each return pops the top of it. The last predictor that has been exploited so far is the memory disambiguator that is involved in store-to-load forwarding. This predictor tries to predict whether a load operation overlaps with a previous store operation [20].

Canella et al. [6] systematically analyzed Spectre-type attacks and how each unit is shared within a processor core. Their analysis showed various potential strategies for mistraining the individual branch predictors. Same-address-space strategies mistrain within the same address space using either the target address or a congruent address that refers to the same entry in the predictors. They also showed that the same two strategies can be exploited in a cross-address-space scenario if the predictor is shared between hyperthreads. Their proposed classification also indicates for each Spectre variant which predictor is exploited.

Spectre-type attacks require different types of gadgets, *i.e.*, code sequences, to execute. These types, as well as different detection techniques, have been analyzed by Canella et al. [6]; hence we refer the reader to their paper for further information.

Spectre-PHT. Spectre-PHT was one of the first two Spectre variants discovered by Kocher et al. [22]. Here, the PHT is exploited by mistraining it repeatedly to take a certain path following, e.g., a bounds check.

Afterward, the attacker can supply a value that would cause the CPU to execute the other path of the bounds check, but due to the prediction, it still executes the mistrained path. Schwarz et al. [34] have demonstrated that such an attack is also possible via the network.

Spectre-BTB. Spectre-BTB is the second variant discovered by Kocher et al. [22]. As the name indicates, it exploits the behavior of the BTB to force the CPU to predict a wrong jump target. With Spectre-BTB, the attacker can redirect control-flow to virtually any address within the victim domain. Similarly to return-oriented programming (ROP) [35], the attacker can chain multiple jump targets together to achieve arbitrary transient execution as far as the transient window allows. Straight-line speculation vulnerabilities [2] exploiting exception generating instructions, unconditional direct and indirect branches also fall into this category.

Spectre-RSB. Horn [15] and Kocher et al. [22] first mentioned the possibility of exploiting the RSB, but the first attack was demonstrated by Koruyeh et al. [24] and Maisuradze and Rossow [26]. In a Spectre-RSB attack, the attacker poisons the RSB with malicious return destinations that the victim program pops from the stack upon a return, redirecting control flow to a leak gadget. Straight-line speculation vulnerabilities [2] exploiting exception or function returns fall into this category as well.

Spectre-STL. Horn [16] demonstrated that loads can transiently obtain stale values. The reason for that is that the memory disambiguator predicts that a current load does not depend on any previous store. The load operation is then scheduled before the preceding store and obtains the old value from the cache. In its essence, Spectre-STL exploits a false negative match in the store buffer.

4.2 Meltdown-type Attacks

Canella et al. [6] proposed a canonical naming scheme for Meltdown-type attacks. In the first layer, they distinguish the type of the fault or the assist that causes transient execution. In the following layers, they distinguish different reasons for the fault or assist and from which buffer they leak from. Consequently, every Meltdown-type attack has a precise canonical name.

In this paper, we want to highlight the similarities between the different Meltdown-type attacks. Based on our analysis of the attacks, we divide Meltdown-type attacks into the following three groups based on their microarchitectural behavior:

- **Deferred Permission Check.** Meltdown-type attacks in this category show architecturally correct behavior but with a lack of permission checks, e.g., Meltdown-US [25]. These Meltdown-type attacks perform operations that, from the CPU's perspective, would be valid and meaningful at a different permission level. For instance, attempting to access a kernel address is valid and meaningful for kernel code.
- **Incorrect Use of Intermediate Values.** Some Meltdown-type attacks exploit the usage of intermediate values to retrieve data, e.g., Foreshadow [39, 41]. The behavior exploited in these attacks is always either not valid or not meaningful, regardless of the permission level. For instance, the architecture defines that a non-present page-table entry may contain any data. Interpreting this data, e.g., as a physical address, is always incorrect.
- **Use-After-Free.** Newer Meltdown-type attacks show behavior that is similar to use-after-free vulnerabilities, causing the usage of stale values, e.g., ZombieLoad [33], RIDL [31], Fallout [7].

Deferred Permission Check

The idea behind Meltdown-type attacks in this category is that data is forwarded despite the permission check failing in parallel. For each load μOP that is added to the re-order buffer, a new entry in the load buffer is reserved. When the load μOP is scheduled, the load buffer is updated with information about the new load, *i.e.*, register number and virtual address information. Simultaneous to the lookup in the memory hierarchy, the TLB is checked to retrieve the corresponding physical address. Once the page-table information has been retrieved, the permissions are checked. As the permission check fails, the processor raises a fault but still updates the physical page number in the load buffer. At this point in time, the data is ready to be forwarded to the register and can be used in subsequent transient instructions.

Attacks. Meltdown-US, the original Meltdown [25], exploits faults following a check on the user-space-accessible bit and allows leaking kernel data. Canella et al. [5] demonstrate such an attack on 32-bit Linux systems using JavaScript and also provide the first detailed analysis of Intel's hardware Meltdown mitigation. Canella et al. [6] used Meltdown-PK to bypass read and write isolation provided by Intel's memory protection keys. Furthermore, they showed that hardware instructions for bounds checking provided on both IA-32 and IA-64 can be bypassed with Meltdown-BND and Meltdown-MPX. Meltdown-BND was also the first demonstration of Meltdown-type effects on AMD processors. Other attacks demonstrated that data can be leaked from registers. The first, Meltdown-GP [1, 17, 19] can be used to leak data from privileged system registers. With the second, Stecklina and Prescher [36] showed that registers that can be switched between kernel and user mode can be exploited. In their paper, they targeted FPU and SIMD registers. Xiao et al. [42] developed a framework to study transient-execution attacks, discovering a new Meltdown variant on AMD processors that bypasses segment limits.

Two cases of store-to-load forwarding, namely the true positive and the true negative match, also belong in this attack category. For a true positive match, Kiriansky and Waldspurger [21] observed that the writable bit is transiently ignored when performing store-to-load forwarding. Schwarz et al. [32] and Canella et al. [5, 6] demonstrated that this also happens for other checks, e.g., the user-space-accessible bit. For a true negative match, Schwarz et al. [32] and Canella et al. [5, 6] showed that the information about the negative case can be abused in combination with the true positive case for distinguishing valid addresses from invalid ones, enabling KASLR breaks.

Incorrect Use of Intermediate Values

This category mainly consists of Meltdown-P attacks [6, 39, 41]. Meltdown-P attacks follow the same behavior as a Meltdown-US attack, but instead of exploiting the user-accessible bit it exploits the present bit. At the point where the TLB is queried for the valid translation, the lookup fails as the page is indicated as not being present. To perform the page-table walk, a microcode assist is issued [33]. As the attacker runs within a virtual machine, two such walks have to be issued as the guest virtual address is first translated into a guest physical which is then translated into a host physical address. When the information of the guest page table is checked, none of the information is valid as the present bit is not set, causing the processor to raise a fault. Nevertheless, the physical address field in the load buffer is filled out with the information from the page table, *i.e.*, the guest physical address. Similar to Meltdown-US, the data is ready to be forwarded to the register, and the processor matches the guest's physical address to the cache line tag of the retrieved data. This allows an attacker to read arbitrary host physical memory by exploiting non-present page-table entries and a transient access to them.

Attacks. Van Bulck et al. [39] have demonstrated this effect in the context of transiently forwarding data from SGX-protected cache lines. Weisse et al. [41] have extended their work by demonstrating the effect in a non-SGX environment, such as the aforementioned virtual machine scenario.

The two remaining cases of store-to-load forwarding, namely the false negative and the false positive match, are also in this category. For a false negative match, Cauligi et al. [8] describe a theoretical attack that combines Meltdown- and Spectre-type effects. In such an attack, the memory disambiguator predicts a dependency, allowing the load to continue. The store buffer then only finds a partial match and returns the incorrectly matched data. Canella et al. [7] exploited a false-positive match to read recent writes from SGX enclaves or the kernel.

Use-After-Free

Use-after-free vulnerabilities are not only common in software [37]. We discuss the basic idea behind attacks in this category based on a ZombieLoad v1 attack. While the steps are the same as in the previous attacks, the suspected reason for the erroneous matching of the secret data is the stale physical address from the load buffer. In a ZombieLoad v1 attack, the store buffer, line-fill buffer, L1 data cache are looked up with the virtual address. Due to a cache line conflict, the L1 data cache lookup fails, leading to an abort and a subsequent re-issuing of the load. Nevertheless, the current load, a zombie load, continues and returns data. The stale physical page number (*i.e.*, a use after free) is used to match the tag of the previously retrieved data. If they match, the data is forwarded to the register and can be leaked by subsequent instructions.

Attacks. The first description of such an attack was given by Lipp et al. [25] when they demonstrated a Meltdown-US attack on uncached and uncacheable memory. They showed that leaking from uncacheable memory is only possible if an architectural access to the same memory location occurs, *i.e.*, a hyperthread performs a legal load. According to Lipp et al. [11, 25], the leakage originates in the line-fill buffer. Subsequent publications further investigate this buffer as a leakage source [18, 27, 30, 31, 33]. On processors that do not leak from L1 directly anymore, Van Schaik et al. [31] and Schwarz et al. [33] showed that there is still leakage possible via so called microarchitectural data sampling (MDS) attacks. For instance, Schwarz et al. [33] demonstrated different variants to leak data from the line-fill buffer due to aborted loads continuing their

Attack	Requirement
LVI-US	victim repeatedly accesses a kernel address
LVI-RW	illegal write to read-only memory
LVI-PK	illegal access to a PKU-protected memory location
LVI-MPX, LVI-BND	illegal out-of-bounds access
LVI-GP	memory access triggering a general protection fault
LVI-NM	FPU register access with FPU being unavailable
LVI on segment limits	memory access outside segment limit

Table 7.1: The different LVI attacks relying on a deferred permission check and what event they require to enter transient execution.

execution. Additionally, they showed that one of these variants even affects Intel Cascade Lake CPUs that contain fixes for Meltdown and MDS attacks. Finally, Schwarz et al. [33] also demonstrated that the initial mitigation of using the `verw` instruction to overwrite buffers does not fully work.

4.3 LVI-Type Attacks

Spectre turned previously known attacks on branch prediction around, not exploiting the branch prediction to retrieve data from a previous execution, but instead injecting a transient control-flow change into a victim to leak data. LVI (Load Value Injection) [40] does the same with Meltdown, *i.e.*, instead of exploiting Meltdown-type effects to retrieve data from a victim domain, it uses them to transiently inject data into the victim. Contrary to Meltdown-type attacks, LVI attacks cannot always control when a fault and other conditions occur as they occur within the victim domain. In that sense, LVI is more similar to Spectre-type attacks where certain gadgets within the victim are required. For LVI, the complexity of gadgets is much lower as a single memory access can already be a LVI gadget. The same is true for a single indirect call, jump, or return.

In principle, an LVI attack attempts to obtain data from the victim domain that the victim can legally access. All three of the previously identified types of Meltdown-type effects can be used for LVI attacks, *i.e.*, deferred permission checks, incorrect use of intermediate values, and use-after-free. While all three types can be exploited, the deferred permission check type of attacks is more realistic in the threat model of SGX because they require repeated illegal behavior of the victim domain. Table 7.1 shows the different attack types relying on the deferred permission check and what they require to trigger the attack.

In a normal scenario, faults are handled by the operating system, preventing them from happening again. Hence, a more realistic scenario for an LVI attack in a normal context is to use misprediction to suppress the fault. Using misspeculation, the attacker can suppress the fault as previous work has already outlined [6, 22, 25], but in this case, a Spectre-type attack might already be sufficient to extract the targeted data from the victim. In this scenario, LVI may only be of use if the Spectre-type attack does not allow enough control of the victim, which LVI might be able to supply.

In the SGX scenario, one type of attack that is particularly relevant is the incorrect use of intermediate values. By inverting Meltdown-P to obtain LVI-P, the attacker can transiently inject data from a chosen physical address into the victim's execution by unmapping a page and loading data into the L1 data cache. For the LVI-NULL case, the attacker can inject a NULL value instead, e.g., on processors with first-generation mitigations. Both cases have been demonstrated by Van Bulck et al. [40]. For LVI-P-NULL, the attacker transiently injects a NULL value when accessing the stack to read the return address from virtual address 0, redirecting control flow to the location that is contained at this location. In LVI-P-L1D, fake return addresses are injected to re-direct control flow to an attacker chosen location.

For non-SGX-based LVI attacks, use-after-free type of attacks have been shown to be the most effective [40]. In these scenarios, microarchitectural assists have been used to trigger the use-after-free situation in the line-fill buffer. The store buffer false positive match is one case that can occur easily, but the requirements on gadgets are much higher than in the LFB case. For this attack, the attacker has to place a matching store in the store buffer, which is statically partitioned. Hence, the attacker has to inject the store on the same hyperthread before a context switch or via a gadget within the victim that stores to the attacker-tweakable address before reading from an unrelated, but partially matching, address. If the store is successfully injected, the store buffer false-positively matches a store, *i.e.*, without a full physical address match. As in the case of ZombieLoad, the load in the victim continues but is ultimately squashed. Nevertheless, the wrong data is transiently forwarded to the dependent operations.

5 Conclusion

For a long time, the non-architectural state was considered to be unobservable. Side-channel attacks have demonstrated that this is not entirely true as meta-information can be extracted from the microarchitectural state. To optimize the performance of modern processors, vendors included various optimizations such as speculative and out-of-order execution. Transient-execution attacks exploit these features as their effect on the microarchitectural state is not reverted even in the case that the result was unrolled. Contrary to side-channel attacks, transient-execution attacks allow direct access to the actual data from another security domain, not just meta-data. In this paper, we showed how the transient-execution landscape has evolved since the initial discoveries. We showed that the understanding and systematic view of the field has advanced and now facilitate the discovery of new attack variants.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] ARM. *Cache Speculation Side-channels.* Version 2.4. 2018.

[2] ARM. *Straight-line Speculation.* Version 1.0. 2020.

[3] Daniel J. Bernstein. *Cache-Timing Attacks on AES.* Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[4] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. "Evolution of Defenses against Transient-Execution Attacks." In: *GLSVLSI.* 2020.

[5] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. "KASLR: Break It, Fix It, Repeat." In: *AsiaCCS.* 2020.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium.* Extended classification tree and PoCs at https://transient.fail/. 2019.

[7] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-resistant CPUs." In: *CCS.* 2019.

[8] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. "Towards Constant-Time Foundations for the New Spectre Era." In: *arXiv:1910.01755* (2019).

[9] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.* 2016.

[10] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA.* 2016.

[11] Daniel Gruss, Michael Schwarz, and Moritz Lipp. "Meltdown: Basics, Details, Consequences." In: *BlackHat USA.* 2018.

[12] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium.* 2015.

[13] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice.” In: *S&P*. 2011.

[14] Mark D Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L Hennessy. “On the Spectre and Meltdown Processor Security Vulnerabilities.” In: *IEEE Micro* 39.2 (2019), pp. 9–19.

[15] Jann Horn. *Reading privileged memory with a side-channel.* 2018.

[16] Jann Horn. *speculative execution, variant 4: speculative store bypass.* 2018.

[17] Intel. *Intel Analysis of Speculative Execution Side Channels.* Revision 4.0. 2018.

[18] Intel. *Intel-SA-00233 Microarchitectural Data Sampling Advisory.* 2019. URL: `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html`.

[19] Intel. *Q2 2018 Speculative Execution Side Channel Update.* 2018.

[20] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks.” In: *USENIX Security Symposium.* 2019.

[21] Vladimir Kiriansky and Carl Waldspurger. “Speculative Buffer Overflows: Attacks and Defenses.” In: *arXiv:1807.03757* (2018).

[22] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution.” In: *S&P*. 2019.

[23] Paul C. Kocher. “Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems.” In: *CRYPTO*. 1996.

[24] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “Spectre Returns! Speculation Attacks using the Return Stack Buffer.” In: *WOOT*. 2018.

[25] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space.” In: *USENIX Security Symposium.* 2018.

[26] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers.” In: *CCS*. 2018.

[27] Giorgi Maisuradze. “Assessing the Security of Hardware-Assisted Isolation Techniques.” PhD thesis. Saarland University, 2019.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA*. 2006.

[29] Colin Percival. "Cache missing for fun and profit." In: *BSDCan*. 2005.

[30] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real." In: *S&P*. 2021.

[31] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load." In: *S&P*. 2019.

[32] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs." In: *arXiv:1905.05725* (2019).

[33] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *CCS*. 2019.

[34] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *ESORICS*. 2019.

[35] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In: *CCS*. 2007.

[36] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels." In: *arXiv:1806.07480* (2018).

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *S&P*. 2013.

[38] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units." In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[39] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium*. 2018.

[40] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *S&P*. 2020.

[41] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution.* 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf`.

[42] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. "SPEECH-MINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities." In: *NDSS*. 2020.

[43] Wenjie Xiong and Jakub Szefer. "Survey of Transient Execution Attacks." In: *arXiv:2005.13435* (2020).

[44] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

8
Evolution of Defenses against Transient-Execution Attacks

Publication Data

Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. “Evolution of Defenses against Transient-Execution Attacks.” In: *GLSVLSI*. 2020

Contributions

Main author.

Evolution of Defenses against Transient-Execution Attacks

Claudio Canella[1], Sai Manoj Pudukotai Dinakarrao[2], Daniel Gruss[1], Khaled N. Khasawneh[2]

[1] Graz University of Technology, Austria [2] George Mason University, USA

Abstract

Transient-execution attacks, such as Meltdown and Spectre, exploit performance optimizations in modern CPUs to enable unauthorized access to data across protection boundaries. Against these attacks, we have noticed a rapid growth of deployed and proposed countermeasures. In this paper, we show the evolution of countermeasures against transient-execution attacks by both industry and academia since the initial discoveries of the attacks. We show that despite the advances in the understanding and systematic view of the field, the proposed and deployed defenses are limited.

1 Introduction

Transient execution enables unauthorized access to data across security protection boundaries. Transient execution refers to the execution of instructions that will eventually get squashed, *i.e.*, their execution results will not be committed to the architectural state. Nonetheless, transient execution can leave a trace in the microarchitectural state, e.g., the cache state. Therefore, transient-execution attacks utilize the execution of transient instructions to access secret data, e.g., a password, and leave a secret-dependent trace in the microarchitectural state that can be recovered later using non-transient execution. These attacks can be classified into three main classes, namely Meltdown, Spectre, and Load Value Injection (LVI), based on the nature of the transient execution and the attack direction. Spectre is based on misprediction in the victim domain, Meltdown is based on faults and assists in the attacker domain, and LVI is based on faults and assists in the victim domain.

The initial discovery of transient-execution attacks, *i.e.*, Meltdown and Spectre, became one of the most complex and largest industry-wide

The original publication is available at `https://dl.acm.org/doi/10.1145/3386263.3407584`.

embargos as processors from various manufacturers turned out to be affected. As a result, many attacks variants were discovered, but more noticeable is the proliferation of countermeasures from both industry and academia. Given the large number and the rapid growth of both adopted and proposed countermeasures, a systematic view is required to understand the scope of current defenses and facilitate the evaluation of future defenses.

In this paper, we show how the landscape of countermeasures against transient-execution attacks evolved since the initial discoveries of the attacks. We build our systematization based on a concurrent 6-phase generalization of transient-execution attacks [16]. We systematically describe hardware- and software-based countermeasure advances from both industry and academia. Beyond previous work [18, 99], our systematic view does not only cover Spectre and Meltdown defenses but also LVI defenses. We show that despite the advances in the understanding and systematic view of the field, the proposed and deployed defenses are limited.

Outline. First, we briefly discuss background in Section 2. The paper then gives a systematic overview of countermeasures for Spectre (Section 3), Meltdown (Section 4), and LVI (Section 5). We conclude in Section 6.

2 Background

Out-of-order and speculative execution. To increase performance, modern CPUs rely on features like speculative and out-of-order execution. With speculative execution, CPUs try to predict the outcome of a potential control-flow change to start the execution of the likely path instead of stalling. For that, the CPU provides various predictors that together comprise the Branch Prediction Unit (BPU) [18]. Out-of-order execution allows executing later instructions that are ready to be executed due to the operands being available in advance but still requires to retire them in order. Recently, these optimizations have resulted in various transient-execution attacks [18, 55, 60].

Transient-execution attacks. Transient-execution attacks exploit modern CPUs performance optimizations to enable unauthorized access to data across protection boundaries. According to a concurrent generalization of transient-execution attacks, these attacks consist of 6 distinct phases [16]: *Phase 1 (preparation):* preparing the micro-architecture to enter transient execution, *Phase 2 (misspeculation):* triggering transient execution using a trigger instruction, *Phase 3 (access):* accessing data of interest, *Phase 4 (encoding):* encoding data of interest in the microarchitecture state, *Phase 5 (leakage):* end of transient window, i.e., the architectural changes are reverted and the pipeline is flushed, and *Phase 6 (decoding):* decoding the microarchitectural state to the architectural state.

3 Spectre Countermeasures

A countermeasure can try to break any phase of a Spectre attack [16]: preparation, misspeculation, access, encoding, leakage, decoding. However, targeting different phases has different effects on security. As the following discussion also shows, mitigating all Spectre attacks in practice likely will remain an open problem in the foreseeable future [62].

3.1 Preparation Prevention (Phase 1)

Phase 1 prepares the microarchitecture, e.g., the cache or branch predictors, for the attack. Defenses targeting this phase usually do not prevent this step entirely but only eliminate the attacker's influence on the victim domain. However, some variants do not require any preparation or run in-place, making it hard to distinguish malicious training from benign execution.

Industry

To prevent mistraining, the industry, e.g., Intel and AMD, extended ISAs with a mechanism for controlling indirect branches [4, 47]. Indirect Branch Restricted Speculation (IBRS) prevents unprivileged code from influencing the prediction of privileged code. Single Thread Indirect Branch Prediction (STIBP) restricts the sharing of branch prediction mechanisms across hyperthreads. The Indirect Branch Predictor Barrier (IBPB) prevents code that executes before it from affecting the prediction of code following it. Some ARM CPUs implement specific controls that invalidate the branch predictor, which should be used during context switches [8]. Linux enabled those by default [52].

For Spectre-STL, ARM introduced new barrier instructions and control registers to prevent the re-ordering of loads and stores [8]. Likewise, Intel [47] and AMD [3] provide Speculative Store Bypass Disable (SSBD) microcode updates.

Academia

In contrast to industry, academia proposed more fundamental architecture and microarchitecture changes. Vougioukas et al. [93] use per-context buffers for branch predictor state to improve performance after branch predictor flushes. Instead of flushing, Zhao et al. [102] randomize the prediction based on the running context. Both proposals maintain performance within a process across context switches. However, in-place same-domain attacks are unaffected by these designs, and the approach by Zhao et al. [102] may allow cross-domain and out-of-place attacks by reverse-engineering the randomization.

3.2 Misspeculation Prevention (Phase 2)

Entirely disabling speculation seems easy, but the performance loss is prohibitive [55, 87]. Hence, more realistic solutions in this phase only selectively disable or stop speculative execution.

Industry

CPU vendors designed solutions using serializing instructions (*lfence*), stopping speculation at security-critical branches. Unfortunately, these branches have to be identified and essentially annotated on all layers.
Software-based defenses. Google proposed *retpoline* [89], a code sequence replacing indirect branches with return instructions, to prevent branch poisoning. Intel proposed *randpoline* [14] as a more efficient alternative. Due to its probabilistic nature, randpoline does not fully mitigate Spectre-BTB but only reduces success and leakage rates of attacks. Linux and Windows use retpoline on affected machines by default [24, 45].
Hardware-based defenses. Both Intel and AMD described fencing-based solutions [4, 46]. However, they also both introduced new architectural features to constrain speculative execution on the microarchitectural level including instructions for synchronization barriers for data (`DSB`) and instructions (`ISB`), broader speculation barriers (`CSDB`) [8], new registers to restrict speculative execution and instructions to restrain control-flow (`cfp`) and data value (`dvp`) prediction, and cache prefetches (`cpp`) [7]. Even more broadly, both Intel (with `serialize`) and ARMv8.5-A [7] (with `sb`) introduced generic speculative execution barriers.

On future CPUs with Control-flow Enforcement Technology (CET) capabilities, retpoline might trigger false positives in the CET defenses [45]. Therefore, these CPUs implement *enhanced IBRS*, a hardware defense for

Spectre-BTB [45]. Intel [45] also provided a microcode update against Spectre-RSB to stop speculation. However, on Skylake and newer architectures, the RSB may fall back to the BTB, re-enabling Spectre-BTB attacks via return instructions. To prevent this, the RSB is stuffed with the address of a benign gadget when entering the kernel [45].

Academia

Academia helped identifying the limitations of the deployed serializing countermeasures [82]. Furthermore, they proposed techniques to reduce the overhead of such defenses.

Software-based defenses. Schwarz et al. [82] showed that *lfence* instructions only stop execution units from running subsequent operations. Thus, fetch and decode still work, potentially leaking data through the power-up of AVX functional units, the TLB, or the instruction cache. Furthermore, performance-wise, serializing every branch can be worse than using a processor without branch prediction in the first place [44]. Shen et al. [83] split code into small blocks and insert fences between the entry point and a potentially leaking memory access to mitigate Spectre-BTB and Spectre-RSB. However, an attacker could still jump unaligned into a code block, *i.e.*, directly to the memory access.

Instead of using *lfence*, Oleksenko et al. [68] propose to introduce data dependencies between branch condition operands and operations following the branch, stalling the execution of dependent instructions. Unfortunately, due to compiler re-ordering, this proposal is limited in its effectiveness.

As an alternative to retpoline and randpoline, Amit et al. [5] designed JumpSwitches, which add a shortcut path for indirect branches with a direct branch for the most likely target.

Hardware-based defenses. Vassena et al. [92] proposed to annotate variables and insert `lfences` in code paths where such variables may be leaked. To reduce the high cost of adding fences, Taram et al. [88] propose a hardware-based technique to dynamically insert fences before potentially leaking loads. Koruyeh et al. [57] argue that Spectre-BTB and Spectre-RSB usually leave the defined control-flow graph. Hence, they propose SpecCFI to repurpose control-flow integrity (CFI) to prevent speculative diversion from the control-flow graph. Capability systems may also contribute to Spectre mitigations [96].

Several designs introduce buffer flushes or hardware partitioning to isolate different domains (e.g., security enclaves) [13, 31, 69]. However, a limitation to the flushing of caches and buffers upon domain switches is that it is difficult to ensure no microarchitectural state persists. A similar argument was made for the Raspberry PI 3 [90]. However, speculative fetches may leave microarchitectural traces sufficient for an attack [9].

3.3 Data Access Prevention (Phase 3)

Preventing access to certain data during speculative execution is a promising approach to fully mitigate Spectre attacks. Solutions in this phase focus on secrets in memory. None of the solutions presented for this phase protect against Spectre attacks on data in registers.

Industry

Mainly, software-based defenses against data access were adopted by the industry. With process isolation, Google presented the first defense for this phase [22, 73]. Leaking secrets from other contexts is mitigated unless the attacker can utilize Meltdown to bypass process isolation permission checks.

Sanitizing values used in speculation can affect *phase 3* and *phase 4* as memory locations may be inaccessible. The idea of Speculative Load Hardening (SLH) [19] is to check loads using branchless code to ensure that it is executing along a valid control-flow path. One prerequisite for this approach is that the hardware enables branchless and unpredicted conditional updates of register values. Both LLVM and GCC support SLH today and provide a builtin function to either emit a speculation barrier or return a safe value if the instruction is transient [29].

WebKit employs two techniques to limit access to secret data [72]. First, bound checks are replaced with index masking, thus, only introducing a maximum range for the out-of-bounds violation. Second, a pseudo-random *poison value* protects pointers from misuse. Using this approach, an attacker would first have to learn the *poison value* to use it. Furthermore, mispredictions on type checks result in the wrong type being used for the pointer.

Academia

Academia proposed software and hardware defenses, including utilizing existing hardware technologies, e.g., Memory Protection Extensions (MPX) and Memory Protection Keys (MPK).

Software-based defenses. Narayan et al. [66] implemented a sandboxing framework for Firefox that supports process-based isolation. Furthermore, Ojogbo et al. [67] used bitmasks to arithmetically guarantee that any speculatively computed index is in bounds. Dong et al. [27] used Intel MPX for this purpose.

As a probabilistic countermeasure, Sianipar et al. [84] constantly move secret data around in virtual and physical memory. However, this only reduces the leakage rate. In contrast, many deterministic proposals also target this attack phase. Palit et al. [70] use a compiler extension that keeps annotated secret data encrypted in memory most of the time. The secret key is stored in a register. Hence, the attack surface is significantly reduced. Kiriansky and Waldspurger [54] propose to restrict access to sensitive data by using protection keys like Intel MPK technology [43]. However, as an attacker could use Spectre to disable MPK using the `wrpkru` instruction, they propose a microcode update for this instruction to include an `lfence`. Nonetheless, an attacker can still access the data if the system is susceptible to Meltdown-PK [18]. Jenkins et al. [48] propose to use ELFbac [10] or Intel MPK against Spectre attacks.

Hardware-based defenses. Schwarz et al. [79] propose multiple defenses against Spectre that all rely on the annotation of secrets. The compiler groups secret variables onto pages and marks these pages as secure. For commodity systems, they then suggest a technique called ConTExT-light [79], which uses uncacheable memory for secrets, making them inaccessible during speculative execution. Kiriansky et al. [53] propose to securely partition the cache across its ways, with protection domains that isolate on a cache hit, cache miss, and metadata level. However, this requires the correct management of these domains in software.

3.4 Data Encoding Prevention (Phase 4)

Kocher et al. [55] proposed to track data loaded during transient execution and prevent its use in subsequent operations. Several academic works propose new processor designs similar to this idea. There is still no industry solution that targets this phase.

Academia

NDA [97] identifies potentially leaking instructions and defers their execution if they depend on a previous, not yet retired, operation. Yu et al. [101] taint data that has not yet been committed and uses light-weight taint tracking to delay instructions that use such tainted inputs. Cabodi et al. [15] use a similar approach and verify it using model checking. Barber et al. [11] defer the wake up of dependent load instructions from when the load instruction it depends on is retired instead of when it is dispatched. Other works [32, 79] propose to annotate secrets and, thus, only track and protect secrets in registers and the cache.

3.5 Leakage Prevention (Phase 5)

Several solutions propose to speculate as usual but to either store results in new buffers or to completely remove the microarchitectural traces. Many of the proposals only focus on memory accesses and the cache. While effective against simple attacks, more sophisticated attacks may remain unaffected [12].

Academia

Several proposed defenses introduce shadow hardware structures for transient execution [1, 34, 51, 63, 77, 100] to squash microarchitectural state changes upon a wrong prediction. Lowe-Power et al. [61] and Saileshwar et al. [76] propose to undo modifications to the microarchitectural state after misspeculation. Li et al. [59] design a solution that targets specifically the Flush+Reload covert channel, which spreads different values to different pages and block speculative instructions that may lead to accesses to different pages. Rockicki [74] also explored a similar direction for in-order processors that use dynamic binary translation optimizations for performance. Sakalis et al. [78] propose to delay L1 misses until they are certain to be committed. Nonetheless, these proposals are vulnerable against side channels other than caches, e.g., DRAM buffers [71], or execution-unit congestion [2, 12].

3.6 Data Decoding Prevention (Phase 6)

Preventing covert channels is most likely infeasible as long as any shared resource remains. Still, several works propose to mitigate or detect Spectre by breaking or detecting the covert channel.

Industry

Accurate timers are a common, but not crucial, building block of covert channels to distinguish microarchitectural states. Hence, to mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript [21, 65, 72, 94]. However, custom timers can always be constructed [81] and, thus, further mitigations are required [80]. After initially disabling `SharedArrayBuffers` in response to Meltdown and Spectre [21], they have been re-enabled with the introduction of site isolation [85]. This is in line with an older research direction of randomizing or reducing the resolution of timing measurements for security [39].

Academia

Several works propose to detect the cache covert channel used in Spectre attacks and stopping the corresponding process. Most solutions proposed so far use hardware performance counters for this purpose [26], while Sabbagh et al. [75] use memory access traces, and Austin et al. [38] use the cyclic interference property of contention-based cache leakage. However, several works show that it is trivial to evade detection [25, 49, 50, 58]. It is important to note that these proposals only consider cache covert channels.

Ge et al. [33] temporarily reduce the timer resolution whenever the cache flush interface is used. Wang et al. [95] explore varying the processor frequency to hinder native cache attacks. To alleviate the performance and energy impact, they introduce value prediction. However, value prediction is not inherently secure against Spectre attacks, and transiently diverting the control-flow of a victim by inducing a false value via value prediction effectively provides the attacker with the same capabilities. Chen et al. [20] propose to mitigate transient-execution attacks on SGX by preventing interruption of enclaves. However, an attacker does not necessarily have to interrupt an enclave to mount an attack.

4 Meltdown Countermeasures

Meltdown attacks exploit deliberate incorrect behavior of the hardware during transient execution. While this may have been assumed secure in the past, it must be considered a hardware bug today. We first discuss applying Spectre-focused defenses to Meltdown attacks followed by Meltdown-focused defenses.

The fundamental difference between Spectre and Meltdown type attacks is based on the transient execution trigger, prediction or fault based, respectively. Spectre-focused defenses that target *Phases 1*, *2*, and *3* cannot mitigate Meltdown attacks, as the attack runs entirely in the attacker domain. *Phase 4* defenses could be used with an additional performance cost [11, 15, 32, 79, 97, 101]. For these defenses, it is important to not just focus on cache accesses to guarantee mitigation of Meltdown attacks but more broadly prevent operations from using non-architectural and potentially secret data. *Phase 5* defenses could be used to prevent or unroll transient execution microarchitectural effects [1, 12, 34, 38, 51, 59, 61, 74, 76–78, 100]. However, mitigating the cache covert channel is not sufficient to mitigate Meltdown attacks. *Phase 6* defenses could be used to break or slow down [21, 33, 65, 72, 81, 94, 95], or detect [28, 38, 58, 75] the covert channel. However, currently, none of these defenses can guarantee the absence of Meltdown-usable covert channels.

4.1 Meltdown-Focused Defenses

Industry

As Meltdown attacks are considered to be a hardware bug, newer CPUs contain patches. For instance, newer Intel CPUs contain fixes for Meltdown-US, which have been reverse-engineered by Canella et al. [17]. They show that instead of returning data, access to privileged memory now returns 0.

For SGX, Intel proposes to either store secrets in uncacheable memory or to flush the L1 data cache when switching protection domains. Hypervisors similarly flush the L1 upon context switches between untrusted virtual machine threads. To prevent attacks from a VM running on a hyperthread, hypervisors implement variants of gang scheduling [41, 64]. SGX takes the hyperthreading status into account for attestation for the same reason. System Management Mode (SMM) rendezvous logical cores and flushes the L1 upon context switches.

Meltdown-GP, *i.e.*, transient reads of system registers, has been fixed via a microcode update [44]. Newer ARM CPUs are also not vulnerable to Meltdown-GP, whereas older ones can be protected via software workarounds [8]. Meltdown-NM (Lazy-FP) [86], which exploited the lazy switching of floating-point unit (FPU) registers, is mitigated by disabling lazy switching.

Academia

The first software-based defense against Meltdown type attacks was KAISER [36, 37], which removes the kernel mapping while running in user space. Unfortunately, on x86, some privileged memory locations must always be mapped in user space, and thus, can still be attacked [17].

KAISER was merged into Linux as kernel page-table isolation (KPTI) [23]. Other operating systems have received similar patches [35]. LAZARUS [6] pursues a similar idea but uses unmapping and re-mapping of pages upon context switching, which is problematic in multi-threaded applications.

Hua et al. [40] propose EPTI (Extended Page Table Isolation), a variant of KPTI relying on extended page tables. As there is hardware support for EPT switching and TLB entries from different EPTs are tagged, e.g., with VM process IDs (VPIDs), the performance loss is not as severe as with KPTI. However, as this approach uses extended page tables, it leaves the system vulnerable to Meltdown-P. MemoryRanger [56] isolates drivers, kernel and user space into separate address spaces using EPTs.

To mitigate Meltdown-P (Foreshadow) on commodity systems, operating systems now sanitize physical page-number fields of unmapped page-table entries [41, 98] by setting the physical page-number field to values that would refer to non-existent physical memory.

Finally, academic research shows how formal verification could more generically prevent Meltdown bugs [15, 30].

5 LVI Countermeasures

LVI attacks exploit deliberate incorrect behavior of the hardware during transient execution, similar to Meltdown. In contrast to Meltdown, LVI attacks run in the victim domain and turn the Meltdown-type leakage around into data injection. That is, the victim erroneously runs into transient execution with the injected data values, similar to Spectre. In contrast to Spectre, LVI attacks trigger the transient execution using illegal data flows instead of misprediction. In principle, any data flow can be attacked using LVI, which, based on the attacker's capabilities, can be every load operation in the victim.

Although LVI has similarities to both Meltdown and Spectre attacks, unfortunately, the industry deployed countermeasures against Meltdown and Spectre (silicon-level and microcode), are orthogonal to LVI attacks. Specifically, Spectre defenses stop speculation around (branch) mispredictions, while LVI defenses should stop speculation around all possible illegal data flows (e.g., all loads in a program). Meltdown software and microcode defenses which flush the microarchitectural structures after victim execution [41] cannot mitigate LVI because LVI runs entirely within the victim domain. Even silicon hardening against Meltdown attacks by zeroing illegal data flows [47] do not fully eliminate the LVI threat [91].

Furthermore, applying defenses that target the covert channel that leaks the secret outside the transient execution, *Phases 4*, *5*, and *6*, could hinder LVI attacks. However, these defenses are limited, as we discussed in Section 4.

5.1 LVI-Focused Defenses

Industry

Intel argues that LVI is not practical in non-SGX environments because the attacker has limited ability to cause faults or assists in the victim process in such environments [42]. Therefore, Intel updated the SGX SDK (compiler and assembler-based mitigations) to enable LVI-resilient enclaves. In contrast, hardware-based mitigations could ultimately address LVI's root cause by ensuring that there are no illegal data flows from faulting or assisted loads to dependent instructions. In principle, mitigating specific Meltdown attacks would provide implicit mitigation of the corresponding LVI attacks.

Academia

To fully mitigate LVI attacks without hardware changes, serialization using `lfence` instructions after possibly every illegal data flow, e.g., memory load, is required [91]. However, the performance overheads of such mitigations are prohibitive, and future work has to find better security-performance trade-offs.

6 Conclusion

This paper shows a systematic evolution of defenses against transient-execution attacks in both industry and academia since the initial discoveries of the attacks. We show that despite the advances in the understanding and systematic view of the field, the proposed and deployed defenses are limited.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Sam Ainsworth and Timothy M Jones. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. *arXiv:1911.08384*. 2019.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port contention for fun and profit." In: *S&P*. 2019.

[3] AMD. *AMD64 Technology: Speculative Store Bypass Disable*. 2018.

[4] AMD. *Software Techniques for Managing Speculation on AMD Processor*. 2018.

[5] Nadav Amit, Fred Jacobs, and Michael Wei. "Jumpswitches: restoring the performance of indirect branches in the era of spectre." In: *USENIX ATC*. 2019.

[6] Orlando Arias, David Gens, Yier Jin, Christopher Liebchen, Ahmad-Reza Sadeghi, and Dean Sullivan. "LAZARUS: Practical Side-channel Resilient Kernel-Space Randomization." In: *RAID*. 2017.

[7] ARM. *ARM Architecture Reference Manual ARMv8*. 2013.

[8] ARM. *Cache Speculation Side-channels*. 2018.

[9] Musard Balliu, Mads Dam, and Roberto Guanciale. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. arXiv:1911.00868. 2019.

[10] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E Locasto, Jason Reeves, Sean W Smith, and Anna Shubina. *ELFbac: using the loader format for intent-level semantics and fine-grained protection*. Dartmouth Technical Report, 2013.

[11] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. "Specshield: Shielding speculative data from microarchitectural covert channels." In: *PACT*. 2019.

[12] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "Smotherspectre: exploiting speculative execution through port contention." In: *CCS*. 2019.

[13] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. "MI6: Secure enclaves in a speculative out-of-order processor." In: *MICRO*. 2019.

[14] R Branco, K Hu, K Sun, and H Kawakami. *Efficient mitigation of side-channel based attacks against speculative execution processing architectures.* 2019.

[15] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. "Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification." In: *Electronics* (2019).

[16] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. "The Evolution of Transient-Execution Attacks." In: *GLSVLSI.* 2020.

[17] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. "KASLR: Break It, Fix It, Repeat." In: *AsiaCCS.* 2020.

[18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security.* Extended classification tree and PoCs at https://transient.fail/. 2019.

[19] Chandler Carruth. *RFC: Speculative Load Hardening (a Spectre variant1 mitigation).* 2018.

[20] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. "Defeating Speculative-Execution Attacks on SGX with Hyper-Race." In: *DSC.* 2019.

[21] Chromium Projects. *Actions required to mitigate Speculative Side-Channel Attack techniques.* 2018.

[22] Chromium Projects. *Site Isolation.* 2018.

[23] Jonathan Corbet. *The current state of kernel page-table isolation.* 2017.

[24] Microsoft Corp. 2019. URL: `https://support.microsoft.com/en-us/help/4482887/windows-10-update-kb4482887`.

[25] Sai Manoj P D, Sairaj Amberkar, Sahil Bhat, Abhijitt Dhavlle, Hossein Sayadi, Avesta Sasan, Houman Homayoun, and Setareh Rafatirad. "Adversarial attack on microarchitectural events based malware detectors." In: *DAC.* 2019.

[26] Jonas Depoix and Philipp Altmeyer. "Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning." In: *WAMOS* (2018).

[27] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. “Spectres, virtual ghosts, and hardware support.” In: *HASP*. 2018.

[28] Swastika Dutta and Sayan Sinha. “Performance statistics and learning based detection of exploitative speculative attacks.” In: *CF*. 2019.

[29] R Earnshaw. *Mitigation against unsafe data speculation (CVE-2017-5753)*. 2018.

[30] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. “Processor hardware security vulnerabilities and their detection by unique program execution checking.” In: *DATE*. 2019.

[31] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. “HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security.” In: *CCS*. 2018.

[32] Jacob Fustos, Farzad Farshchi, and Heechul Yun. “SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks.” In: *DAC*. 2019.

[33] Jingquan Ge, Neng Gao, Chenyang Tu, Ji Xiang, and Zeyi Liu. “AdapTimer: Hardware/Software Collaborative Timer Resistant to Flush-Based Cache Attacks on ARM-FPGA Embedded SoC.” In: *ICCD*. 2019.

[34] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. “Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture.” In: *CARRV*. 2019.

[35] Daniel Gruss, Dave Hansen, and Brendan Gregg. “Kernel isolation: From an academic idea to an efficient patch for every computer.” In: *; login: the USENIX Magazine* (2018).

[36] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “Kaslr is dead: long live kaslr.” In: *ESSoS*. 2017.

[37] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR.” In: *CCS*. 2016.

[38] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. "Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference." In: *MICRO*. 2019.

[39] Wei-Ming Hu. "Reducing timing channels with fuzzy time." In: *Journal of computer security* (1992).

[40] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. "EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs." In: *USENIX ATC*. 2018.

[41] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault*. 2018.

[42] Intel. *Deep Dive: Load Value Injection*. 2020.

[43] Intel. *Intel 64 and IA-32 architectures software developer's manual*. 2019.

[44] Intel. *Intel Analysis of Speculative Execution Side Channels*. 2018.

[45] Intel. *Retpoline: A Branch Target Injection Mitigation. Revision*. 2018.

[46] Intel. *Side Channel Mitigation by Product CPU Model*. 2020.

[47] Intel. *Speculative Execution Side Channel Mitigations*. 2018.

[48] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J Peter Brady, Sergey Bratus, and Sean W Smith. "Ghostbusting: Mitigating spectre with intraprocess memory isolation." In: *HoTSoS*. 2020.

[49] Khaled N Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. "RHMD: Evasion-resilient hardware malware detectors." In: *MICRO*. 2017.

[50] Khaled N Khasawneh, Nael B Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. "Adversarial Evasion-Resilient Hardware Malware Detectors." In: *ICCAD*. 2018.

[51] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Safespec: Banishing the spectre of a meltdown with leakage-free speculation." In: *DAC*. 2019.

[52] Russel King. *Spectre-v2: harden branch predictor on context switches*. 2018.

[53] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. “DAWG: A defense against cache timing attacks in speculative execution processors.” In: *MICRO*. 2018.

[54] Vladimir Kiriansky and Carl Waldspurger. “Speculative buffer overflows: Attacks and defenses.” In: *arXiv:1807.03757* (2018).

[55] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution.” In: *S&P*. 2019.

[56] Igor Korkin. “Divide et Impera: MemoryRanger Runs Drivers in Isolated Kernel Spaces.” In: *arXiv:1812.09920* (2018).

[57] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation.” In: *arXiv:1906.01345* (2019).

[58] Congmiao Li and Jean-Luc Gaudiot. “Challenges in Detecting an “Evasive Spectre”.” In: *IEEE Computer Architecture Letters* (2020).

[59] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. “Conditional Speculation: An effective approach to safeguard out-of-order execution against spectre attacks.” In: *HPCA*. 2019.

[60] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space.” In: *USENIX Security*. 2018.

[61] Jason Lowe-Power, Venkatesh Akella, Matthew K Farrens, Samuel T King, and Christopher J Nitta. “Position Paper: A case for exposing extra-architectural state in the ISA.” In: *HASP*. 2018.

[62] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. “Spectre is here to stay: An analysis of side-channels and speculative execution.” In: *arXiv:1902.05178* (2019).

[63] Avi Mendelson. “Secure Speculative Core.” In: *IEEE SOCC*. 2019.

[64] Microsoft. *Microsoft Techcommunity. Hyper-V HyperClear Mitigation for L1 Terminal Fault*. 2018.

[65] Microsoft. *Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer*. 2018.

[66] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. “Retrofitting Fine Grain Isolation in the Firefox Renderer.” In: *USENIX Security*. 2020.

[67] Ejebagom John Ojogbo, Mithuna Thottethodi, and TN Vijaykumar. "Secure automatic bounds checking: prevention is simpler than cure." In: *CGO*. 2020.

[68] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. "You shall not bypass: Employing data dependencies to prevent bounds check bypass." In: *arXiv:1805.08506* (2018).

[69] Hamza Omar and Omer Khan. "IRONHIDE: A Secure Multicore Architecture that Leverages Hardware Isolation Against Microarchitecture State Attacks." In: *arXiv:1904.12729* (2019).

[70] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. "Mitigating data leakage by protecting memory-resident sensitive data." In: *ACSAC*. 2019.

[71] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM addressing for cross-cpu attacks." In: *USENIX Security*. 2016.

[72] Filip Pizlo. *What Spectre and Meltdown mean for WebKit*. 2018.

[73] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site isolation: process separation for web sites within the browser." In: *USENIX Security*. 2019.

[74] Simon Rokicki. "GhostBusters: Mitigating Spectre Attacks on a DBT-Based Processor." In: *DATE*. 2020.

[75] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A Adam Ding. "SCADET: a side-channel attack detection tool for tracking Prime+Probe." In: *ICCAD*. 2018.

[76] Gururaj Saileshwar and Moinuddin K Qureshi. "CleanupSpec: An "Undo" Approach to Safe Speculation." In: *MICRO*. 2019.

[77] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. "Ghost loads: what is the cost of invisible speculation?" In: *CF*. 2019.

[78] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. "Efficient invisible speculative execution through selective delay and value prediction." In: *ISCA*. 2019.

[79] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre." In: *NDSS*. 2020.

[80] Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks." In: *NDSS*. 2018.

[81] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript." In: *FC*. 2017.

[82] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. "Netspectre: Read arbitrary memory over network." In: *ESORICS*. 2019.

[83] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. "Restricting control flow during speculative execution." In: *CCS*. 2018.

[84] Johannes Sianipar, Muhammad Sukmana, and Christoph Meinel. "Moving Sensitive Data Against Live Memory Dumping, Spectre and Meltdown Attacks." In: *2018 26th International Conference on Systems Engineering (ICSEng)*. IEEE. 2018.

[85] Ben Smith. *Enable SharedArrayBuffer by default on non-android*. 2018.

[86] Julian Stecklina and Thomas Prescher. "Lazyfp: Leaking fpu register state using microarchitectural side-channels." In: *arXiv:1806.07480* (2018).

[87] SUSE. *Security update for kernel-firmware*. 2018. URL: `https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1`.

[88] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. "Context-sensitive fencing: Securing speculative execution via microcode customization." In: *ASPLOS*. 2019.

[89] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018.

[90] Eben Upton. *Why Raspberry Pi isn't vulnerable to Spectre or Meltdown*. 2018.

[91] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *S&P*. 2020.

[92] Marco Vassena, Klaus V Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. "Automatically eliminating speculative leaks with blade." In: *arXiv:2005.00294* (2020).

[93] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. "BRB: Mitigating Branch Predictor Side-Channels." In: *HPCA*. 2019.

[94] Luke Wagner. *Mitigations landing for new class of timing attack.* 2018.

[95] Han Wang, Hossein Sayadi, Tinoosh Mohsenin, Liang Zhao, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "Mitigating Cache-Based Side-Channel Attacks through Randomization: A Comprehensive System and Architecture Level Analysis." In: DATE. 2020.

[96] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. *Capability hardware enhanced RISC instructions (CHERI): Notes on the Meltdown and Spectre attacks.* Tech. rep. 2018.

[97] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. "Nda: Preventing speculative execution attacks at their source." In: *MICRO*. 2019.

[98] Ofir Weisse et al. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution.* 2018.

[99] Wenjie Xiong and Jakub Szefer. "Survey of Transient Execution Attacks." In: *arXiv:2005.13435* (2020).

[100] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. "Invisispec: Making speculative execution invisible in the cache hierarchy." In: *MICRO*. 2018.

[101] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. "Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data." In: *MICRO*. 2019.

[102] Lutan Zhao, Peinan Li, Rui Hou, Jiazhen Li, Michael C Huang, Lixin Zhang, Xuehai Qian, and Dan Meng. "A Lightweight Isolation Mechanism for Secure Branch Predictors." In: *arXiv:2005.08183* (2020).

9 Automating Seccomp Filter Generation for Linux Applications

Publication Data

Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. “Automating Seccomp Filter Generation for Linux Applications.” In: *CCSW*. 2021

Contributions

Main author.

Automating Seccomp Filter Generation for Linux Applications

Claudio Canella[1], Mario Werner[1]
Daniel Gruss[1], Michael Schwarz[2]

[1] Graz University of Technology, Austria [2] CISPA Helmholtz Center for Information Security, Germany

Abstract

Software vulnerabilities undermine the security of applications. By blocking unused functionality, the impact of potential exploits can be reduced. While seccomp provides a solution for filtering syscalls, it requires manual implementation of filter rules for each individual application. Recent work has investigated automated approaches for detecting and installing the necessary filter rules. However, as we show, these approaches make assumptions that are not necessary or require overly time-consuming analysis.

In this paper, we propose Chestnut, an automated approach for generating strict syscall filters for Linux userspace applications with lower requirements and limitations. Chestnut comprises two phases, with the first phase consisting of two static components, *i.e.*, a compiler and a binary analyzer, that statically extract the used syscalls. The compiler-based approach of Chestnut is up to factor 73 faster than previous approaches with the same accuracy. On the binary level, our approach extends over previous ones by also applying to non-PIC binaries and thus a broader set of applications. An optional second phase of Chestnut is dynamic refinement to restrict the set of allowed syscalls further. We demonstrate that Chestnut on average blocks 302 syscalls (86.5 %) via the compiler and 288 (82.5 %) using the binary analysis on a set of 18 widespread applications. Chestnut blocks the dangerous `exec` syscall in 50 % and 77.7 % of the tested applications using the compiler- and binary-based approach, respectively. For the tested applications, Chestnut blocks exploitation of more than 61 % of the 175 CVEs that target the kernel via syscalls. Our 6 month long-term study of a sandboxed Nginx server confirms correctness and stability of our approach.

The original publication is available at `https://www.usenix.org/conference/usenixsecurity19/presentation/canella`.

1 Introduction

The complexity of applications is steadily growing, and with that, also the number of vulnerabilities found in applications [38]. A consequence is that the attack surface for exploits is also growing. Especially in applications written in memory unsafe languages such as C, bugs often lead to memory safety violations that potentially enable exploits [59]. Even with state-of-the-art defenses, a risk remains that an attacker can exploit a remaining vulnerability in an application. For privileged applications such as `setuid` binaries, this can, in the worst case, fully compromise the entire system.

The remaining exploitation risk can be addressed by reducing the post-exploitation impact (cf. principle of least privilege). With available resources and interfaces limited to those strictly required by the application, a successful exploit cannot use arbitrary other functionality [34]. Especially blocking dangerous syscalls and syscall parameters that are not required by many applications, e.g., the `exec` syscall to execute a new program, reduces an attacker's possibilities in the post-exploitation phase. Application sandboxing limits the resources available to an application [22, 44] and, ideally, untrusted and potentially malicious, or benign but compromised applications cannot escape the sandbox.

On Linux, seccomp [15] and the extended seccomp-bpf can be used by applications to restrict the syscall interface. Seccomp-bpf [15] supports developer-defined filter rules. Each syscall can be blocked entirely or specific arguments for it. However, the correct usage of seccomp-bpf requires the developer to know which syscalls are used by the application and the included libraries. Given the considerable effort, seccomp is mainly used in applications that provide isolation mechanisms, e.g., sandboxes [26, 50].

Recent works proposed two methods to automatically generate such seccomp filters [11, 20]. The first approach utilizes the compiler and various external tools to derive the filters during compilation [20]. To minimize the set of syscalls, the approach relies on sophisticated points-to analysis [2] to generate a call graph of reachable functions and syscalls. The second approach relies on binary analysis to determine the syscalls an existing binary intends to use [11]. While these are first solutions to the problem of automating filter generation, both come with clear limitations. For instance, the first approach does not scale with the program size due to the points-to analysis [2, 24]. In practice, the overheads can be prohibitively large as they would require a massive upscaling of development and build server resources. The second approach comes with a strong requirement that the application is compiled as a position-independent code (PIC) binary (PIE) [11]. While PIE is the default on recent Ubuntu distributions for C and C++ compiled programs, static C and C++ binaries are by default not compiled as PIE. Other compiled binaries are often not PIE either, e.g., 'golang' binaries such as the popular git server Gogs, which are not supported by these previous works. Both limitations reduce the set of applications that can be protected with these solutions substantially.

In this paper, we present a novel approach that overcomes the limitations of previous ones and automatically generates strict seccomp filters for native Linux userspace applications. We show that our approach is a significant improvement over the compiler-based approach by Ghavamnia et al. [20] without the expensive points-to analysis to generate filter rules. Instead, a faster *has address taken* approach achieves the same accuracy but at a fraction of the performance impact on compilation time. In contrast to DeMarinis et al. [11], we demonstrate that the requirement of a PIC binary is not necessary, significantly extending the set of target applications. We implement our method in a proof-of-concept tool, Chestnut.[1]

Chestnut uses a two-phase process: A static first phase $\boldsymbol{\mathcal{P}1}$ consisting of two static components (**Sourcalyzer** and **Binalyzer**), and an optional dynamic second phase $\boldsymbol{\mathcal{P}2}$ (**Finalyzer**). Using static analysis, Chestnut first identifies unused syscalls without running the application in $\boldsymbol{\mathcal{P}1}$ and dynamically refines this set in $\boldsymbol{\mathcal{P}2}$ to reduce the inherent limitations of the static analysis in $\boldsymbol{\mathcal{P}1}$.

For **Sourcalyzer**, we extend the LLVM framework to detect the syscalls used by the application already during compile- and link-time. The syscall information for each shared library is either extracted using the compiler or using Binalyzer. **Binalyzer** can be used for applications and libraries which are either not compatible with LLVM or where the source code is not available. We rely on capstone [47] to disassemble applications and to locate syscalls. Using symbolic backward execution [36] from the `syscall` instruction, we infer the syscall number used in the identified syscall. Additionally, we use the control-flow graph (CFG) recovery functionality of angr [64] to map exported functions to identified syscalls. Exactly as in previous work, an inherent limitation of static approaches is that they can miss syscalls in rare cases if control-flow cannot be inferred correctly. However, we observe that more frequently, the set of used syscalls is overapproximated. To refine the number of allowed syscalls, we

[1]The prototype and several demo videos can be found in our anonymous GitHub repository `https://github.com/chestnut-sandbox/Chestnut`.

provide a complementary optional dynamic approach in the second phase of Chestnut. In this second phase, **Finalyzer** traces all syscalls of the application and then refines the allowlist to further restrict or relax the seccomp filters.

We demonstrate our approach's feasibility by applying it to various real-world client applications, such as git and busybox, database applications, such as redis and sqlite3, and Nginx as a server application. We show that Chestnut does not impair their functionality but significantly reduces the attack surface. On average, Chestnut blocks 295 syscalls (84.5 %) on Linux kernel 5.0. In the 18 real-world binaries we evaluated, Chestnut blocked the `exec` syscall for 50 % of the applications using Sourcalyzer and in 77.7 % using Binalyzer. We block the `mprotect` syscall in 61.1 % of the tested applications using Sourcalyzer. Furthermore, we evaluate our approach on real-world exploits, showing that Chestnut blocks exploitation of 64 % and 62 % of CVEs using Sourcalyzer and Binalyzer, respectively. We compare our approaches with related previous work [11, 20] and show that we are similarly effective in mitigating CVEs via compiler-based approaches [20]. However, we improve the performance by up to factor 73. In contrast to previous work [11], we show that binary-based approaches can also be applied to non-PIC binaries. We evaluate the functional correctness of Sourcalyzer with test suites as well as a 6-month long-term case study of a Sourcalyzer-protected Nginx production server. Furthermore, we are the first to evaluate how tight automatically generated filter rules are by relying on available test suites. We also advance the state of the art in evaluation of automatic syscall filtering, with a long-term case study and by measuring code coverage to confirm our approach's validity.

Filters generated automatically with a tool might not always be as strict as theoretically possible. However, there is no time investment required by the developer, making it a very inexpensive defense in depth. More importantly, Chestnut can be applied to and improve the security of existing and widely-used technology, *i.e.*, seccomp, making syscall filtering available to commodity applications. The only runtime overhead introduced is the small overhead of using seccomp, similar as containers already do today.

To summarize, we make the following contributions:

1. We present a new compiler-based approach for automated syscall-filter generation, up to 73x faster than previous work.
2. We present a dynamic method to refine the filters.
3. We show that Chestnut blocks the exploitation of more than 61 % of the 175 CVEs in the Linux kernel exploitable via syscalls.

4. We show that previous requirements can be lifted and show that our approach can be applied to PIC and non-PIC binaries.
5. In a 6 month long-term study on Nginx, we demonstrate the correctness of our approach and do not observe a single crash.

Outline. Section 2 provides background, Section 3, threat model, and design of Chestnut. Section 4 discusses our static and Section 5 our dynamic approach. We evaluate Chestnut in Section 6 and discuss related work and limitations in Section 7. Section 8 concludes.

2 Background

Sandboxing is a security mechanism that constrains software within a tightly controlled environment by restricting the available resources to a required minimum [22, 44]. Hence, the damage in case of exploitation is limited. These restrictions may encompass the ability to access the network, limit the amount of storage, file descriptors, or inhibit the application from issuing specific syscalls. By now, different forms of sandboxing have been adopted by many browser vendors to secure their products [43, 49, 58, 66].

2.1 Linux Seccomp

To facilitate operations that require higher privileges or direct hardware access, the kernel provides syscalls to every userspace application. As with other interfaces, they also contain bugs that can lead to privilege escalation [30–32]. Hence, platform security profits from limiting the amount of syscalls that an application can perform. With Secure Computing (seccomp) [15], Linux provides a filter that allows a userspace program to specify the syscalls it performs over its lifetime. The kernel then blocks the remaining syscalls for the sandboxed application that might originate from an application being hijacked. As seccomp filters do not dereference pointers, so-called time-of-check time-of-use attacks [37] common in syscall interposition frameworks are not possible. Examples of applications that rely on seccomp are Chromium [9], Firefox [40], and the zygote process in Android systems [27].

2.2 Memory Safety

Memory safety is an essential concept in computer security, and its violation can lead to exploitation. One way to exploit a program is to corrupt its memory and to divert control flow to a previously injected code sequence. This code sequence, *i.e.*, the payload, is called *shellcode* and is commonly written in machine code. These types of attacks are commonly known as control-flow hijack attacks [59].

ROP attacks [53] allow chaining existing code gadgets within an application to perform malicious tasks. Each gadget is a sequence of instructions ending with a return instruction. ROP attacks are hard to defend as all the information is already present within the application, *i.e.*, an attacker does not need to inject code. While ROP attacks overwrite saved return addresses, similar attacks exist that overwrite other pointers [4, 6, 7, 21, 51] or signal handlers [5].

2.3 Executable and Linkable Format

On Unix-based systems, the highly flexible and extensible Executable and Linkable Format (ELF) [13] is used for shared libraries and executables. ELF files consist of header and data, including a program and a section header table for segments and sections. Segments contain information for the run-time execution of the binary, while sections contain information for linking and relocating.

Dynamic Linking. The dynamic linker is responsible for loading and linking shared libraries used by an executable during runtime [13]. For that, the dynamic linker maps the shared library's content into memory and ensures its functionality, e.g., filling jump tables and relocating pointers. On Unix-like systems, the dynamic linker is selected at link time and is embedded into the ELF file.

3 Design of Chestnut

In this section, we introduce our threat model, outline challenges of automatic filter generation, and discuss the high-level idea of Chestnut. We introduce the main components of Chestnut (Figure 9.1), *i.e.*, the compiler modification *Sourcalyzer*, the binary analyzer *Binalyzer*, and the dynamic refinement tool *Finalyzer*.

3.1 Threat Model and Idea of Chestnut

Chestnut supports Linux applications available as either C source code or as a binary, and is not limited to PIC binaries as previous work [11]. These applications can range from server applications to applications executing

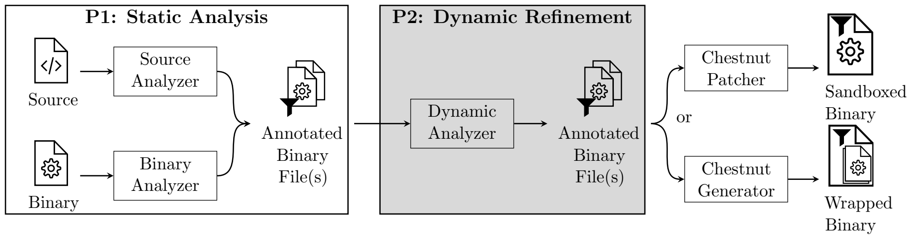


Figure 9.1: Chestnut consists of $\mathcal{P}$**1**, a static analysis on source and binary files, and an optional dynamic analysis, $\mathcal{P}$**2**, refining the filters. Chestnut can either rewrite the binary or generate a tailored sandbox to block unused syscalls.

potentially malicious code that is not controlled by the user, such as browsers, office applications [41], and pdf readers [16, 17]. Chestnut can also restrict the syscall interface of messenger applications which have been used to compromise systems [23, 54]. We assume correct usage of Chestnut in one of its variants (cf. Figure 9.1). Chestnut assumes that the application is not malicious but potentially vulnerable to exploitation, e.g., due to a memory-safety violation, enabling an attacker to gain arbitrary code execution within the application. We assume that post-exploitation requires syscalls, e.g., to gain kernel privileges. Syscalls provided for file operations can potentially be hijacked by an attacker to modify configuration files. Argument-level API specialization [39] can be used to protect against such attacks. We consider this an orthogonal problem, in line with related work [20], and do not consider such attacks. Chestnut is orthogonal to other defenses such as CFI, ASLR, NX, or canary-based protections and enhances the security in case these other mitigations have been circumvented. Side-channel and fault attacks are out of scope.

3.2 Challenges

Automatic filter generation using a static approach requires solving the following four challenges that we detail in Sections 4 and 5.

$\mathcal{C}$1: Identifying Syscall Numbers for each Syscall. To automatically block unused syscalls, we must identify the syscalls used by the application. The syscall itself is usually a single instruction, e.g., `syscall` (x86_64) or `svc #0` (AArch64). The actual syscall is specified as a number in a CPU register, e.g., `rax` (x86_64) or `x8` (AArch64) [12]. Hence, the first challenge is to identify the syscall number for a specific syscall. Syscalls have many different forms within a program, e.g., inline assembly, assembly file, or issued with the libc *syscall* wrapper function. Moreover, syscalls might not be called directly, but via a call chain through various libraries.

$\mathcal{C}$2: Reconstruct Call Sites of Syscalls. By solving challenge **$\mathcal{C}$1**, we know which syscalls are potentially called by the target application. Unfortunately, including all detected syscalls of the binary and the used libraries does not suffice. Most binaries link against libc, which provides an implementation of almost all syscalls. Hence, the generated filters would be too permissive as they would basically allow all syscalls. We have to analyze the reachability of the identified syscalls by constructing a call graph for every binary.

$\mathcal{C}$3: Generate Set of Syscalls. To generate the final set of syscalls for our application, the information from **$\mathcal{C}$1** and **$\mathcal{C}$2** has to be combined for the application and its libraries. By combining the call graph obtained in **$\mathcal{C}$2** with the information which functions are used in the application and libraries, we create a set of functions potentially called by the application. In combination with the call graph (**$\mathcal{C}$2**) and syscall numbers (**$\mathcal{C}$1**), this set provides the information about all the syscalls that the application can execute.

$\mathcal{C}$4: Install Filters. Our library (libchestnut) relies on seccomp to apply the syscall filters. This library uses the allowlist (**$\mathcal{C}$3**), generates the seccomp rules, and installs the resulting filters before the actual application starts at the main entry point.

3.3 High-Level Idea

This section discusses how the components of Chestnut solve the challenges, with the implementation details in Sections 4 and 5.

Sourcalyzer. Sourcalyzer is the compiler-based component of Chestnut for static analysis of the application source code. It is a compiler pass in LLVM that identifies all syscalls at compile time.

For statically linked applications, and given that libraries are compiled with Sourcalyzer, the compiler and linker are aware of the entire codebase and can thus identify every syscall instruction of the final binary. As the C standard library implements almost all syscalls, linking against it would allow almost all syscalls, which renders the filters ineffective. Hence, we need to determine further which syscalls are used by the application by analyzing the CFG to solve challenges **$\mathcal{C}$2** and **$\mathcal{C}$3**. While comparable work [20] needs to perform the same task, we demonstrate a solution that is up to factor 73 faster. We discuss this in Section 4.1.

Binalyzer. Sourcalyzer requires the source code of the application and all used libraries. Binalyzer has the same goal but works directly on the binary level. With this, our approach is also applicable to programs where the source code is not available or not compatible with LLVM, retrofitting the approach to binaries. In contrast to previous work [11], Binalyzer is not restricted to PIC binaries.

The idea is to scan binaries and libraries for *syscall* instructions and use symbolic backward execution [36] from these locations to infer the respective syscall number, again solving challenge $\boldsymbol{\mathcal{C}1}$. To reduce overapproximation, Binalyzer leverages CFG analysis of all dependencies to map exported functions to syscall numbers ($\boldsymbol{\mathcal{C}2}$, $\boldsymbol{\mathcal{C}3}$). Binalyzer also works on stripped binaries as all the required information is still included for dynamic linking.

Finalyzer. Working around limitations of static analysis, we propose an optional dynamic phase, Finalyzer, based on syscall tracing, removing or adding filters that cannot be identified statically. Finalyzer is solely intended to refine filters identified by our static approaches in a controlled, benign environment.

The dynamic nature of Finalyzer allows simplifying challenges $\boldsymbol{\mathcal{C}1}$ to $\boldsymbol{\mathcal{C}4}$. Finalyzer extracts the syscall number during runtime ($\boldsymbol{\mathcal{C}1}$) by intercepting all syscalls for the target application. By intercepting the syscall, it is inherent that the syscall is reachable ($\boldsymbol{\mathcal{C}2}$). In this step, missed syscalls are added to refine the installed filter list ($\boldsymbol{\mathcal{C}4}$). We discuss this process in more detail in Section 5.

Combining Components. Chestnut is designed to allow combining all three components (cf. Figure 9.1). For instance, Finalyzer is intended to be used as an optional step after the static components if they cannot infer the used syscalls due to the static analysis's limitations. An instance where this is necessary is when an application dynamically starts other applications. The child process inherits the parent's filters, which cannot be relaxed anymore. By combining the static approaches with Finalyzer, the syscalls of the child process are identified and added to the application's allowlist. Sourcalyzer can also be used in combination with Binalyzer, e.g., if the source is available for the application but not for a used library.

Applying Syscall Filters. The output of each component is a set of syscalls the application can call. For Sourcalyzer, the syscall filters are directly compiled into the target application. However, if this is either not desirable or possible, e.g., because only the binary is available, we provide two tools to apply the syscall filters (cf. Figure 9.1). ChestnutGenerator creates a sandbox tailored to the target application. Alternatively, Chest-

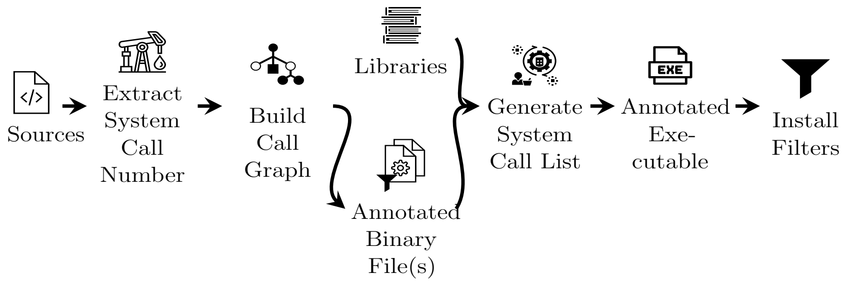


Figure 9.2: The different steps of Sourcalyzer, which starts with the source and ends with a fully sandboxed application.

nutPatcher directly patches the target application to include the syscall filters and libchestnut.

4 Static Filter Extraction

We now present the two static approaches of $\mathcal{P}\mathbf{1}$ to automatically generate syscall filters. We highlight the necessary steps, cf. Figure 9.2, for solving the outlined challenges in a fast and efficient way in both a compiler and a binary-based approach in more detail.

4.1 Compiler-Based Approach

Sourcalyzer utilizes the LLVM compiler framework [33] to extract syscalls from source code. It uses module passes (*i.e.*, one analysis and one transformation pass) that operate on the LLVM intermediate representation (IR). Additionally, LLVM's linker lld is extended to combine the extracted information from multiple translation units. We use an unmodified compiler-rt and musl libc. Hence, using Chestnut with the Sourcalyzer approach is as simple as compiling and linking an application with our extended toolchain.

$\mathcal{C}$1: Identify System Call Numbers. To invoke a syscall, x86_64 provides the *syscall* and AArch64 the *svc #0* instruction. The extraction of the syscall number is the only architecture-dependent part of Chestnut. Given the syscall number, the rest of the approach is the same for all architectures supported by LLVM. Syscalls are typically abstracted by the standard C library via the *syscall()* function. musl additionally provides the function *__syscall_cp()*. To detect all invocations of syscalls, we need to detect all three cases, *i.e.*, inline assembly, the *syscall* function, and the *__syscall_cp()* function.

The LLVM analysis pass iterates over all functions within a translation unit. For each function, we iterate over every LLVM IR instruction to check whether it is a call site. If it is, we check whether it is an inline syscall assembly statement or a call to either one of the *syscall* or *__syscall_cp* functions. In all three cases, we extract the first argument as it is the number of the requested syscall. Due to the way we traverse the IR, we also know precisely what function performs the respective syscall.

Our proof-of-concept implementation currently does not parse assembly files as they are treated differently by LLVM than normal source files. Hence, if a syscall is implemented in one, e.g., *clone*, we cannot detect it, but a full implementation can handle this case.

$\mathcal{C}$2: Reconstruct Syscall Call Sites. Sourcalyzer uses the syscall numbers extracted in **$\mathcal{C}$1** as a starting point for further analyzing which syscalls are used based on the call graph. The main challenge here is extracting a reasonably precise call graph without inducing huge performance overheads or by requiring changes to the common compilation model (e.g., by demanding link-time optimization (LTO)). In particular, restricting indirect function call sites to a set of possible call targets is a necessary, but typically quite expensive, task that commonly relies on inter-procedural pointer analysis (e.g., Andersen [2], Steensgaard [57]). This type of analysis requires access to the whole program and often does not scale efficiently to larger program sizes [2, 24]. An automated syscall-detection system based on this form of analysis and its impact on the compile-time performance has been demonstrated by Ghavamnia et al. [20]. This approach also requires changes to the common compilation model, which is not supported by every application.

Hence, as we want to avoid changes to the compilation model and given that our application can tolerate some imprecision, we do not use sophisticated pointer analysis in our prototype implementation and opt for a function-signature based heuristic to determine possible call targets. Every function in the program where the function type of the call site matches the function type of the definition is considered a possible call target. For correctly typed programs, this heuristic is an overapproximation of the actual possible call targets, which corresponds to permitting more syscalls than are actually needed (cf. Section 6.4).

Both the LLVM IR passes and the linker are involved in mapping syscall numbers to functions. Our analysis pass traverses over all defined functions within the module. It extracts function signature, direct and indirect function calls, and functions referenced in the code (for which function pointers exist, *i.e.*, functions that have their address taken). The latter is similar to what LLVM uses for software-based CFI. This gives rise to the assumption that the resulting call graph is precise enough as applications that use software-based CFI would otherwise not work correctly.

We perform our analysis in the same traversal in which we locate the used syscall numbers (see ***C*1**). We also support function aliases by treating them as copies of the original function. Finally, references to functions in global initializers are extracted, as they are used, e.g., for global file structures.

Our IR transformation pass stores the information collected from the analysis pass in the ELF object for the linker to use.

***C*3: Generate Syscall Set**. By solving challenges ***C*1** and ***C*2**, we generate object files containing the serialized syscall and call graph information. The linker extracts this information from all the provided input files to perform the actual call graph construction and syscall number propagation. Finally, the linker can either generate the set of relevant syscalls for the application or a flattened call graph for further processing.

In more detail, after loading the call graph metadata, all reachable functions are resolved according to their symbol's linkage specification (e.g., local or global, strong or weak), and a list of indirect callable functions is generated. In the next step, a call graph is constructed in which each node represents a function, and each directed edge represents a possible control-flow transfer from the caller to the callee. The linker transforms this call graph into a directed acyclic graph (DAG) using Tarjan's algorithm [60], enabling efficient propagation of the information. Namely, each graph node has to be updated only once by visiting the DAG in post-order. Using the discovered strongly coupled components, circular call dependencies are directly resolved by merging the information from all functions that are part of the respective cycle in the original call graph. As a result, the linker has access to a flattened call graph in which, for every function, all reachable syscall numbers are known.

With the flattened call graph, we determine which syscalls our final application needs. For a static binary, we extract all syscalls that can be reached from the *main* and the *exit* function and embed them as a simple list of numbers into the final ELF binary. For dynamic binaries or shared libraries, we instead embed the flattened call graph into the linked binary for further processing.

***C*4: Install Seccomp Filters**. After linking with Sourcalyzer, the binary contains annotations for the application's used syscall numbers directly or its flattened call graph that still needs to be combined with the

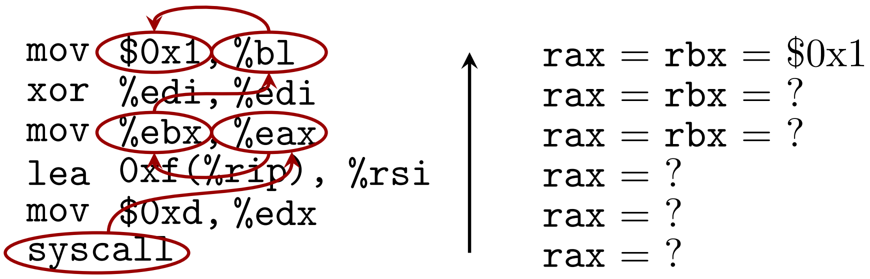


Figure 9.3: Symbolic backward execution starts from the syscall instructions and finds the syscall number by symbolically tracking the corresponding CPU register.

additional dynamic libraries. For static linkage, we delegate the processing to the application itself by additionally linking against our libchestnut library. This library contains a constructor that extracts the syscall numbers and installs the seccomp filters using libseccomp [14] before the application starts executing.

In the second case, dynamic linkage, we provide two options. ChestnutPatcher extracts the embedded call graph from all library dependencies and determines all syscalls from functions that are reachable from the *main* and *exit* function. Finally, the tool adds a new note section with information on syscall numbers. As the compiler has generated the dynamic binary, we can already link libchestnut against it automatically. ChestnutGenerator performs the same steps except that it does not modify the binary but creates a launcher that sets up the filters before executing the actual binary.

4.2 Binary Syscall Extraction

The second static approach of Chestnut, Binalyzer, works on the binary level. With less semantic information available than on the compiler level, Binalyzer works without access to the source code and even for stripped binaries. In contrast to previous work [11], we demonstrate our approach on PIC and non-PIC binaries.

$\mathcal{C}$1: Identify Syscall Numbers. The syscall number is not encoded in the instruction but is provided in a register, *i.e.*, `rax` on x86_64 or `x8` on AArch64. Hence, Binalyzer has to reconstruct the syscall number by inferring the content of this register.

Binalyzer uses the capstone framework [47] to disassemble a binary as this framework supports various ISAs, e.g., x86_64 and AArch64. Starting from a syscall instruction, Binalyzer leverages symbolic backward execution [36]. Tracking back from the syscall instruction, Binalyzer tracks the register's symbolic value containing the syscall number. In many cases, the immediate for the syscall number is directly moved to the register before the syscall instruction as it is a constant value. However, in some cases, there is at least some form of register-to-register transfer involved. These transfers also include register copies where only a lower part of the register is involved. Thus, as illustrated in Figure 9.3, Binalyzer keeps the content of the register symbolic and steps back through the binary, symbolically evaluating operations. This symbolic backward execution is repeated until either a concrete immediate for the syscall number is identified, or after a user-definable number of instructions have been analyzed without successfully identifying the immediate.[1] One failure reason can be that the syscall instruction is a call or jump target, *i.e.*, there are potentially multiple call sites reaching the instruction with different syscall numbers. Luckily, the syscall instruction is usually inlined, and thus, we do not consider such situations for our proof of concept.

***$\mathcal{C}$*2: Reconstruct Syscall Call Sites**. To reduce the overapproximation of syscalls, Binalyzer analyzes the CFG to map syscalls to (exported) functions. We rely on angr [64] to statically create a CFG of the binary. Based on the basic blocks of all functions in the CFG, we assign every syscall identified in **$\mathcal{C}$1** to a function. Binalyzer traverses the CFG from each exported function as the root node to identify reachable functions with a syscall instruction. Assuming a correctly reconstructed CFG from angr and correctly identified syscall numbers (**$\mathcal{C}$1**), this yields a set of possible syscalls per exported function.

$\mathcal{C}$3: Generate Syscall Set. To solve challenge **$\mathcal{C}$3**, we have to combine the information created from solving **$\mathcal{C}$2** for all binaries, *i.e.*, the application binary and all of its libraries. We cannot create a complete CFG over the application binary and its libraries as this would take multiple hours to days, depending on the size of the application and the number of dynamic libraries. Instead, we chose to overapproximate the number of possible syscalls by relying on individual CFGs that we merge. We only consider functions that are defined in the dynamic symbol table of the ELF file. These functions are found in the dynamic libraries loaded by the application. Hence, we search for these functions in the shared object dependencies and look up the used syscalls for the function in the respective library. As shared libraries can also have a dynamic symbol

[1]For the evaluation, we set this number to 30, which was sufficiently large.

table if they call functions from other libraries, this process is repeated for all dynamic symbols of all shared object dependencies.

Solving challenge $\mathcal{C}$**3** yields a set of syscalls that the application can potentially call. This assumes that no dynamic libraries are loaded at runtime, e.g., via `dlopen`, and that the application does not execute a different binary at runtime, e.g., via `exec`. In such cases, we need to resort to $\mathcal{P}$**2** as the complete set of syscalls cannot be determined statically.

$\mathcal{C}$4: Install Seccomp Filters. From the full set of syscalls, Binalyzer has to create filter rules and apply them to the binary. We cannot simply compile the filters with the application (cf. Sourcalyzer). Instead, Binalyzer supports binary rewriting or alternatively building a sandbox wrapper (cf. Figure 9.1). ChestnutGenerator is a simple tool that sets up the filter rules and starts the target program.

With binary rewriting, Binalyzer stores the syscall numbers in the ELF binary and injects a new shared object dependency, libchestnut. The library provides a constructor function, which is called before the actual application starts and which parses the filters stored in the binary to apply the seccomp filter rules. The advantage of a rewritten binary is that it does not need any launcher.

5 Dynamic Refinement

In this section, we discuss the optional $\mathcal{P}$**2** component Finalyzer, a method to dynamically refine the previously detected syscall filters. It simplifies the challenges $\mathcal{C}$**1** to $\mathcal{C}$**4** by inspecting syscalls just-in-time in a secure and controlled environment during development.

5.1 Limitations of Static Approaches

While our approach for statically detecting an application's syscalls works well for most binaries (cf. Section 6), there are inherent limitations to a static approach. Dynamically loaded libraries, e.g., codecs, plugins, self-modifying, or JIT-compiled code, often cannot be analyzed statically. Moreover, seccomp is not flexible enough to handle scenarios involving child processes with a different set of syscalls, as a child inherits its parent's filters and can only further restrict but not relax them. Hence, the parent also needs to install filters for the child's syscalls.

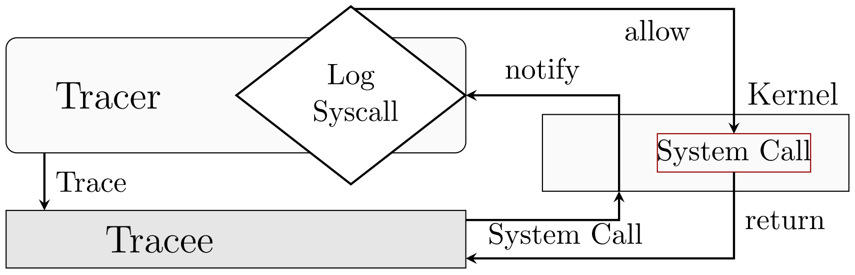


Figure 9.4: The tracer gets notified by the kernel when the tracee executes a syscall. The tracer logs the syscall and informs the kernel to execute the syscall.

5.2 Implementation Details

In our prototype, Finalyzer is an strace-like syscall-tracing component linked against the target application or used as a standalone wrapper for a binary (cf. Figure 9.1). This allows Finalyzer to work with Binalyzer and Sourcalyzer. If desired, it can also be used without the static components to identify required syscalls.

In either case, Finalyzer, *i.e.*, the *tracer*, first creates a child process, the *tracee*. Finalyzer then installs seccomp filters for all syscalls in a way that informs the tracer about a seccomp violation. To enable this behavior, the tracer needs to attach itself to the tracee. The tracee then stops execution until it receives the continue signal from the tracer to ensure that it successfully attached itself. If the child process creates a new child process, the tracer is automatically attached to the newly created child process. The tracer is then also informed of the unsuccessful execution of the child's syscalls.

When receiving the notification of a violating syscall, Finalyzer extracts the syscall number ($\boldsymbol{\mathcal{C}1}$), logs it ($\boldsymbol{\mathcal{C}3}$), and allows it for all future occurrences (cf. Section 3.3), as illustrated in Figure 9.4. As the syscall is indeed executed, it is inherent that it is reachable ($\boldsymbol{\mathcal{C}2}$).

Once Finalyzer has finished tracing the application, it cross-references the set of obtained syscalls with the ones obtained in $\boldsymbol{\mathcal{P}1}$. If a syscall is missing, it adds the newly detected syscall to the allowlist. Optionally, it can also be used to remove syscalls that $\boldsymbol{\mathcal{P}1}$ identified but which were never executed during $\boldsymbol{\mathcal{P}2}$.

6 Evaluation

In this section, we evaluate the performance, functional correctness, and security of Chestnut. Our evaluation is in line with related work [11, 20] while improving on it in several points, *i.e.*, we evaluate several parts that were not yet evaluated in these works. In the performance evaluation, we evaluate the one-time overheads of Chestnut, such as compile time and binary-analysis time. We also discuss the runtime overhead seccomp introduces. For the functional correctness, we evaluate whether Chestnut causes any issues in terms of functionality of existing real-world software, e.g., crashes. We also perform a 6-months long evaluation of an Nginx server with its syscall interface restricted by Sourcalyzer. In the security evaluation, we evaluate the ability of Chestnut to block the dangerous `exec` syscall and the overapproximation of syscalls in general. With the latter, we are the first to demonstrate how tight the automatically generated filters are. Furthermore, we evaluate how well Chestnut can mitigate real-world exploits. Finally, we detail differences between our work and related works in this field.

6.1 Setup

For the evaluation of Chestnut, we focus on x86_64. Note that the only architecture-dependent part of Chestnut is the extraction of the syscall number. Hence, we do not expect significant differences for other architectures. We also verified that the general approach works across architectures by successfully extracting the syscall numbers from musl libc for both x86_64 and AArch64.

We evaluate Chestnut on various real-world applications (cf. Table 9.1), including client, server, and database software. While busybox may be seen as a non-obvious choice, it is in line with previous work that used coreutils for the evaluation [46]. We instead chose busybox as the number of provided utilities is 3 times higher, making it a better choice for our evaluations. For evaluating Sourcalyzer, we compile the binaries statically with and without Chestnut enabled using our modified compiler. For Binalyzer, we compile the applications dynamically using GCC 7.5.0-3 on Ubuntu 18.04.4. For the sake of brevity, we do not evaluate every combination of components and sandboxes but focus on libchestnut for Sourcalyzer and ChestnutGenerator for Binalyzer.

Table 9.1: Results for the compiler- (</>) and binary-based (⚙) approach of Chestnut, respectively. We show the number of detected syscalls in $\mathcal{P}\mathbf{1}$, used syscalls, and added syscalls in $\mathcal{P}\mathbf{2}$, the size overhead of the annotations, compile-time overhead (for Sourcalyzer), and binary analysis time (for Binalyzer). The `exec` and `mprotect` columns indicate whether Chestnut blocks (✓) the respective syscalls or not (✗). We also show the percentage of fully mitigated CVEs and the individual subvariants. Syscalls added in $\mathcal{P}\mathbf{2}$ are only necessary for edge cases in our proof-of-concept implementation of Chestnut.

	Software	**#Syscalls Found / Used / $\mathcal{P}$2 Added**		**Size Overhead**		**Analysis Time**		`exec`		`mprotect`		**Fully Mitigated**		**Subvariant Mitigated**	
		</>	⚙	Compiler (</>)	Binary (⚙)	Compiler (</>)	Binary (⚙)	</>	⚙	</>	⚙	</>	⚙	</>	⚙
Client	ls	24 / 14 / 0	39 / 18 / 1	+173 kB (253 %)	+288 B (1.08 %)	+0.38 s (1.72 %)	3.041 s	✓	✓	✓	✗	81.1 %	81.1 %	87.5 %	87.5 %
	chown	22 / 11 / 0	36 / 14 / 0	+174 kB (369 %)	+280 B (1.52 %)	+0.29 s (1.35 %)	2.777 s	✓	✓	✓	✗	81.7 %	81.7 %	87.8 %	87.8 %
	cat	18 / 6 / 0	34 / 13 / 0	+174 kB (397 %)	+272 B (1.9 %)	+0.08 s (2.29 %)	2.576 s	✓	✓	✓	✗	82.3 %	81.7 %	88.1 %	87.8 %
	pwd	16 / 4 / 0	34 / 14 / 0	+175 kB (430 %)	+272 B (1.92 %)	+0.21 s (0.98 %)	2.507 s	✓	✓	✓	✗	85.1 %	81.7 %	90.3 %	87.8 %
	diff	25 / 9 / 0	36 / 16 / 0	+173 kB (304 %)	+280 B (1.25 %)	+0.06 s (1.44 %)	2.946 s	✓	✓	✓	✗	81.1 %	81.7 %	87.5 %	87.8 %
	dmesg	15 / 5 / 0	34 / 14 / 0	+176 kB (439 %)	+272 B (1.92 %)	+0.08 s (2.17 %)	2.452 s	✓	✓	✓	✗	85.1 %	81.7 %	90.3 %	87.8 %
	env	15 / 3 / 0	33 / 13 / 0	+175 kB (416 %)	+272 B (1.92 %)	+0.07 s (1.88 %)	2.416 s	✗	✓	✓	✗	81.7 %	81.7 %	88.4 %	87.8 %
	grep	20 / 11 / 0	34 / 16 / 0	+174 kB (177 %)	+272 B (1.49 %)	+0.41 s (1.88 %)	2.748 s	✓	✓	✓	✗	81.7 %	81.7 %	87.8 %	87.8 %
	true	3 / 1 / 0	32 / 12 / 0	+200 kB (3277 %)	+264 B (4.4 %)	+0.09 s (2.55 %)	9.908 s	✓	✓	✓	✗	98.3 %	83.4 %	98.4 %	88.7 %
	head	17 / 7 / 0	33 / 13 / 0	+174 kB (434 %)	+272 B (1.92 %)	+0.06 s (1.64 %)	2.436 s	✓	✓	✓	✗	82.3 %	81.7 %	88.1 %	87.8 %
	git	82 / 42 / 1	85 / 42 / 2	+219 kB (4.5 %)	+448 B (0.003 %)	+18.5 s (28.18 %)	247 s	✗	✗	✗	✗	34.3 %	58.3 %	55.9 %	73.4 %
	FFmpeg	63 / 27 / 1	91 / 27 / 2	+190 kB (0.21 %)	+472 B (0 %)	+268 s (27.14 %)	643 s	✗	✓	✗	✗	33.1 %	34.9 %	57.8 %	44.1 %
	mutool	61 / 16 / 1	69 / 15 / 0	+189 kB (0.48 %)	+376 B (0.001 %)	+3.17 s (0.69 %)	164 s	✗	✓	✗	✗	52.0 %	38.3 %	69.4 %	60.3 %
	memcached	88 / 54 / 1	102 / 59 / 4	+216 kB (28.9 %)	+456 B (0.13 %)	+0.35 s (5.5 %)	8 s	✗	✓	✗	✗	30.3 %	33.7 %	50.0 %	41.9 %
DB	redis-server	85 / 35 / 1	93 / 42 / 3	+216 kB (11.2 %)	+472 B (0 %)	+2.4 s (2.7 %)	41 s	✗	✗	✗	✗	30.3 %	32.0 %	54.1 %	54.4 %
	sqlite3	92 / 72 / 1	102 / 72 / 13	+215 kB (5.9 %)	+456 B (0.02 %)	+0.8 s (7.2 %)	45 s	✗	✗	✗	✗	62.9 %	32.6 %	75.3 %	57.5 %
Server	Nginx	105 / 48 / 0	106 / 51 / 4	+217 kB (1.5 %)	+528 B (0.003 %)	+7.9 s (10.53 %)	277 s	✗	✓	✓	✗	32.0 %	30.9 %	38.8 %	40.0 %
	httpd	98 / 50 / 1	106 / 46 / 0	+218 kB (8.3 %)	+504 B (0.04 %)	+4.1 s (5 %)	16.8 s	✗	✗	✗	✗	29.7 %	30.3 %	50.9 %	44.4 %

6.2 Performance Evaluation

In this section, we evaluate the performance of Chestnut. This includes the one-time overheads for compiling (Section 6.2) or binary analysis (Section 6.2), the increase in binary size (Section 6.2), and runtime overheads (Section 6.2).

Compile-Time Overhead

We analyze the impact Sourcalyzer has on the compile time of an application. To make comparison possible, we compile the application 10 times with and without our modification enabled, always using our modified compiler, and use the average compile time over these runs.

As the results show, we observe the worst-case overhead for the git application with an increase from an average of 65.5 s ($\sigma_{\bar{x}} = 0.094$, $N = 10$) to 84 s ($\sigma_{\bar{x}} = 0.054$, $N = 10$), an increase of 28 %. For the busybox utilities combined, the average increases from 10.94 s ($\sigma_{\bar{x}} = 2.88$, $N = 10$) to 10.99 s ($\sigma_{\bar{x}} = 2.79$, $N = 10$). When compared to related work [20], we observe a speedup of factor 73 for Nginx when using Sourcalyzer. This low overhead makes it a feasible approach to be used in everyday development cycles.

Binary Extraction Runtime

For Binalyzer, we evaluate the time it takes to extract the syscalls from the dynamic binary. We assume that default dependencies like *libc.so* have already been processed, *i.e.*, their extracted call graph is available. For completeness, we timed the extraction of syscalls from *libc.so*, which takes on average 44.66 s ($\sigma_{\bar{x}} = 0.18$, $N = 10$). For the applications themselves, we can see in Table 9.1 that the extraction process is in the range of 2 to 10 s for the individual busybox utilities, with an average time of 3.4 s ($\sigma_{\bar{x}} = 0.73$, $N = 10$). For large binaries like FFmpeg (> 100 MB) and its dynamic dependencies, the extraction takes ≈11 min.

Binary Size Analysis

The code size does not increase with added filters, but the binary size increases by the meta-information.

Compiler. We analyze the size of the binary produced by Sourcalyzer compared to a vanilla application. Chestnut needs to treat static and dynamic ELF files differently as syscall numbers of externally linked libraries are not known. In a static binary, we only add the set of syscall numbers to the binary and link against libchestnut and libseccomp. As

Table 9.2: We evaluate the size overhead of Sourcalyzer on shared libraries compared to a vanilla version.

Shared library	**Vanilla**	**Annotated**	**Overhead**
musl libc.so	815 kB	1007 kB	23.63 %
libssl.so	657 kB	1.7 MB	161 %
libcrypto.so	4.1 MB	23 MB	460 %

both libraries are of fixed size, the maximum overhead in a static binary is limited by the number of syscalls Linux provides, *i.e.*, 349 on Linux 5.0. Table 9.1 shows the overhead for statically linked binaries. As expected, the overhead is quite small in large binaries, e.g., FFmpeg. In the small busybox utilities, the overhead appears to be huge ($> 177\,\%$), but as these binaries sizes are in the lower kilobyte range (40-100 kB), linking against two additional libraries drastically increases the size. Nevertheless, the binaries remain in the kilobyte range.

For dynamic binaries and shared libraries, we have to embed the entire call graph as we need the information later on to determine the required syscalls. Table 9.2 shows the size increase for three shared libraries. In *libcrypto.so*, we observe a worst-case increase from 4.1 MB to 23 MB (460 %). The overhead also increases with the size of the binary as the call graph is larger for the larger codebase.

Binary. Table 9.1 shows the increase for Binalyzer. We opted to generate a binary that needs to be launched by ChestnutGenerator instead of rewriting the binary. Still, for simplicity, we embed the detected syscalls in the binary from where our wrapper extracts the information. As we embed only the numbers, the overhead in all 18 applications is less than 2 %. Binary rewriting incurs the overhead of the dependency on libchestnut and libseccomp, but this increase is again negligible beneath the other overheads.

Runtime Overhead and Seccomp

For the static approaches, the only overhead compared to manually crafted seccomp filters is the parsing of the syscall numbers. As this is done during application startup, it is a one-time overhead that depends on the number of rules that need to be set up. We investigate the overhead for setting up the application with the smallest (*true*) and largest (*Nginx*) number of syscalls based on Sourcalyzer. For Nginx, the setup time takes on average 9.92 ms ($\sigma_{\bar{x}} = 0.007$, $N = 10\,000$) while it only takes

0.58 ms ($\sigma_{\bar{x}} = 0.004$, $N = 10\,000$) for *true*. The remaining slowdown is then introduced by seccomp itself, which is unavoidable if a developer decides to use it for syscall filtering.

Dynamic Refinement Overhead

As a microbenchmark, we analyze the impact of Finalyzer on the syscall latency. We first benchmark the latency of the *getppid* syscall without Finalyzer in place 1 million times. The latency of *getppid* on our test system (Ubuntu 18.04.4, kernel 5.0.21-050021-generic) is 1358 ($\sigma_{\bar{x}} = 0.91$, $N = 1\,000\,000$) cycles. With Finalyzer, we observe an average latency of 17 103 ($\sigma_{\bar{x}} = 5.52$, $N = 1\,000\,000$) cycles, an increase of approximately 1160 %. While this increase seems large, it is intended as an optional step during development. Hence, we consider this less of a problem without impact on the released application.

6.3 Functional-Correctness Evaluation

Binaries sandboxed with Chestnut must be guaranteed to still work as intended. Related work [20] tested each application 100 times using various workloads. For a fair comparison, we perform the same tests. For applications where a test suite is available, we execute them to reach higher coverage, ensuring that we do not miss edge cases. Beyond previous work [11, 20], we evaluate code coverage to show that large parts of the application are executed. Finally, we perform a 6-month long test of Nginx sandboxed by Sourcalyzer.

In more detail, we first apply Chestnut to the binaries, cf. Table 9.1. Obtaining a sound ground truth of whether all syscalls are detected is infeasible and would require time-consuming formal proofs that are out-of-scope for this paper. Hence, we rely on executing the available test suites that should cover many of the different code paths available in the tested application. This is, for instance, possible for FFmpeg, memcached, redis, Nginx, and sqlite3. In other cases, we execute the binaries with different configurations to reach as many different code paths as possible [20]. We observed no crashes in applications sandboxed with Chestnut. Even if a syscall is missed in $\boldsymbol{\mathcal{P}1}$, $\boldsymbol{\mathcal{P}2}$ can be used to add it, ensuring correct functionality.

While this is not an exhaustive test, it can be assumed that test suites for large applications are designed for complete functionality coverage and thorough testing of critical components in particular. Based on the latter, it is a reasonable assumption that our test tests whether all syscalls in the core functionality of the application are found. To further substantiate this, we perform line and function coverage tests for a selection of applications, cf. Table 9.1. We perform these tests for FFmpeg (Lines: 59.3 %, Functions: 61.7 %), memcached (77 %, 91.9 %), and redis (77 %, 61.5 %). Additionally, the sqlite developers always maintain 100 % branch and 100 % MC/DC coverage [56]. While not perfect, the results indicate that large parts of the respective applications are executed and, to a certain degree, demonstrate that Chestnut does not impede them. In future work, we want to employ coverage-guided fuzzing to better estimate whether all required syscalls are found.

Programs using `fork+exec`, e.g., *git-diff*, exhibit the inherent problem of seccomp, namely that a child program inherits its parent's filters. If the child uses a syscall blocked by the parent, the child crashes. For such applications, $\mathcal{P}$**2** is necessary to ensure functionality. Out of the 18 tested applications, $\mathcal{P}$**2** was only necessary for two of them, namely *git-diff* and *git-log* as they performed syscalls blocked by their parent. After refining the filters using Finalyzer, both successfully completed their task.

Adding Missed Syscalls using $\mathcal{P}$2. We evaluate how many syscalls the static approaches miss. For Sourcalyzer, Finalyzer adds 4 syscalls to musl libc, which then propagate to applications if the corresponding function is used, e.g., *clone*. Table 9.1 shows how many syscalls are added in $\mathcal{P}$**2**. For Binalyzer and busybox, $\mathcal{P}$**2** only adds a syscall in *ls*. Sqlite3 misses the most as Finalyzer needs to add 13 syscalls. These missed syscalls are only a limitation of our proof-of-concept implementation, occurring in edge cases that can be handled in a full implementation. Hence, Chestnut also works without Finalyzer.

Long-Term Study using Nginx. To demonstrate the functional correctness of Chestnut, we performed a long-term study of 6 months using Nginx. In this test, we compiled a static version of Nginx using Sourcalyzer (105 allowed syscalls), which we then deployed to a real-world server to host a website. Within 6 months, the server handled ≈ 100 000 requests without ever triggering a seccomp violation. This shows that Sourcalyzer can infer all syscalls necessary for a successful operation on a real-world system.

6.4 Security Evaluation

To evaluate how Chestnut increases the security of sandboxed applications, we analyze how often dangerous syscalls, e.g., `exec`, are blocked (Section 6.4), the number of syscalls not blocked even though they are not used by the application (Section 6.4), the number of mitigated real-world exploits (Section 6.4), and how malicious SGX enclaves can be blocked (Section 6.4).

Blocking Dangerous Syscalls

Three of the more dangerous syscalls that Linux provides are the two syscalls in the `exec` group, *i.e.*, `execve` and `execveat`, and the `mprotect` syscall. With the `exec` syscalls available, an attacker can execute an arbitrary binary in the presence of an exploitable memory safety violation [6]. In fact, most libc versions even contain a ROP gadget that leverages the `exec` syscall to open a shell [29]. Hence, an attacker can execute a new program in the context of the current one. With `mprotect`, an attacker can modify the permissions of existing memory, *i.e.*, make it executable. While `mmap` can be used to map memory as executable, we did not consider it in our evaluation. We consider attacks not relying on syscalls [8] as out of scope.

Even with Chestnut, certain attacks are still possible, e.g., adding an ssh key if a privileged application is hijacked and the *open*/*write* syscalls are allowed. These attacks are also possible with Chestnut, but other attacks are blocked, improving the overall system security. Hence, Chestnut still improves the status quo.

Compiler. We evaluate how often Sourcalyzer can block `exec` and `mprotect` (Table 9.1). In busybox, we block the `exec` syscalls in 9 out of 10 cases and `mprotect` in all 10. Additionally, we also evaluated all the remaining busybox utilities and blocked `exec` in 313 out of 396 (79.0 %) of them and `mprotect` in all 396 (100 %). In Nginx, we cannot block `exec`, but we block `mprotect`. In the other applications, we can block neither of them as our compiler detects a potential call to a function that contains the respective syscalls.

Binary. Binalyzer blocks the `exec` syscalls in all busybox utilities, cf. Table 9.1, where Sourcalyzer could not block the `exec` syscall in the *env* utility. We manually verified that the syscalls are indeed not required. The `mprotect` analysis showed the opposite behavior as it is not blocked in any of the applications. For Nginx, memcached, mutool, and FFmpeg, we also block the `exec` syscalls without crashing the application, but not `mprotect`. We could not block either one of them in git, httpd, redis, and sqlite3. For git and the `exec` syscalls, the reason is that some of git commands rely on other applications, *i.e.*, the configured pager for commands like *diff* or *log.* The explanation of why we cannot block `mprotect` using Binalyzer is the point of time at which we start blocking syscalls. In Sourcalyzer, we block syscalls that are reachable only from the *main* and *exit* functions, while we block them from the start of the application in Binalyzer. Hence, we need to allow `mprotect` as it is required for setting up the application. In a full implementation, functions necessary for program startup can be removed from the analysis, e.g., the *mprotect* syscall.

Overapproximation of Syscalls

Chestnut can drastically reduce the number of syscalls available to an application (cf. Table 9.1). For our 18 tested applications, Nginx and httpd block the least number of syscalls with 106 being allowed. However, without Chestnut, 349 syscalls in Linux 5.0 would be available [35]. While Chestnut drastically reduces the attack surface, both Sourcalyzer and Binalyzer often allow more syscalls than necessary. In this section, we estimate how tight automatically generated syscall filters are that an automated approach can generate. We are the first to demonstrate this for an automated seccomp-filter generation tool. We apply this analysis also to related work [11] to compare the different approaches (cf. Section 6.5).

Setup. To evaluate our static components' overapproximation, we leverage the functionality of Finalyzer in libchestnut. This has the advantage over *strace* that we do not include syscalls that are needed for setting up the application, *i.e.*, we only log syscalls after the main entry point. We then either execute the applications test suite or execute the program with different arguments to trigger different code paths, *i.e.*, try to trigger as many of the existing syscalls as possible. The accuracy of our results depends on the code coverage of the respective test suites. As was the case in Section 6.3, we argue that despite this not being an exhaustive test, test suites typically cover at least the core functionality of the application and its critical components. We substantiate this claim with the code coverage metrics discussed in Section 6.3 The results show that large parts of the respective applications are executed, demonstrating that this is an adequate but not perfect approach to detect overapproximation. This depicts a first step to the measurement of the tightness of automatically generated filters.

Using the aforementioned approach, we obtain a list of syscalls that the evaluated program issued. We calculate that list's intersection with the allowed syscalls as detected by Sourcalyzer or Binalyzer. This gives us a list of syscalls that our approach allows but that are never executed by the application in our tests. If a syscall was triggered that our static approaches block, Finalyzer automatically refines the application's filter list.

Compiler. As Table 9.1 shows, overapproximation varies between different applications when using Sourcalyzer. In the busybox utilities, we observed the largest overapproximation for *env*, where only 20 % of the detected syscalls are actually used. For the larger applications, we observe the largest overapproximation in mutool, with only 26.23 % being used. Note that the results cannot be compared to Binalyzer due to different libc versions being used, *i.e.*, musl libc for applications compiled with Sourcalyzer and glibc in Binalyzer.

Binary. For the evaluation of Binalyzer, we slightly deviate from the outlined setup just to ease the evaluation. Instead of using libchestnut, we use the standalone implementation of Finalyzer. Hence, we observe a larger amount of syscalls as we also record syscalls executed during program startup, similar to *strace*.

In busybox, we overapproximate the most in the *true* utility, where only 37.5 % are being used. In the larger applications, we observe the lowest percentage of actually used syscalls in mutool, with only 21.74 % being used. Future work could extend the functional-correctness evaluation, estimating overapproximation using coverage-guided fuzzing.

Mitigating Real-World Exploits

For evaluating the effectiveness of Chestnut in mitigating real-world exploits, we assume an attacker that can either inject shellcode or mount a ROP attack [59] in one of our target applications. We define an exploit as successful if the attacker can exploit a kernel bug from the application context. These bugs either trigger a privilege escalation or result in a denial of service. As seccomp filters restrict the available syscalls, they reduce the attack surface of the kernel.

For the evaluation, we extract a list of 175 CVEs from the CVE database [61] that exploit syscalls on the x86_64 Linux kernel. From this list, we extract the necessary syscalls, resulting in a list of 231 malicious samples. The reason is that a CVE can be triggered by different syscalls that are independent of each other. As some syscalls have equivalent versions, we extend our list of samples to 320 by substituting the syscall numbers where applicable. We provide a list of these equivalent syscalls in Appendix A. As we want to show that Chestnut-sandboxed applications impede the exploitation of unpatched kernel vulnerabilities, we assume a kernel that is vulnerable to all these CVEs.

To determine the effectiveness of Chestnut, we cross-reference the syscall numbers from each sample with the ones we block in Table 9.1. If one of the syscalls required for the exploit is blocked, we determine that this application cannot trigger the exploit in the kernel, indicating that Chestnut increased the security of the system. We consider both the number of CVEs that we fully mitigate and the number of subvariants mitigated by Chestnut.

Compiler. With Sourcalyzer, we can fully mitigate 84.04 % of the CVEs and 89.42 % of the subvariants in the case of busybox. The reason for that is that the busybox utilities are rather small, allowing only a few syscalls. Even with larger applications, our compiler still increases the system's security, fully mitigating 38.08 % of CVEs and 56.53 % of the subvariants.

Binary. In busybox, Binalyzer mitigates 81.8 % of the CVEs fully and 87.9 % of the subvariants. In the larger binaries, Binalyzer can fully mitigate 36.38 % of the CVEs and 52 % of the subvariants.

Blocking Malicious SGX Enclaves

Intel SGX enclaves cannot directly execute any syscalls, but only use functionality provided by the host application. The host application can use syscalls to provide this functionality to the enclave. Schwarz et al. [52] presented a technique to execute arbitrary syscalls from an SGX enclave via a ROP attack on the host application. This allows malicious or hijacked enclaves to mount attacks on the kernel.

Weiser et al. [65] presented SGXJail as a generic defense for malicious enclaves, blocking them from executing arbitrary syscalls. Binalyzer achieves a similar goal without affecting the performance of required syscalls. For the evaluation, we used the public proof-of-concept exploit provided by Schwarz et al. [52]. The Intel SGX SDK currently does not support LLVM; hence, we can only evaluate Binalyzer. As enclaves cannot contain syscalls, Binalyzer only has to scan the host application and allow only syscalls legitimately used by the host application. Out of the 349 syscalls provided by Linux 5.0, 279 (79.9 %) are blocked, including `exec`. We verified that the benign functionality of the host and enclave is not impacted. As a result, the malicious (or hijacked) enclave cannot run arbitrary programs anymore, and the attack surface is drastically reduced.

6.5 Comparison to Other Approaches

Two recent approaches on automating seccomp filter generation [11, 20] were published after the start of our 6-month long-term case study of Chestnut on Nginx. In this section, we compare our work to these two approaches and discuss the differences.

Temporal Syscall Specialization. Ghavamnia et al. [20] propose an automated approach to detect the used syscalls during compilation. Their approach is limited to applications that can be split into an initialization and serving phase, *i.e.*, server applications. The idea is to detect syscalls used after the server's initialization phase, *i.e.*, the point in time where it starts handling requests. Thus, this approach is not directly applicable to applications that cannot be easily split into these two phases, potentially enabling attacks through browsers, malicious PDFs [16, 17], messengers [23, 54], and office applications [41]. We explicitly consider such applications in our approach as our threat model is broader and includes local attackers additionally to remote ones. Similar to Chestnut, they also extract a sufficiently precise call graph to be able to extract which syscalls are reachable by the application. Their approach relies on Andersen's points-to analysis, which is known to not scale with program size [2, 24]. We evaluated an orthogonal *has address taken* approach as is used by LLVM's CFI implementation. As this is already used for the CFI implementation of LLVM, we know that the resulting CFG is reasonably precise as otherwise applications that rely on software-based CFI would not work. Our approach achieves similar results in terms of detected syscalls as the more complex and slower approach used by Ghavamnia et al. [20]. In contrast to our approach, they require a multitude of tools for the compilation and link-time optimization that is not supported by every application. As neither Andersen's points-to nor our address taken approach can guarantee a complete CFG, we rely on the more practical address-taken algorithm. This choice significantly reduces the compile time. For instance, syscall extraction for Nginx using Andersen's algorithm shows an increase in compilation time from 1 min to 83 min (+8300 %) [20] compared to an increase of 7.9 s (+10.53 %) with Chestnut.

In summary, we improved the approach's performance significantly while maintaining accuracy and security. Additionally, our approach is applicable to a broader range of applications, including local applications that are commonly hijacked. We also provide an evaluation of the tightness of the resulting filters.

Sysfilter. A second approach, *sysfilter* [11] focuses on extracting syscalls from existing binaries. While sysfilter and Binalyzer share the same goal, the approaches differ in the used tools, *i.e.*, Binalyzer relies on the angr framework that already supports parts of what sysfilter manually implemented. Both approaches show similar success rates in mitigating exploits in their respective test sets.

Sysfilter provides no analysis of the approach's overapproximation, making it hard to estimate how tight the resulting syscall filters are. Hence, we perform such an analysis to show differences between the approaches. As we discussed in Sections 6.3 and 6.4, obtaining a ground truth is infeasible and would require computational intensive formal proofs. Hence, we need another source for a reliable baseline to which we can compare the results of the evaluation for Binalyzer and sysfilter.

To provide this baseline, we rely on the results of Sourcalyzer when generating a static binary, for two reasons: First, the compiler has the most information about the application as it needs to generate a functioning binary, *i.e.*, it needs to know which functions are actually required and called. The second reason is based on what a compiler like clang does when it generates the static binary that we use. When generating this binary, the compiler already removes all unnecessary functions, *i.e.*, functions that are never called and never have their address taken, from the binary. So the resulting binary only contains functions and their respective syscalls if the compiler determined a potential path to the respective function. Therefore, any syscall found by the two binary tools within the static binary can be reached and is necessary for the application to work correctly. This number may differ from the one detected by Sourcalyzer due to the inherent overapproximation of the function signature heuristic, *i.e.*, read and write have the same function signature, so if one is used, the other one is automatically included in the set. In this case, the syscall of a function is included even though the compiler removed the function's actual code. Nevertheless, we expect the numbers to be in a similar range.

In this evaluation, both sysfilter and Binalyzer work on the exact same static binaries. We ensured that the binary still contains the stack unwinding information (*.eh_frame*) and other necessary sections (*.init, .fini*) on which sysfilter relies for its precise disassembly. While sysfilter notes that one requirement is a PIC binary, we note that the additional tasks that sysfilter performs for PIC binaries, *i.e.*, relocations or checking the dynamic symbol table, are by the design of static binaries simply not necessary. Building the call graph does not depend on these steps either. In fact, for binary analysis tools like sysfilter and Binalyzer, a static binary

Table 9.3: The number of extracted syscalls by Sourcalyzer, Binalyzer, and the two modes of sysfilter.

Binary	Sourcalyzer	Binalyzer	sysfilter (vacuumed-fcg)	sysfilter (universal-fcg)
FFmpeg	63	53	18	53
busybox	163	144	15	152
Redis-server	85	74	12	74

can be considered the most straightforward use case as all information is already contained within the static binary.

We consider two different modes of sysfilter, *i.e.*, the default behavior that prunes the call graph based on a reachability analysis and the universal approach that assumes that every function is reachable by every other function. As the binary is compiled statically, we expect that both modes produce the same result as only functions that are reachable from the main entry point are included. We show the result of this analysis in Table 9.3.

As our analysis shows, the assumption that both modes of sysfilter produce the same result does not hold as the pruning-based mode significantly underapproximates in all three evaluated binaries. The low number of detected syscalls hints at some mistake in the pruning algorithm as the number is too low for such complex applications. In two out of three binaries, Binalyzer and sysfilter using the universal approach produce the exact same result while the third binary only shows a small difference of 8 syscalls. In this case, the difference to Sourcalyzer is within an expected range due to the overapproximation of Sourcalyzer. This is not true for the pruning-based approach of sysfilter as the difference is too large, and the number of detected syscalls is lower than the number of syscalls that are actually used (cf. Table 9.1). Interestingly, the universal-fcg implementation of sysfilter also supports our observation that a PIC binary is not a requirement for these types of binary analysis tools as it produces similar results to Binalyzer, contradicting the statement by its developers. Nevertheless, there is still a difference in the operation between the universal-fcg approach of sysfilter and Binalyzer as the latter achieves this result by not assuming that every function is reachable by every other function. Instead, it still builds a correct call graph and derives the information from it, which fails for the vacuum-fcg approach of sysfilter.

We investigated the low number of syscalls found in the pruning-based approach of sysfilter. This analysis showed that during the pruning, the main function is removed from the set of reachable functions, which results in the whole application being removed from the analysis. We leave the analysis of whether this is purely an implementation bug or a hint that this is a general problem in the approach for future work as this is out of scope for this paper.

7 Discussion

Limitations and Future Work. Fast and reliable points-to analysis with limited overapproximation is still an unsolved problem [24]. In some cases, we also exhibit the opposite effect in *angr* that it is not able to detect the call target of an indirect call, hence missing a potentially reachable syscall. In contrast to previous work, we evaluate this problem by measuring the code coverage on real-world code examples. Future work may extend our analysis with coverage-guided fuzzing to obtain more precise estimates for the overapproximation of automated seccomp filter generation tools.

Future work may investigate the possibility of extending the syscall filtering with argument tracking. While detecting constant syscall arguments is possible, the precise propagation of this information throughout the call graph is not trivial. Solving this problem would allow restricting syscalls further, e.g., only allow certain hardcoded paths for `exec` or limit possible permissions passed to `mprotect`, e.g., no executable permissions. Finally, one limitation is the performance of seccomp [25, 62] imposed by the underlying system. Since this is not a weakness of Chestnut itself, we consider improving the performance of seccomp out of scope for this paper.

Related Work. Several related works also discuss the problem of automating sandboxing mechanisms, e.g., reducing the attack surface of applications by removing unused code. One of the first approaches for library debloating is based on removing non-imported functions from a shared library during load time [42]. This approach has been further improved by removing all unused functions from shared libraries during load time by extending the compiler and the loader [46]. Agadakos et al. [1] proposed a binary-level approach for library debloating, based on function boundary detection and dependency identification to identify and erase unused functions. Davidson et al. [10] analyzed the entire software stack for web applications to create specialized libraries based on the requirements for PHP code and the server binaries. *Shredder* [39] instruments binaries to restrict arguments to system APIs to a predefined allowlist. Another approach is to apply data dependency analysis for fine-grained customization of static libraries [55].

More closely related to our work is the approach by Ghavamnia et al. [20]. However, their approach suffers from a significantly higher execution time for the analysis during compilation while achieving a comparable accuracy in detecting syscalls. Wagner and Dean [63] propose a static approach to build an IDS that uses a similar approach to Sourcalyzer for pointer analysis to extract a model of expected application behavior. In general, several papers have proposed static analysis of syscalls for anomaly detection and IDS [18]. Rajagopalan et al. [48] propose to replace syscalls with authenticated syscalls that specify a policy and provide a cryptographic MAC that guarantees the integrity of the syscall. sysfilter [11] uses the Egalito framework to statically extract the syscalls from the binary similar to Binalyzer, but has a strong requirement on PIC binaries which Binalyzer does not.

Other approaches reduce the attack surface using training to identify the unused code sections, e.g., Ghaffarinia and Hamlen [19]. Without access to the source code, training and heuristics can be used to identify and remove unnecessary basic blocks [45].

Previous work focused mostly on C/C++ software with few solutions for software in other languages. For Java, one approach uses static code analysis to remove unused classes and methods [28]. For PHP, Azad et al. [3] proposed a framework using dynamic analysis to remove superfluous features.

8 Conclusion

Chestnut is an automated approach to block unused syscalls in applications, identified using static analysis and an optional dynamic refinement. The compiler-based approach is up to factor 73 faster than previous work without any loss in accuracy. On the binary level, our approach extends over previous ones by also applying to non-PIC binaries and thus a broader set of applications. Chestnut increases platform security without manual effort as shown in our evaluation of correctness and overapproximation, using test suites, code coverage, and a 6-month long-term evaluation. Chestnut blocks more than 82.5 % of all syscalls and 61 % corresponding kernel CVEs from these applications.

Acknowledgments

This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM, Intel, and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

A List of equivalent syscalls

In this appendix, we provide a list of equivalent syscalls (cf. Table 9.4).

Table 9.4: Syscalls and their equivalents.

Syscall	Equivalents
munlockall	munlock
listxattr	llistxattr, flistxattr
epoll_create	epoll_create1
mlockall	mlock, mlock2
execve	execveat
recvfrom	recvmsg, recvmmsg
writev	pwritev
mknod	mknodat
open	openat
accept	accept4
getdents	getdents64
sendto	sendmmsg, sendmsg
getxattr	fgetxattr, lgetxattr
rename	renameat, rename2
epoll_ctl	epoll_ctl_old

References

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. "Nibbler: debloating binary shared libraries." In: *ACSAC*. 2019.

[2] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language." PhD thesis. University of Copenhagen, 1994.

[3] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. "Less is More: Quantifying the Security Benefits of Debloating Web Applications." In: *USENIX Security Symposium*. 2019.

[4] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack." In: *AsiaCCS*. 2011.

[5] Erik Bosman and Herbert Bos. "Framing Signals - A Return to Portable Shellcode." In: *S&P*. 2014.

[6] Nicholas Carlini and David A. Wagner. "ROP is Still Dangerous: Breaking Modern Defenses." In: *USENIX Security Symposium*. 2014.

[7] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns." In: *CCS*. 2010.

[8] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. "Non-Control-Data Attacks Are Realistic Threats." In: *USENIX Security Symposium*. 2005.

[9] Chromium. *Linux Sandboxing*. URL: `https://chromium.googlesource.com/chromium/src/+/0e94f26e8/docs/linux_sandboxing.md`.

[10] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. "Towards automated application-specific software stacks." In: *ESORICS*. 2019.

[11] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. "sysfilter: Automated System Call Filtering for Commodity Software." In: *RAID*. 2020.

[12] David Drysdale. *Anatomy of a system call, part 2*. 2014. URL: `https://lwn.net/Articles/604515/`.

[13] David Drysdale. *How programs get run: ELF binaries.* 2015. URL: https://lwn.net/Articles/631631/.

[14] Jake Edge. *A library for seccomp filters.* 2012. URL: https://lwn.net/Articles/494252/.

[15] Jake Edge. *A seccomp overview.* 2015. URL: https://lwn.net/Articles/656307/.

[16] Jose Miguel Esparza. *Quick analysis of the CVE-2013-2729 obfuscated exploits.* 2014.

[17] Jose Miguel Esparza. *Static analysis of a CVE-2011-2462 PDF exploit.* 2012.

[18] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. “Formalizing sensitivity in static analysis for intrusion detection.” In: *S&P*. 2004.

[19] Masoud Ghaffarinia and Kevin W Hamlen. “Binary Control-Flow Trimming.” In: *CCS*. 2019.

[20] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. “Temporal System Call Specialization for Attack Surface Reduction.” In: *USENIX Security Symposium.* 2020.

[21] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. “Out of Control: Overcoming Control-Flow Integrity.” In: *S&P*. 2014.

[22] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. “A secure environment for untrusted helper applications: Confining the wily hacker.” In: *USENIX Security Symposium.* 1996.

[23] Samuel Groß. *Remote iPhone Exploitation Part 1: Poking Memory via iMessage and CVE-2019-8641.* 2020.

[24] Michael Hind. “Pointer analysis: Haven’t we solved this problem yet?” In: *PASTE*. 2001.

[25] Tom Hromatka. *seccomp and libseccomp performance improvements.* 2018.

[26] Google Inc. *Sandbox2.* 2019. URL: https://developers.google.com/sandboxed-api/docs/sandbox2/overview.

[27] Google Inc. *Seccomp filter in Android O.* 2017. URL: https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html.

[28] Yufei Jiang, Dinghao Wu, and Peng Liu. “JRed: Program Customization and Bloatware Mitigation Based on Static Analysis.” In: *COMPSAC*. 2016.

[29] Mateusz Jurczyk and Gynvael Coldwind. “Permissions overview.” In: *Insomni'hack* (2015).

[30] Vasileios Kemerlis. “Protecting Commodity Operating Systems through Strong Kernel Isolation.” PhD thesis. Columbia University, 2015.

[31] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. “ret2dir: Rethinking kernel isolation.” In: *USENIX Security Symposium*. 2014.

[32] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. “kGuard: Lightweight Kernel Protection against Return-to-User Attacks.” In: *USENIX Security Symposium*. 2012.

[33] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *IEEE / ACM International Symposium on Code Generation and Optimization – CGO*. 2004.

[34] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. “A measurement study on Linux container security: Attacks and countermeasures.” In: *ACSAC*. 2018.

[35] Linux. *64-bit system call numbers and entry vectors*. 2019. URL: https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl.

[36] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. “Directed symbolic execution.” In: *International Static Analysis Symposium*. 2011.

[37] W. S. McPhee. “Operating system integrity in OS/VS2.” In: *IBM Systems Journal* (1974).

[38] Matt Miller. “Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape.” In: *Bluehat IL* (2019).

[39] Shachee Mishra and Michalis Polychronakis. “Shredder: Breaking exploits through API specialization.” In: *ACSAC*. 2018.

[40] Mozilla. *Seccomp filter in Android O*. 2016. URL: https://wiki.mozilla.org/Security/Sandbox/Seccomp.

[41] Jens Müller, Fabian Ising, Christian Mainka, Vladislav Mladenov, Sebastian Schinzel, and Jörg Schwenk. "Office Document Security and Privacy." In: *WOOT*. 2020.

[42] Collin Mulliner and Matthias Neugschwandtner. "Breaking Payloads with Runtime Code Stripping and Image Freezing." In: *Black-Hat USA* (2015).

[43] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting Fine Grain Isolation in the Firefox Renderer." In: *USENIX Security Symposium*. 2020.

[44] Vassilis Prevelakis and Diomidis Spinellis. "Sandboxing Applications." In: *USENIX ATC*. 2001.

[45] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. "RAZOR: A Framework for Post-deployment Software Debloating." In: *USENIX Security Symposium*. 2019.

[46] Anh Quach, Aravind Prakash, and Lok Yan. "Debloating Software through Piece-Wise Compilation and Loading." In: *USENIX Security Symposium*. 2018.

[47] Nguyen Anh Quynh. "Capstone: Next-gen disassembly framework." In: *Black Hat USA* (2014).

[48] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. "Authenticated system calls." In: *DSN*. 2005.

[49] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *USENIX Security Symposium*. 2019.

[50] *sandboxing:Firejail Security Sandbox*. 2018. URL: `https://firejail.wordpress.com/`.

[51] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *S&P*. 2015.

[52] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *DIMVA*. 2019.

[53] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In: *CCS*. 2007.

[54] Natalie Silvanovich. *Exploiting Android Messengers with WebRTC: Part 1.* 2020.

[55] Linhai Song and Xinyu Xing. "Fine-grained library customization." In: *SALAD.* 2018.

[56] SQLite. *How SQLite Is Tested.* 2020.

[57] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time." In: *POPL.* 1996.

[58] Nicolas Sylvain. *A new approach to browser security: the Google Chrome Sandbox.* 2008.

[59] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *S&P.* 2013.

[60] Robert Tarjan. "Depth-first search and linear graph algorithms." In: *SIAM journal on computing* (1972).

[61] The MITRE Corporation. *Common Vulnerabilities and Exposures.* URL: `http://cve.mitre.org/`.

[62] Tizen. *Security:Seccomp.* 2018. URL: `https://wiki.tizen.org/Security:Seccomp`.

[63] David Wagner and R Dean. "Intrusion detection via static analysis." In: *S&P.* 2000.

[64] Fish Wang and Yan Shoshitaishvili. "Angr - The Next Generation of Binary Analysis." In: *2017 IEEE Cybersecurity Development (SecDev).* 2017.

[65] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGXJail: Defeating Enclave Malware via Confinement." In: *RAID.* 2019.

[66] Mozilla Wiki. *Security/Sandbox.* 2019. URL: `https://wiki.mozilla.org/Security/Sandbox`.

10
Domain Page-Table Isolation

Publication Data

Canella, Claudio and Kogler, Andreas and Giner, Lukas and Gruss, Daniel and Schwarz, Michael. "Domain Page-Table Isolation." In: *arXiv:2111.10876* (2021)

Contributions

Main author.

Domain Page-Table Isolation

Claudio Canella[1], Andreas Kogler[1], Lukas Giner[1]
Daniel Gruss[1], Michael Schwarz[2]

[1] Graz University of Technology, Austria [2] CISPA Helmholtz Center for Information Security, Germany

Abstract

Modern applications often consist of different security domains, requiring mutual isolation. While several solutions exist, most rely on specialized hardware, hardware extensions, or less-efficient software instrumentation of the application.

In this paper, we propose Domain Page-Table Isolation (DPTI), a novel mechanism for hardware-enforced security domains that can be readily used on commodity off-the-shelf CPUs. DPTI uses two novel techniques for dynamic, time-limited changes to the memory isolation at security-critical points, called memory *freezing* and *stashing*. We demonstrate the versatility and efficacy of DPTI in two scenarios: First, DPTI *freezes* or *stashes* memory to support more fine-grained syscall filtering than state-of-the-art seccomp. With the provided memory-safety guarantees, DPTI can securely support deep argument filtering, such as string comparisons. Second, DPTI *freezes* or *stashes* memory to efficiently confine potentially untrusted SGX enclaves, outperforming existing solutions by 14.6 %-22 % while providing the same security guarantees. Our results show that DPTI is a viable mechanism to isolate domains within applications using existing mechanisms available on CPUs, without relying on special hardware instructions or extensions.

1 Introduction

Memory isolation is a vital primitive to ensure the security of modern systems. However, this isolation is not only necessary between different applications, but it becomes more and more important within applications as well. Often, applications consist of multiple security domains that should be isolated from each other. Such in-process isolation ensures that vulnerabilities in one domain cannot easily affect a different domain. The separation of user and kernel space is a well-established isolation mechanism; trusted execution environments (TEEs) such as Intel SGX or ARM

TrustZone are more recent examples. The interface and memory model of TEEs share similarities with the kernel-user-space boundary, where the enclave is trusted, but the rest of the application is not. Other new ISA extensions, such as Intel MPK [43], allow setting up multiple security domains within applications to efficiently control access permissions for different memory ranges. In all these cases, the isolation is configured in software and enforced by the hardware.

However, while the isolation must provide strong security guarantees, it is still necessary that the domains can communicate with each other. This interface between low-privilege and higher-privilege domains is often exploited. For instance, the syscall interface between unprivileged applications and the kernel is often used for privilege-escalation attacks [89]. Hence, state-of-the-art isolation techniques such as seccomp [21] provide developers with the ability to block syscalls, reducing the kernel's attack surface in case of exploited applications. While this works well for blocking syscalls, the performance overhead can increase linearly with the length of the blocklist [38, 90]. Still, while securing this interface is important, there is currently no option to create sophisticated filters. For example, it is not possible to block the `exec` syscall for all but a hard-coded set of applications. This lack of support for deep argument filtering is not merely a matter of implementation but stems from the current filtering-mechanism design [22, 23].

Existing isolation solutions are often limited to very specific scenarios, require specialized hardware, or source- or binary-level instrumentation of the application. For example, the hardware isolation of TEEs can only be applied to enclaves. However, in practice, it is infeasible to run all software in enclaves. Enclaves have limitations, such as the inability to interact with the kernel directly, and are only available on a subset of CPUs. Their asymmetric trust model was exploited in recent work [84, 97], requiring mitigations with high overheads or special hardware features. Proposals for domain isolation require hardware extensions or modifications as well [27, 79, 97]. Moreover, hardware extensions such as Intel MPK are not broadly available, e.g., MPK is only available on Xeon Scalable CPUs and some 10th-generation Intel CPUs. Furthermore, an attacker can disable Intel MPK security domains after gaining code execution within the protected application, as preventing this is not in scope for Intel MPK [91]. Hence, while these proposed isolation mechanisms are effective, they *cannot be used on commodity CPUs.*

In this paper, we introduce the notion of dynamic, time-limited changes to the memory isolation at security-critical points, a security-domain mechanism we call **DPTI** (Domain Page-Table Isolation). As DPTI-based solutions rely only on existing hardware-enforced memory protection of the memory-management unit (MMU), we can implement memory isolation for sandboxing and domain-isolation scenarios on commodity, off-the-shelf CPUs. Still, DPTI has the same page granularity as other approaches that require dedicated hardware or hardware extensions [71, 79, 91, 97]. Similar to KPTI [36, 64], DPTI uses efficient page-table modifications to create security domains that are temporarily inaccessible from another domain. However, in contrast to KPTI, DPTI does not have to maintain a second set of page tables for every process, and it has to solve additional challenges such as multiple mappings to one physical page. In the remainder of this work, we use *DPTI* as shorthand for both the concept and its proof-of-concept implementations.

We evaluate DPTI on Intel CPUs ranging from 2015 (for SGX support) to 2018. However, as DPTI only relies on the MMU, it runs on all CPUs with virtual-memory support, in contrast to previous works [27, 37, 54, 68, 71, 79, 91]. Contrary to software fault isolation (SFI), we do not use any instrumentation of the application. With DPTI, it is even possible to add stricter syscall filters to existing (legacy) applications via a wrapper application.

We demonstrate the versatility of DPTI in two scenarios, enhanced syscall filtering and SGX protection domains, leading to high-performance, feature-rich solutions. In these, DPTI relies on two new techniques: memory *freezing* and *stashing*.

In our first application of DPTI, we present syscall filtering that is more fine-grained than seccomp. DPTI temporarily *freezes* or *stashes* the memory ranges of complex syscall parameters from the untrusted user-space application on a syscall, preventing modification while the kernel performs the syscall. Such a design requires solving multiple challenges, including process creation and replacement, multithreading support, alias mappings, and alternative memory-access interfaces. However, by solving these challenges, DPTI-based filtering can evaluate complex parameters, such as strings, without introducing time-of-check-to-time-of-use (TOCTOU) vulnerabilities into the syscall interface [22, 23, 82]. In a microbenchmark, non-argument-inspecting DPTI-based filters incur an overhead of 28.7 % compared to seccomp's 16.9 %.

In the second use case, DPTI improves performance and security of SGX enclave confinement [84, 97]. DPTI prevents the rewriting of host memory from SGX, again by *stashing* or *freezing* host memory, making it inaccessible to enclaves. However, this is not trivial: switching to and from an enclave requires some host pages. We address this challenge by securing SGX entries and exits with a special code bridge page. Thus, we mitigate host impersonation for arbitrary syscall execution by a malicious or hijacked SGX enclave [84]. While DPTI outperforms existing SGX enclave confinement solutions [97] by 14.6 %-22 %, the most important advantage is that DPTI works on commodity systems without hardware changes and maintains full compatibility with existing enclaves.

DPTI is an efficient and easily implementable solution for isolating security domains in various scenarios. Due to its software-only implementation without specific ISA-extension dependencies [27, 37, 54, 68, 71, 79, 91], it can be readily used on commodity, off-the-shelf hardware.

To summarize, we make the following contributions:

1. We introduce DPTI,[1] an MMU-based isolation mechanism inspired by page-table isolation, enabling fine-grained security domains and security policies.
2. We use DPTI to implement extended syscall filtering, enabling sophisticated argument-inspecting filter rules not supported by seccomp or isolation:AppArmor.
3. We show that DPTI can replace existing SGX enclave confinement solutions, isolating unmodified untrusted enclaves on commodity hardware.
4. We thoroughly evaluate the security and performance of DPTI and show that it provides higher performance and security than previous solutions in both case studies.

Outline. Section 2 discusses background. Section 3 describes the high-level idea and threat model. Section 4 details the two case studies for DPTI. Section 5 evaluates DPTI's security and performance. Section 6 discusses future and related work. Section 7 concludes.

2 Background

2.1 Sandboxing

Sandboxing provides an extra layer of security by strictly controlling the resources an application can access [34, 73]. In many cases, sandboxing is a last line of defense that assumes that the sandboxed application was already exploited. Thus, a sandbox should drastically limit the impact of an exploit. Existing sandboxes typically restrict access to the network and file system and limit the available syscalls. Sandboxes are widespread

[1]Prototype can be found at `https://github.com/domain-isolation/DPTI`.

in browsers [76, 98, 99] and on mobile operating systems [4, 42]. Linux provides sandboxing via the SELinux [100] and isolation:AppArmor [44] frameworks.

2.2 Linux Seccomp

A vital part of sandboxes is the ability to restrict the syscall interface. The syscall interface provides functionality from the operating system to user-space applications. An exploit with unrestricted access to the syscall interface can read, write, and execute files on the system. In the worst case, the syscall interface itself is exploitable, leading to a privilege escalation [47–49]. Hence, sandboxes try to minimize the number of exposed syscalls to a minimum required for the application to work. Secure Computing (seccomp) [21] is a syscall filter that is integrated into the Linux kernel. An application can specify allowed syscalls, and the kernel blocks all others. In addition to the syscall itself, seccomp can define filters for integer parameters. Filters based on the content of strings or structures are not supported, as seccomp cannot dereference parameters [22, 23].

Due to the complexity of limiting the syscall interface, several approaches to automatically generate syscall filters have been published recently [11, 20, 30, 31]. The common approach is to dynamically or statically analyze an application to detect all required syscalls.

2.3 Runtime Attacks

Many security vulnerabilities are caused by memory safety violations. Typical memory safety violations, such as buffer overflows, enable attackers to modify an application in an unintended way [89]. In many cases, attackers try to overwrite code pointers to hijack the control flow. On modern systems, data is typically not executable, and thus an attacker cannot inject so-called shellcode into an exploited application [89]. Thus, exploits reuse existing program parts and divert the control flow to so-called *gadgets* [70]. Shacham [86] generalized such control-flow-hijacking attacks using gadgets as return-oriented programming (ROP). By chaining multiple gadgets, it is possible to build arbitrary exploits. In addition to such control-flow-hijacking that overwrite pointers [12, 33, 57, 80, 86], data-only attacks [45, 77] can also violate memory safety.

Race conditions are a type of vulnerability where a data structure is accessed in parallel, and the actual order of accesses affects the correctness of the program. A special type of race condition are time-of-check-to-time-of-use (TOCTOU) vulnerabilities. A TOCTOU bug exists if a memory location is accessed multiple times, and an attacker can manipulate the data between the access. Such an exploitable TOCTOU bug is called a double-fetch bug [82, 95]. Double-fetch bugs are especially dangerous in the syscall interface as they are hard to detect [95] and often easy to exploit [82].

2.4 Memory Isolation

Memory isolation has been used to provide security for a long time. Segmentation and paging are two of the most well-known approaches to isolate memory on x86. The operating system configures the memory, and the CPU then enforces this configuration. While segmentation is no longer used to enforce access permissions on x86-64, security researchers continued to propose information hiding techniques based on this legacy feature [5, 56, 63]. Modern systems rely on paging to translate the virtual addresses of a process to physical addresses through the help of page tables. Page tables also contain access permissions which determine whether a page is read-only or user-space accessible.

Other memory isolation techniques use newer hardware features such as Extended Page Tables (EPT) [54, 62], Software Guard Extensions (SGX) [26], and Memory Protection Keys (MPK) [37, 54, 71, 79, 91]. EPTs facilitate memory virtualization, while SGX allows protecting code and data even from a compromised operating system. MPK introduces a new register containing a protection key and allows a developer to associate memory with such a key. The key itself is stored in the page tables, and an access is only allowed if the current register key matches the one stored in the page table.

2.5 SGX

Intel SGX [43] is a trusted execution environment (TEE) introduced with the Intel Skylake microarchitecture in 2015. Using this instruction-set extension, applications can be divided into an untrusted and a trusted part. The trusted part, the so-called enclave, is integrity- and confidentiality-protected by the CPU. Thus, even in the case of a malicious operating system, the content of the enclave is protected. Enclaves are hosted by ordinary, untrusted applications. Both enclave and untrusted application run in the same virtual address space. While the hardware prevents outside access to the address range of the enclave, the enclave can access all

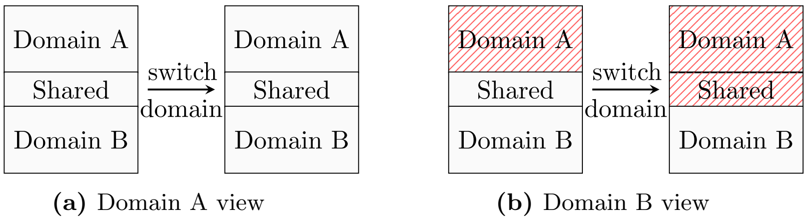


Figure 10.1: The shared address space and the different views of it. No change for Domain A upon switching while parts are stashed or frozen (red pattern) for Domain B with DPTI.

memory of the host application. The SGX threat model assumes that the entire software stack is malicious. However, there is no consideration that an enclave might be malicious. This asymmetry enables enclave malware that can impersonate the host application to execute arbitrary syscalls [84, 97]. Enclaves themselves cannot execute syscalls. Instead, enclaves communicate with the host application via ECALLs and OCALLs. After loading the enclave, the host application can call secure enclave functions using ECALLs via a call gate, similar to syscalls. If the enclave wants to use functionality from the operating system, such as a syscall, it has to use an OCALL to call into the host application. The ECALL/OCALL interface is defined by the developer at compile time.

3 High-Level Idea & Threat Model

On a high level, DPTI provides software-defined memory protection for fine-grained sandboxing policies, e.g., deep argument filtering. There are two variants for this protection: read-only protection or entirely inaccessible. We refer to the former as *DPTI-Freeze* and to the latter as *DPTI-Stash*. Either variant guarantees that the memory cannot be modified by an untrusted domain.

Figure 10.1 shows the idea of DPTI. Two security domains share one virtual address space and use a dedicated memory region for communication, *i.e.*, for passing data across the security boundary. In many cases, while there is a defined memory region for the communication, Domain A has access to the entire Domain B. As both domains can execute code in parallel, Domain A cannot ensure the atomicity of the accessed data in this memory region. Hence, Domain B could modify the data while Domain A accesses it, potentially leading to data corruption or TOCTOU vulnerabilities. DPTI solves this problem by protecting the content of this memory region as long as Domain A accesses it. We achieve this protection by modifying page-table entries of the respective memory region in one of two ways. With DPTI-Stash, we modify the user-accessible bit in the page table, temporarily making the region accessible only to the kernel. Contrary, DPTI-Freeze modifies the read-write bit, preventing write access while allowing read accesses to the region.

While we envision that DPTI can be applied in many different ways, we present two case studies that help illustrate its behavior and demonstrate the provided security. The scenario shown in Figure 10.1 is common for operating systems, where the kernel (Domain A) is typically mapped into the upper half of the virtual memory of every user application (Domain B). In Section 4.1, we present a case study with these two security domains. Another scenario is the execution of SGX enclaves, which share the address space with the host application. In Section 4.2, we present a case study in which DPTI ensures that a malicious or exploited enclave (Domain B) cannot modify data of the host application (Domain A).

3.1 Threat Model

For DPTI, we assume two different security domains in one application, such as user and kernel space, or untrusted application and trusted enclave. DPTI provides additional sandboxing to existing isolation mechanisms and hence assumes that the used isolation mechanisms are reliable. In particular, DPTI presumes that the MMU-based isolation cannot be circumvented architecturally. Transient-execution attacks circumventing security domains do not undermine the security of DPTI. Meltdown [60] shows that the US bit can be circumvented during transient execution, but it only allows reading the page. Foreshadow [92] circumvents the present bit for reading enclave memory. However, DPTI does not protect confidentiality. Hence, Meltdown or Foreshadow do not cause a security problem. Similarly, while Spectre v1.1 [52], Store-to-Leak [81], and LVI [92] show that values can be transiently written to inaccessible pages, applications can issue a serializing instruction before reading from the DPTI-isolated memory region to avoid reading transient data. Fault attacks [50, 51, 69] are out of scope.

4 Case Studies

In this section, we present two case studies showing how DPTI can efficiently isolate different domains. First, we apply our method to facilitate efficient and complex syscall argument filtering that is not vulnerable to TOCTOU vulnerabilities (Section 4.1). Second, we apply DPTI to SGX, preventing attacks from untrusted or exploited enclaves [84] (Section 4.2).

4.1 Enhanced Syscall Filtering

To restrict syscalls, Linux provides developers with seccomp. While seccomp allows filtering static, primitive syscall arguments, it does not support complex arguments such as strings or structs due to TOCTOU vulnerabilities [22, 23]. Other approaches are based on syscall interposition, where the syscall is delegated to another process which decides whether a syscall is allowed [24, 29, 32, 74, 93]. Additionally to the cost of delegating the syscall, these approaches suffer from the same TOCTOU problem as seccomp [28, 96].

We focus on deep argument filtering without introducing TOCTOU vulnerabilities while maintaining the performance of seccomp in typical scenarios. By applying DPTI, we remove an attacker's ability to modify the syscall argument while the kernel performs operations, such as checks, on it. Specifically, we prevent the attacker from rewriting a pointer or modifying a string while the kernel inspects that syscall argument. On a high level, we identify the page used by the syscall argument and make it read-only or fully inaccessible by modifying specific bits in the page-table entry. The argument is then compared to the allowed values for the respective syscall, and if the argument check succeeds, the syscall is executed. By design, this approach cannot suffer from TOCTOU vulnerabilities. Such an approach requires solving several challenges such as process creation and replacement, multithreading, alias mappings, and other memory-access interfaces.

By providing the means for deep argument filtering, we improve the system's security by further limiting the post-exploitation impact of a memory safety vulnerability as we can better restrict syscalls such as *exec*. Other seccomp functionality, *i.e.*, filtering syscalls without argument checking and checking of static integer arguments, is supported too. We refer to this as *simple filtering.*

Threat Model

Beyond the threat model of DPTI (cf. Section 3.1), we assume that the application itself is not malicious but exploitable, e.g., due to a memory-safety violation, allowing an attacker to gain arbitrary code execution within the application. Similarly, the kernel is considered to be trusted

```
filter_info_t *filters = dpti_create_filters();
dpti_add_filter_rule(filters, SYS_read);
dpti_add_filter_rule_string(filters, SYS_write,
    1, EQ, "teststring");
dpti_install_filters(filters);

```

Listing 4.1: Example of a simple allow/reject filter for *read* and a complex filter requiring deep argument filtering on the first argument of the *write* syscall using our support library.

but potentially exploitable. We assume that the post-exploitation phase targets the system and requires syscalls, e.g., to gain kernel privileges. Contrary to previous work [11, 20, 31], we can filter syscalls related to file operations by inspecting complex data types. Our approach is orthogonal to other defenses such as CFI, ASLR, NX, or canary-based protections and improves security if these were circumvented.

Implementation

Our enhanced sandboxing consists of two parts: a kernel part that performs the actual task of filtering syscalls and the respective arguments, and a support library that can be linked to the application. The support library generates the individual syscall filters and installs them. As such, it is a similar high-level abstraction as *libseccomp* is for seccomp. However, filter generation in libseccomp is much more complex due to the usage of cBPF. During the seccomp filter installation, the cBPF is verified and converted to eBPF, which introduces additional overhead. As our filters are not expressed in BPF, the setup is significantly easier. For our proof-of-concept implementation, we implement our filtering entirely as a standalone kernel module. For a production-ready system, the filtering should be implemented directly in the kernel.

A prerequisite for all syscall-filtering approaches is to be able to intercept syscalls. Fortunately, entry points for syscalls are clearly defined, making it easy to intercept all syscalls. This includes both legacy 32-bit syscalls as well as 64-bit syscalls.

Setup For the setup, the user-space application sends the filters to the kernel module. The filters are defined in a high-level representation similar to libseccomp, as illustrated in Listing 4.1. The module then copies the filters into kernel memory. Preventing TOCTOU vulnerabilities is

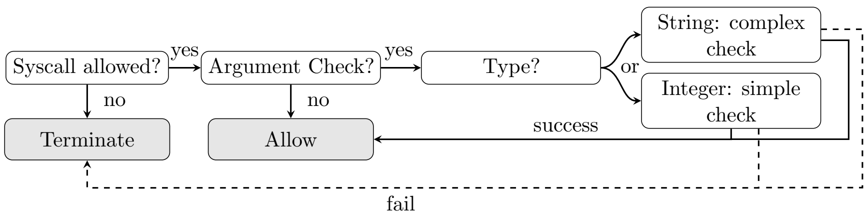


Figure 10.2: Overview of the 3 DPTI syscall-filtering cases.

not necessary at this point, as the application is still considered benign. Otherwise, if the application was already exploited, an attacker can either manipulate the filters or skip their installation entirely, rendering the filtering useless. This is in line with other filtering approaches, such as seccomp. After copying the filters, the application is considered sandboxed, and no further updates are possible. A full implementation can consider allowing further restrictions of the filters, similar to seccomp.

Filtering Syscalls Every requested syscall is delegated to our generic syscall function inside the module. If the syscall originates from a sandboxed application, the syscall function uses the syscall number to retrieve the syscall's filter rules. Contrary to seccomp, checking the filter for a specific syscall does not require scanning all syscall filters. Instead, it is a simple array access; the lookup time for a filter does not depend on its position within the set of filters [38], reducing the asymptotic runtime from linear to constant.

Figure 10.2 illustrates the 3 cases when checking a syscall:

(1) Syscall not allowed: The syscall fails the initial check whether it is allowed, making it unnecessary to check potential argument filters, and the application can be terminated.

(2) Syscall allowed, no argument filtering requested: The syscall passes the initial check, *i.e.*, the application explicitly allowed the syscall, requiring the check of potential argument filters. However, the syscall is allowed unconditionally as no filters are present. Thus, DPTI calls the original syscall function with the same arguments. We refer to these first two cases as simple allow and reject filters.

(3) Syscall allowed, argument filtering requested: The syscall itself is allowed, making it necessary to check the installed argument filters. As the ABI [66] defines that syscalls can have up to 6 arguments, every argument must be checked against possible filters. We iterate over all potential arguments, checking for each whether a filter is defined. Checks

then distinguish between primitive and complex data types. Primitive data types fully fit into the 64-bit register used as syscall argument. Such data types include, e.g., integers or booleans. They do not require any special handling as they are already copied to the kernel. Hence, they are not vulnerable to TOCTOU and are similarly handled as they are in seccomp.

Complex data types cannot be contained fully in the 64-bit register and are typically pointers to either strings or structures. These pointers point to user-space memory and can thus be modified by a concurrently running thread during the syscall. If the modification happens between applying the filter and executing the syscall, the filter is effectively rendered useless. We focus on checking string arguments, as strings are widely used as syscall parameters. As string parameters are especially used with file names [82], they are an excellent target for sandboxing. Our method is not limited to strings but also applies to structures. For these types of arguments, we have to ensure that they cannot be modified between applying the filter and the execution of the syscall. Thus, we first resolve the page table mappings of the argument pointer. At this point, we can differentiate between our two variants of DPTI, providing different behavior and security guarantees.

DPTI-Stash: Our first variant relies on modifying the US bit, essentially making the page a kernel page and fully inaccessible to user space. To achieve this, we clear the US bit and flush the corresponding TLB entry from all necessary CPU cores. If the executable bit is set, we temporarily clear it as well so that no user-controlled executable pages are brought into the kernel. When an attacker tries to modify the argument in a different thread, the MMU enforces the protection by generating a page fault due to the privilege violation. Our module can now perform a simple string comparison with all developer-provided values without having to consider TOCTOU vulnerabilities. If the check succeeds, the syscall is allowed, and the original syscall is executed. Otherwise, the application is killed due to the violation. Before returning from the syscall, page-table entries are restored if the page is currently not used in another, concurrently executed, syscall where string filters are used.

DPTI-Freeze: The second variant relies on the RW bit, making the page write-protected. By clearing it in the kernel data structures as well as in the page-table entry, the page is read-only. A malicious thread can only read from it but no longer modify its content, preventing TOCTOU vulnerabilities. The syscall filtering is equivalent to DPTI-Stash. After the syscall is done executing, we restore the previously cleared bits. This requires some additional tracking of pages to determine whether this step is necessary. To prevent a concurrent thread from issuing an *mprotect* syscall to make the page writable again during the execution of another syscall, it is necessary to stall *mprotect* with write permissions on a page that had its write permissions cleared by DPTI-Freeze. We can use the same stalling functionality that is already available in the Linux kernel, e.g., for swapping pages.

Special Cases

The approach outlined above works for most syscalls and can be handled by our generic syscall function. Only creating (*fork*, *clone*) and destroying (*exit*) processes need additional treatment as well replacing an existing process with *exec*. Linux creates a new task for a newly created process or thread, including a new process identifier. Hence, for the `fork` and `clone` syscall, we increase the reference counter of the filters and share them with the newly created process or thread. When calling `exec` after `fork`, we ensure that the copy-on-write syscall-parameter page is made accessible again for the other process(es) after the `exec` succeeds. In our proof of concept, we simply trigger a copy-on-write fault on this page before manipulating the page-table entry.

A second special case is the handling of the *exit* and *exit_group* syscalls, as they clean up our filters. When a sandboxed application executes one such syscall, we decrease the reference counter of the filters by the number of threads in the group and mark the thread group as no longer sandboxed. If the reference counter reaches 0, *i.e.*, no thread or forked process needs the filters anymore, we free the memory before terminating the process.

Multithreading

Due to the page size granularity, it is likely that the string is not the only content of the associated memory page. While read accesses from a concurrent thread are no problem for DPTI-Freeze, write accesses to the page lead to a page fault. For DPTI-Stash, both read and write accesses trigger a page fault.

In the case of DPTI-Stash, the page-fault handler can easily determine that the access would normally be legal, as kernel pages are not found within the user-space address range. We experimentally verified this by scanning the pages of all running user-space processes on an Ubuntu 18.04 machine. If the page-fault handler determines that the faulting page is not an actual kernel page, it stalls the offending thread until the syscall has finished and the page is again available to user space. The kernel already needs to stall processes for swapping; this functionality can be reused. We discuss potential stalling times in more detail in Section 5.1.

For DPTI-Freeze, we already track all pages that are modified for deep argument filtering, allowing us to easily differentiate a potentially legal write from one to a page that was never writable. If the page fault occurs on an access to such a tracked page, we again stall the offending thread until the original page access rights are restored. As stated before, read accesses cause no problem.

Alias Mappings

Although multiple *writable* mappings to the same physical address are rare (cf. Section 5.1), DPTI has to consider such scenarios as well. With code execution, an attacker can create two virtual address mappings to the same physical page, *i.e.*, by using *mmap* and other shared-memory-related syscalls. While the virtual address mapping used by the syscall is stashed or frozen, the content of the physical page can be modified via the second, unaltered mapping. This once again facilitates a TOCTOU vulnerability. Hence, it is necessary that alias memory mappings are tracked, and if such a mapping is used during the argument check, all other mappings to the same page must be modified too.

A probe on the page-fault handler ensures that all possible methods of creating an alias mapping are detected. For each physical page, we store all virtual addresses mapping this page, including meta information such as the permission and the process, independent of whether the mapping is in the same or a different process. If a shared page is used during deep argument filtering, all mappings to it are modified in the previously outlined way. We discuss the performance impact and how frequently such mappings are used by real-world software in Section 5.1.

Modifying Memory through /proc/self/mem

Finally, an attacker can potentially circumvent our TOCTOU-free argument filtering via the */proc/self/mem* interface, which allows directly modifying the content of a physical page, ignoring missing write permissions. To protect against this possibility, it is necessary to either restrict the access to this file entirely, *i.e.*, make it read-only, or to the offset that corresponds to the physical page currently used in a syscall. In our proof-of-concept, we rely on the former by installing a probe on the *mem_write* function that prevents the write if the application is currently sandboxed. As the read-only approach is already done by REL 5 and 6, it is reasonable to assume that this does not impair the functionality of applications [65]. Moreover, to understand whether this interface is used in open-source software, we looked at all code on GitHub that opens this file for writing. After filtering out all PoCs for exploits, we only found one project (`rr`) that uses this functionality. Hence, while a production-ready implementation can implement more fine-grained control, it is questionable whether write access to this file is necessary at all.

4.2 SGX-Protection Domain

The classical SGX threat model is asymmetric, *i.e.*, it enforces strong protection guarantees for enclaves but does not protect the user application from the loaded enclave. Enclaves can lead to various security concerns, as neither the operating system nor the user application can verify the code executed inside the enclave, as intended by the design of SGX. One limitation of SGX is that it does not allow the enclave to execute code outside the enclave boundaries [16]. However, Schwarz et al. [84] showed malware inside enclaves can simply manipulate the user applications' stack to execute arbitrary code outside the enclave. To balance the protection guarantees for the user application, SGXJail [97] proposed to isolate enclaves by moving them to a separate process with strict syscall filters, or by relying on hardware modifications, *i.e.*, memory protection keys (Intel MPK).

Our DPTI-based SGX protection domain extends the memory isolation guarantees of SGXJail. However, with DPTI there is no increased overhead of process isolation, and no need for MPKs or for changing the SGX specification. DPTI replaces a thread's page-table mapping and isolates the user application's memory during the execution of an enclave. As most available pages, *i.e.*, the entire user-space application, have to be isolated, switching the mapping is faster than iterating over all mapped pages.

The isolated mapping only contains the original mappings for the loaded enclave, a single additional code page mapping, and a few data page mappings from the user application. The single code-page mapping is used to enter the enclave, while the additional data-page mappings are used to transfer data from and to the enclave. Upon enclave exit, our protection verifies that the enclave did not tamper with the provided pages or registers and restores the non-isolated mapping for normal execution.

All other pages are either protected by DPTI-Freeze or DPTI-Stash, depending on what security guarantees the user application enforces (cf. Section 5.2).

In contrast to our proposed syscall filtering, the SGX protection domain implements DPTI-Stash by not mapping a page instead of clearing the US bit. Therefore, the enclave can no longer access any of the non-isolated pages as these pages are no longer mapped inside the thread's virtual address space. When using DPTI-Freeze, all of the non-isolated pages are mapped as read-only pages in the isolated mapping, allowing the enclave to read but not modify them.

To perform fast transitions, we construct the isolated mapping once and then reuse and update it upon entering. We redesign the data flow between the enclave and the user application to remove unnecessary copying of the data passed to an enclave, leading to a high-performance SGX protection domain with only an overhead of 9.9 % to 24.0 % for the worst-case scenario (cf. Section 5.2).

Threat Model

For the SGX protection domain, we reverse the classical SGX threat model. We assume that the loaded enclave is untrusted and potentially malicious, whereas the app loading the enclave is trusted. We assume that the enclave tries to read or modify data of the user application, e.g., to mount a ROP attack [84]. The trusted user application does not use seccomp filtering to restrict syscalls. We assume that the enclave code is unknown and imposes no restrictions on limiting the interfaces between the enclave and the user application in the sense of ECALLs and OCALLs.

Implementation

The SGX protection domain requires no changes to existing enclaves. All parts are implemented in the driver and the Untrusted RunTime System (URTS).

Isolated Mapping For the isolated mapping, we extend the SGX driver, which provides the necessary information about pages associated with a given enclave. We create one isolated mapping per enclave to ensure that multiple threads within enclaves are supported while colluding enclaves cannot circumvent the isolation. The enclave's isolated mapping is extended by a Code Bridge Page (CBP) and Data Bridge Pages (DBPs). The CBP contains the `EENTER` instruction used to enter enclaves inside

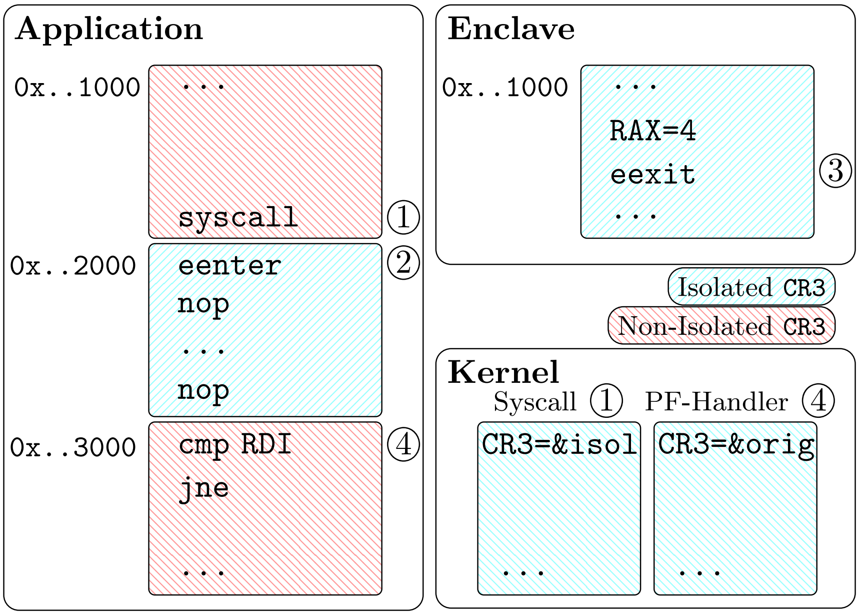


Figure 10.3: Control-flow transition of entering and leaving DPTI-based SGX protection domain.

the URTS. It is shared amongst all enclaves as the URTS library is only loaded once inside the user application. The Data Bridge Pages (DBPs) are used to transfer data between the enclave and the user application. To facilitate fast switches between two mappings, we only exchange the actual hardware mapping inside the `CR3` register and retain the VMA structures of the original mapping.

Entering and Exiting Isolation Figure 10.3 depicts the control flow for switching protection domains. When a thread executes an IOCTL syscall into the SGX driver ①, the driver switches the calling thread's virtual address mappings to the isolated address mapping. DPTI ensures that the syscall instruction is the last instruction on the page before the CBP. Hence, the page containing the syscall is not part of the isolated mapping, as returning from the syscall does not require this page. The enclave is entered and executed in isolation via the `EENTER` instruction ②. To exit an enclave, the `EEXIT` instruction is invoked from inside the enclave, with the RAX register set to 4 and the return address specified in RBX ③. While DPTI ensures that the only executable mapping is the CBP, an enclave can potentially still return to an arbitrary location inside the CBP. Hence, the CBP must not contain exploitable code. As x86 relies on variable-length opcode and does not enforce instruction alignment, we opted for not placing any functional code on the CBP. Instead, we use the CBP as a trampoline by filling it with single-byte NOP instructions. DPTI then leverages the page fault triggered by the first instruction after the CBP ④. The SGX driver catches the page fault and verifies that the fault originates from the first instruction after the CBP. If this is the case, the driver switches the address space back to the non-isolated mapping.

Interrupts A special case is an asynchronous enclave exit due to an interrupt or fault. The CPU stores the enclave state, hands control to the interrupt or fault handler, and then resumes the enclave using an `ENCLU` instruction. As this instruction has to be mapped, we place it on the CBP. We verified that a misaligned jump into this instruction cannot be exploited.

Data Bridges and Stack DPTI must be aware of the pages used for data exchange between the enclave and host application, hence the dedicated DBPs. These DBPs are defined when setting up the isolation. Note that DPTI works on page granularity. Thus, the user must ensure that the data marked as DBPs is aligned and padded to the page boundary to ensure no additional data is exposed.

Before entering the isolation, DPTI aligns the stack pointer to a page boundary, allowing isolation of pages above the thread's stack pointer. The stack pages below the aligned stack pointer are accessible from within the isolation, as the SGX-SDK uses the user application's stack to pass arguments to OCALLs. However, these pages do not contain any data the application uses after the enclave returns. If destroyed stack frames were used for sensitive data, they should be manually zeroed before entering the enclave. When returning to the user application, DPTI verifies that the stack and the base pointer match the values stored when entering the enclave, ensuring that the enclave did not alter the stack (cf. Section 5.2).

Page Fault Handling Page faults on EPC pages are allowed, as they can be lazily mapped. Page faults on host-application pages from isolated threads are only allowed on the first instruction of the page after the CBP, and the signal handler. Otherwise, the enclave attempted to access data not marked as DBP or jumped to an isolated code page with the `EEXIT` instruction, leading to a termination of the enclave. Note that DPTI can distinguish a fault generated inside an enclave from a malicious attempt to execute the signal handler directly.

Optimizations

We propose an optimization that can be used if the application does not require the *stat* syscall, or if an untrusted application cannot abuse this syscall. Instead of switching from the SGX domain to the non-isolated domain using a relatively slow page fault (cf. Sections 5.2 and 6.1), we can rely on a syscall.

While placing a syscall instruction on the CBP could potentially lead to exploitation of this instruction, we utilize the nature of the `EEXIT` instruction. This instruction requires the `RAX` register to be set to 4. Hence, an attacker can only execute the *stat* syscall. This syscall typically does not allow any control over an application, nor does it allow to leak valuable information. To communicate with the driver, we hardcode the necessary register values before the syscall. Thus, the enclave can only execute the syscall communicating with the driver or the *stat* syscall by jumping to the syscall instruction.

5 Evaluation

In this section, we evaluate DPTI based on our two use cases, enhanced syscall filtering (Section 4.1) and SGX-protection domain (Section 4.2), in terms of security and performance.

5.1 Enhanced Syscall Filtering

We evaluate the performance (Section 5.1) and security (Section 5.1) of both variants of DPTI, clearly showing advantages and disadvantages of DPTI-Stash and DPTI-Freeze.

Performance Evaluation

We first evaluate the performance when executing a single syscall with reject filters for other syscalls. Second, we analyze the overall performance on various real-world applications with simple allow/reject filters. We compare the results to an unsandboxed version and one using seccomp. Hence, we only evaluate cases that are supported by seccomp. Third, we analyze the performance impact of our newly proposed method for deep argument filtering. Finally, we perform an in-depth analysis of the various

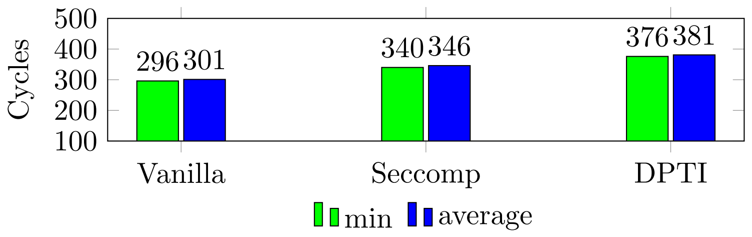


Figure 10.4: Syscall latency with no, seccomp-, and DPTI-based filtering over 1 million iterations of the *getppid* syscall.

steps discussed in Section 4.1, *i.e.*, resolving page tables, bit manipulation, TLB flush, and the string comparison, further substantiating the previous analysis results.

Setup The micro- and macrobenchmark are performed on an Intel i7-4790K running Ubuntu 21.04 with kernel version 5.13 at a stable frequency of 4 GHz. All other experiments are performed on an Intel Skylake i7-6600U running Ubuntu 18.04.1 with kernel version 5.4.0-72-generic at a stable frequency of 2.6 GHz. For seccomp, we rely on *libseccomp* (2.5.1) to generate and install the filters, and enable the BPF JIT compiler.

Simple Syscall-filtering Benchmark We microbenchmark the execution time when filtering a simple syscall (*getppid*). We compare the results to an unsandboxed (vanilla) application and one using seccomp. *getppid* is a fast syscall without side effects used by previous work and kernel developers [9, 11, 38]. As this experiment does not involve deep argument filtering, there is no difference between DPTI-Stash and DPTI-Freeze.

The sandboxed benchmark application allows 8 and blocks 341 syscalls using simple allow/reject filters. We measure the average execution time of the syscall in cycles over 1 000 000 executions. We show the results of this experiment in Figure 10.4.

The average execution of a vanilla *getppid* syscall requires 296 cycles and 346 cycles (+16.9 %) with seccomp active. With DPTI, the average execution time is 381 cycles (+28.7 %). Hence, seccomp outperforms DPTI in our microbenchmark. This is to be expected as seccomp is a highly-optimized component of the Linux kernel, while DPTI is not optimized for speed and is implemented as a kernel module. We expect that an in-kernel implementation can achieve at least similar performance as seccomp.

Table 10.1: The results of our runtime performance evaluation of unsandboxed, seccomp- and DPTI-sandboxed applications without deep argument filtering. Overhead shows the percentage overhead compared to our baseline, *i.e.*, an unsandboxed version of the respective application. Sample sizes differs for various applications due to the difference in required runtime.

	Software	Sample Size	Normal	Seccomp	DPTI
			Time / SEM	Time (Overhead) / SEM	Time (Overhead) / SEM
busybox	diff	10 000	0.0015 s / 3.374×10^{-7}	0.0073 s (386.67 %) / 5.493×10^{-7}	0.002 s (33.33 %) / 4.075×10^{-7}
	ls	10 000	0.0015 s / 3.352×10^{-7}	0.0073 s (386.67 %) / 6.466×10^{-7}	0.0021 s (40.0 %) / 4.414×10^{-7}
	head	10 000	0.0015 s / 3.605×10^{-7}	0.0073 s (386.67 %) / 6.040×10^{-7}	0.002 s (33.33 %) / 3.538×10^{-7}
	dmesg	10 000	0.0015 s / 3.355×10^{-7}	0.0074 s (393.33 %) / 2.201×10^{-6}	0.002 s (33.33 %) / 3.163×10^{-7}
	true	10 000	0.0015 s / 3.677×10^{-7}	0.0073 s (386.67 %) / 5.821×10^{-7}	0.002 s (33.33 %) / 3.567×10^{-7}
	cat	10 000	0.0015 s / 3.593×10^{-7}	0.0073 s (386.67 %) / 8.072×10^{-7}	0.002 s (33.33 %) / 6.047×10^{-7}
	env	10 000	0.0015 s / 3.541×10^{-7}	0.0073 s (386.67 %) / 6.257×10^{-7}	0.002 s (33.33 %) / 3.880×10^{-7}
	grep	10 000	0.0017 s / 3.557×10^{-7}	0.0075 s (341.18 %) / 6.451×10^{-7}	0.0022 s (29.41 %) / 4.009×10^{-7}
	pwd	10 000	0.0015 s / 3.678×10^{-7}	0.0073 s (386.67 %) / 6.440×10^{-7}	0.002 s (33.33 %) / 3.642×10^{-7}
git	diff	10 000	0.004 s / 1.421×10^{-6}	0.0077 s (92.5 %) / 3.475×10^{-6}	0.0061 s (52.5 %) / 2.216×10^{-6}
	status	10 000	0.004 s / 1.010×10^{-6}	0.0076 s (90.0 %) / 3.392×10^{-6}	0.006 s (50.0 %) / 1.821×10^{-6}
ffmpeg	convert	100	9.5795 s / 3.180×10^{-2}	9.5817 s (0.02 %) / 3.311×10^{-2}	9.6609 s (0.85 %) / 3.727×10^{-2}
	change	100	9.3627 s / 3.419×10^{-2}	9.494 s (1.4 %) / 2.336×10^{-2}	9.4412 s (0.84 %) / 3.438×10^{-2}
	extract	100	6.3684 s / 2.638×10^{-2}	6.4448 s (1.2 %) / 2.119×10^{-2}	6.4039 s (0.56 %) / 2.043×10^{-2}
	crop	100	6.7168 s / 1.248×10^{-2}	6.8011 s (1.26 %) / 2.026×10^{-2}	6.8055 s (1.32 %) / 2.013×10^{-2}
	info	10 000	0.0062 s / 1.419×10^{-6}	0.0409 s (559.68 %) / 2.015×10^{-6}	0.0405 s (553.23 %) / 7.732×10^{-6}
	remove	100	9.2065 s / 2.441×10^{-2}	9.2666 s (0.65 %) / 2.462×10^{-2}	9.3499 s (1.56 %) / 2.486×10^{-2}

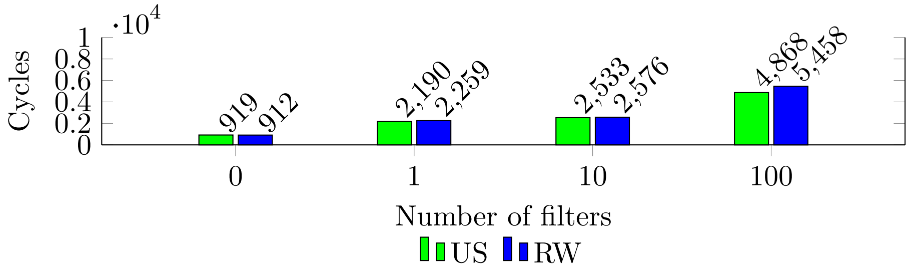
Real-world Application Benchmark Microbenchmarks show the specific effects of DPTI on syscalls, but do not allow reasoning about real-world performance impact. As no deep argument filtering is used, there is no difference between DPTI-Stash and DPTI-Freeze.

To determine the application's syscalls, we extend existing automated approaches [11, 20, 31] to support DPTI. Our baseline is a vanilla version of the respective application. We show the result of our evaluation in Table 10.1. The commands used for each application are provided in Table 10.4 in Appendix B.

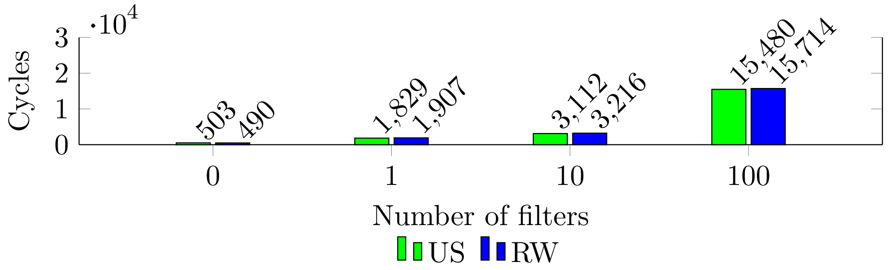
We first consider the results of small applications and simple tasks of large ones, *i.e.*, busybox, git, the ffmpeg info command. For these applications, the overhead appears huge, ranging between 29 % and 554 % for DPTI. Still, for all cases, the overhead is lower than for seccomp where we measured an overhead between 90 % and 560 %. For more complex tasks in ffmpeg, the overhead never exceeds 1.56 % with DPTI, and 1.4 % for seccomp. The reason why seccomp performs worse in the benchmarks of small applications is due to the larger overhead in setting up the more complex cBPF filters, which must be verified and converted to eBPF during the setup. In large applications, this complex setup does not impact the overall performance. The benchmark shows that even an unoptimized kernel module version of DPTI can achieve similar performance to the highly optimized seccomp.

Deep Argument Filtering To evaluate the performance of DPTI for memory isolation, we use both variants for deep argument filtering. We consider two simple syscalls, *openat* and *write*. For both syscalls, we measure the average execution time of the syscall, including deep argument filtering, over 100 000 invocations. As a syscall can allow multiple strings, we consider different number of strings in the filter, placing the correct one at its end, *i.e.*, for n strings in the filter we require n comparisons. As a baseline, we measure the syscall execution time without argument checking.

Figure 10.5 shows the results of this experiment. Deep argument filtering has a non-negligible performance overhead: With DPTI-Stash, a single string check increases the execution time of the *openat* syscall from 920 to 2038 cycles. This increase is mostly due to the one-time overhead of isolating the memory region. With 10 string checks, the execution time is only slightly higher (2351 cycles). In all cases, DPTI-Stash and DPTI-Freeze perform similarly.



(a) openat



(b) write

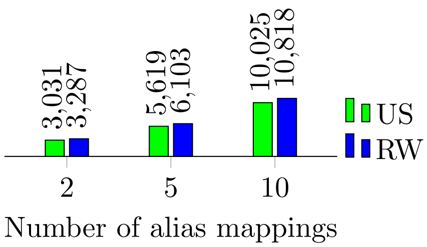
Figure 10.5: Cycles required for string filtering for *openat* and *write* syscalls using DPTI, either using the US or RW bit. The x-axis indicates the position of the allowed string within the filter for the syscall, *i.e.*, the number of strings that need to be checked until a match occurs with the current argument.

In-Depth Analysis of Filtering Components To analyze the overhead further, we perform an in-depth analysis to determine the overhead of each individual step, *i.e.*, resolving page tables, bit manipulation, TLB flush, and the actual string comparison.

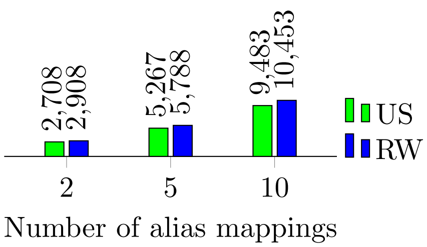
We instrument DPTI-Stash and DPTI-Freeze to measure the required cycles for each step. We use the same syscalls as in the previous experiment and filter outliers—detected using the modified z-score—by replacing them with the median. Table 10.2 shows the results of the evaluation. The majority of the overhead is introduced by the page-table flush, which is necessary to perform twice. The string check differs between the two evaluated syscalls due to different string lengths used in the evaluation. Clearing the additional bit in the kernel data structures for DPTI-Freeze has no impact. The execution time of all steps roughly sums up to the time difference shown for the case with no and one string filter in Figure 10.5. More efficient caching of page-table translations can potentially further improve the performance.

Table 10.2: Cycle amounts of the individual components of our deep argument filtering over 100 000 executions of the respective syscall. Outliers were replaced with the median.

Syscall			Resolve PT	Manipulate PT	Flush PT	Check
openat	Stash	cycles	86	27	458	40
		SEM	0.0886	0.0028	0.0533	0.0024
	Freeze	cycles	81	28	473	46
		SEM	0.0939	0.0041	0.0683	0.0028
write	Stash	cycles	45	27	396	137
		SEM	0.1751	0.0028	0.0442	0.0442
	Freeze	cycles	49	27	422	185
		SEM	0.2662	0.0034	0.1067	0.1480



(a) openat



(b) write

Figure 10.6: Cycles required for string filtering with alias mappings for *openat* and *write* syscalls using DPTI. The x-axis indicates the number of alias mappings for the filtered syscall argument, including the mapping used in the syscall itself.

Alias Mappings To evaluate the overhead of alias mappings, we build on our previous benchmark with the *openat* and *write* syscalls. Each syscall uses a single string filter, and we vary the number of alias mappings to the syscall argument, *i.e.*, 2, 5, and 10 mappings. Each syscall is executed 100 000 times and the average execution time is taken. Figure 10.6 shows the result of this evaluation.

Unsurprisingly, the average execution time of a syscall increases with the number of alias mappings that have to be modified. In combination with the results of the previous in-depth analysis (Table 10.2) and our performance analysis of non-alias mappings in syscalls (Figure 10.5), it is clear that the majority of the overhead is due to the additional, necessary TLB flushes. Improving the performance of TLB flushes automatically improves the performance of our filtering [58]. Note that this overhead only materializes in the case of at least 2 alias mappings to the same page; otherwise the overhead shown in Figure 10.5 applies.

We also investigate the performance of our tracking itself, which shows that adding a new mapping on average adds 1222 cycles (N=100 000, +47.5 %) to the page fault. Most of this overhead, though, is not due to the tracking itself but a limitation of our proof-of-concept implementation as we need to probe (kretprobe) the page-fault handler. A full implementation can perform the tracking by extending it directly, removing on average 638 cycles of overhead from the page fault. Hence, the overhead of the tracking itself is 24.8 %. Additionally, a full implementation can modify existing kernel data structures for a more efficient tracking mechanism.

As we observe a more significant overhead when an alias mapping is used in a syscall, we investigate whether such mappings are frequently used in syscalls in real-world applications. For this evaluation, we rely on the applications shown in Table 10.1, *i.e.*, git, busybox, and ffmpeg, as well as visudo (Section 5.1). We track the number of alias mappings created over the execution of each one of our commands and whether such a mapping is then used in a string filter. This analysis revealed that not a single alias mapping is created and hence not used in a string filter. For these applications, there is no additional overhead. We leave a broader analysis of alias mappings in real-world applications for future work.

Security Evaluation

For the security evaluation, we show that clearing the US- or the RW-bit prevents data modification. We show that we can perform the deep argument check without interference from user space, *i.e.*, that our approach does not suffer from TOCTOU vulnerabilities. Finally, we discuss the security implications of bringing user-controlled pages into the kernel.

Modifying Bits in Page Tables To ensure that the MMU-based isolation is reliable, we create a simple program that allocates a page of memory and then accesses it, observing, as expected, no crash. In the case of DPTI-Stash, we then use PTEditor [83] to clear the US-bit and access it again. As expected, the second access now results in a crash due to a privilege boundary violation. For DPTI-Freeze, we modified PTEditor such that it clears the VM_WRITE bit in the vm_flags of the associated vm_area_struct. While reading from the page still works, storing data to the page results in a crash due to the violation of the write protection.

Hence, clearing the respective bits prevents another thread from modifying the data once the protection bits have been set appropriately.

Eliminating TOCTOU While the previous experiment shows that the underlying principle works, we verify that it can prevent the exploitation of TOCTOU vulnerabilities. In this experiment, we try to re-direct an *openat* syscall such that a wrong file is opened. We create a multi-threaded application that uses filters that only allow opening the file *file1*. Thread 1 tries to open the specified file and print its content. Simultaneously, Thread 2 tries to modify the filename while the syscall is executed. In a vanilla or seccomp-based version, the syscall continues and opens the wrong file. With DPTI, the page containing the syscall is unmodifiable for the user. Hence, Thread 2 triggers a segfault due to the illegal access, and the thread is stalled, mitigating the attack.

User-space Pages in Kernel Space When using DPTI-Stash, our enhanced filtering effectively brings a user-controlled page into the kernel space (cf. Section 4.1). To ensure that this cannot be exploited to inject arbitrary code into the kernel, DPTI-Stash first verifies that the page is not marked as executable. As DPTI-Stash already has to modify the page-table entry for isolating the page, it can additionally clear the executable bit. Hence, as long as the page resides in the kernel space, it is not executable anymore. A thread cannot access the page while it is isolated; hence, removing the execute permission does not require any additional handling. After the syscall is done, DPTI-Stash can simply restore the executable bit. Thus, an attacker cannot exploit DPTI-Stash to inject executable pages into the kernel. Note that the kernel always has to deactivate SMAP to access the syscall parameters, enabling the access to the entire user-space memory during this time. Hence, temporarily bringing one data page into the kernel during a syscall does not increase the attack surface.

We consider the possibility of DPTI-Stash potentially weakening or breaking KASLR due to the user-space page being brought into the kernel. As it is only a data page, it can only be used as a stack page in a code-reuse attack where the attacker places the return addresses of the ROP gadgets. This already requires a KASLR break as the *ret* instruction can only return to absolute addresses. Hence, we can conclude that DPTI-Stash does not weaken or break KASLR.

Visudo

To demonstrate how deep argument filtering can improve the security of the system, we harden the *visudo* application with DPTI. *visudo* is used to edit the *sudoers* file with automatic validity checks using an editor from a pre-defined set of editors. It can also be used to open other files than the *sudoers* file by using a command-line switch. Interestingly, the tool's manpage already mentions that this behavior can be a security hole:

"if visudo is configured with the --with-env-editor option or the env_editor Default variable is set in sudoers, visudo will use any the editor defines by VISUAL or EDITOR. [..] this can be a security hole since it allows the user to execute any program they wish simply by setting VISUAL or EDITOR."

For the evaluation, we restrict *visudo* to only allow the *vi* editor to open the *sudoers* files as well as files necessary for *vi* to work, e.g., configuration files. Hence, in addition to allowing syscalls required for *visudo*, we also need to allow syscalls that *vi* requires. In total, we generate 52 simple syscall filter rules without any argument filters. We allow the *execve* syscall with a deep argument filter on */usr/bin/vi*, hence an invocation with a different editor is prevented. Naturally, the list of pre-defined editors can be extended, and the restriction on *vi* is simply done to ease our proof-of-concept implementation. We add a deep argument filter on the *openat* syscall such that it only allows opening the *sudoers* file and all strictly necessary library and file dependencies of *visudo* and *vi*. This results in 47 deep argument filters for *openat*. In total, we define 100 filters. Although the simple syscall filters are supported by seccomp, seccomp cannot restrict access to files and editors as filters based on string comparisons are unsupported.

To demonstrate that the resulting *visudo* binary is still able to perform its job, we use it to modify the *sudoers* file using *vi*. When doing that, we did not observe a single crash, and the task completes successfully. Still, when we try to override the editor or try to manipulate another file, our module detects that the specified filter rules are being violated and kills the application. Hence, DPTI can be used to close the previously mentioned security hole.

We evaluate the performance of our protected version of *visudo* over 10 000 executions. For the evaluation, we measure the performance of the non-interactive verification mode of *visudo*. Nevertheless, the filtering is the same as in the interactive modification mode, *i.e.*, filters for the *openat* syscall are still checked. The vanilla version requires, on average,

Table 10.3: Comparison of the two variants of DPTI. Write and read possible only consider the ability of the user-space application to read or write, not the kernel.

Approach	Modified Bit	Read possible	Write Possible	Tracking
Stash	US	✘	✘	✘
Freeze	RW / VM_WRITE	✓	✘	✓

12.8 ms, DPTI-Stash 13.2 ms, and DPTI-Freeze 13.0 ms, demonstrating a negligible overhead.

Comparison DPTI-Stash and DPTI-Freeze

While both variants of DPTI solve the problem of deep argument filtering, they differ in what MMU mechanism enforces the protection. As the used bits differ in their semantics, the two variants exhibit different properties, which we discuss in more detail now. The results of our comparison are presented in Table 10.3.

DPTI-Freeze requires manipulation of bits in two different locations, but has the advantage that a user-space thread can read data from the modified page, which is not possible with DPTI-Stash. For DPTI-Freeze, we need to explicitly track the previous permissions of the modified page to restore the correct state after the syscall completes, which is not necessary for DPTI-Stash. However, the additional bit to modify does not introduce any overhead (Table 10.2).

In both cases, writes are not possible from user space. Additionally, DPTI-Freeze blocks writes from the kernel. While this does not provide security benefits, it can be disadvantageous if the kernel wants to write to the isolated page. However, we have not encountered such a case in our tests.

5.2 SGX-Protection Domain

We evaluate the performance (Section 5.2) and the security guarantees (Section 5.2) of the SGX protection domain, including the more optimized version from Section 4.2.

Performance

As page table mappings are enforced by hardware, no additional performance overhead is observed during the regular execution of an enclave.

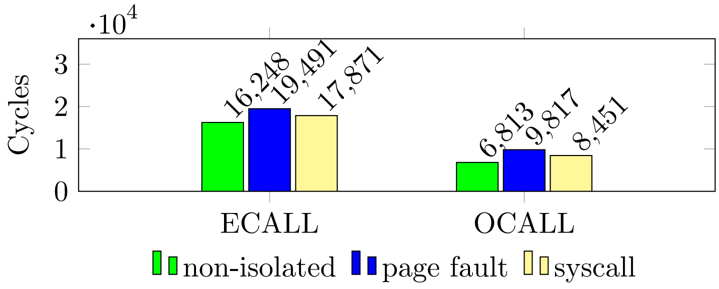


Figure 10.7: Microbenchmark results for ECALLs and OCALLs for a non-isolated enclave, one using a page fault for ending isolation, and one using a syscall instead of the page fault.

The protection domain alters the page tables during enclave enter and exit, affecting the latency of enclave transitions only. We evaluate the overhead using *isolation:sgxbench* [75]. For our evaluation, we restrict ourselves to the *empty ECALL* and *empty OCALL* benchmark of *isolation:sgxbench* as these model the worst-case scenario for our protection. These benchmarks perform ECALLs or OCALLs without any code in the function.

Setup We perform our benchmarks on an *Intel Core i5-8265U* CPU with a fixed frequency of 2.9 GHz running Ubuntu 20.04 with Linux kernel 5.8.0. We compare DPTI-Stash to a reference URTS SGX-PSW library as baseline, and execute each benchmark 2 000 000 times. We also evaluate the performance improvements of the slightly less secure, optimized version (cf. Section 4.2).

Results Figure 10.7 shows the benchmark results. We observe a latency increase of 19.9 % for ECALLs and 44.0 % for OCALLs. When using the optimized version using the syscall instruction, we observe only an overhead of 9.9 % for ECALLs and 24.0 % for OCALLs. This improvement is solely caused by the different execution times of the page-fault handler and the syscall handler. We discuss how future work can optimize our implementation in Section 6.1. In similar microbenchmarks, SGXJail [97] had an overhead of 41.4 % for ECALLs and 45.2 % for OCALLs. Hence, as shown by the microbenchmarks, optimized DPTI outperforms SGXJail by 22 % for ECALLs and 14.6 % for OCALLS.

Security

In this section, we evaluate the security the DPTI-Freeze- and DPTI-Stash-based enclave isolation.

Data-Only Attacks A non-isolated enclave can read or modify host application memory, hence allowing a malicious enclave to perform data-only attacks. DPTI prohibits data-only attacks by limiting write access to host pages, allowing the enclave to modify pages marked explicitly as DBPs only. As these pages are, by definition, used to pass data from and to a possibly untrusted enclave, we require the host to verify the correctness of the received data.

SGX-ROP To perform ROP-based attacks, the enclave needs to scan the host application for gadgets to build a ROP chain and overwrite the stored return address to execute the chain [84]. Both DPTI-Freeze and DPTI-Stash prohibit modifications to the return address on the stack by aligning the stack downwards before entering isolation, and restricting access to the upper stack pages containing the return address. As the enclave can freely modify stack and base pointer upon leaving isolation, DPTI verifies that the latter was not altered. For data pushed on the stack, we consider the two reasons a thread can exit an enclave: First, the thread can exit the enclave to return from an ECALL. In this case, we verify that the stack pointer was restored correctly and restore the saved registers before executing the return instruction. Second, the enclave exited to perform an OCALL. The data is pushed onto the user stack, and the protection only verifies that the stack pointer grew downwards. No return instruction is executed directly after the isolation end as an OCALL adds additional call frames instead of removing them.

EEXIT Destination The `EEXIT` instruction allows arbitrary return locations outside the SGX enclave, allowing returns to arbitrary, potentially missaligned, instructions [97]. We consider two cases: First, the enclave returns to a page different than the CBP, raising a page fault. As discussed in Section 4.2, this results in our page-fault handler raising a segmentation fault. Second, the enclave returns to the CBP, potentially misaligned. This poses no problem as the CBP is only filled with `NOP` instructions following the `EENTER`.

Data Confidentiality DPTI-Stash provides data confidentiality as host application pages are not mapped in the isolated address space. DPTI-Freeze does not guarantee confidentiality as pages are just mapped as read-only, allowing the enclave to read host data, but it still prevents malicious modification.

Multithreading and TOCTOU Multiple threads inside the same enclave share an isolated mapping (cf. Section 4.2). This does not pose a security problem, as threads within an enclave already share the enclave memory. Nevertheless, we ensure that multiple threads running inside the enclave cannot alter each other's stack during an OCALL by removing that thread's stack from the isolated mapping.

6 Discussion

In this section, we discuss future work and related work.

6.1 Future Work

Enhanced Syscall Filtering The largest bottleneck for efficient deep argument filtering are TLB flushes, which can be eliminated by employing protection keys. Previous work has explored the possibility of using such keys in the kernel [35]. A more recent patch set has even provided protection-key functionality to the kernel [15]. As protection keys contradict the aim of this work for providing a method for commodity, off-the-shelve systems, they were not used. Similarly, IMIX [27] could improve the performance of our deep argument filtering, but no hardware supports it. Future work can investigate whether seccomp can be extended to include our approach directly. To generate tight filters for complex arguments, the automated generation of syscall filter rules [11, 20, 31] can be extended to extract arguments automatically.

SGX-Protection Domain Section 5.2 shows that the overhead between raising a page fault and handling it is quite high. Merging the proof-of-concept directly into the kernel would increase the performance by greatly reducing the amount of code executed.

6.2 Related Work

Syscall Filtering Multiple works have been published on syscall checking [2, 3, 17, 25, 29, 32, 34, 46, 59, 72, 74, 93]. Some rely on kernel tracing [2, 3, 29, 34, 46, 74, 93], but the performance suffers from additional context switches. Hence, Linux relies on seccomp to perform syscall-filtering if it is requested by a developer. As discussed, seccomp requires a developer to manually identify the syscalls required by an application. Several recent works have investigated the feasibility of automatically identifying and generating these filter rules, eliminating the need for manual analysis [11, 20, 31]. While seccomp improves the security, it negatively affects the performance. Recent work has therefore proposed changing how seccomp handles filters [38, 88] or proposed additional hardware features [88] to improve the performance. Nevertheless, deep argument filtering is still not supported as of Linux 5.11. Salaün [78] restricts ambient rights, such as global filesystem accesses, for a set of processes. This is an orthogonal approach as it does not attempt to filter syscalls, but instead focuses on access control on kernel objects directly [14]. More recent work proposed to create on-demand snapshots and copies of accessed data, ensuring that reads to a userspace object during a syscall always returns the same value [7]. While this can be used for deep argument filtering, it cannot be used to protect against malicious enclaves. Recently, the usage of syscall sequences and origins has been explored for syscall-flow-integrity protection [10]. DPTI can extend this work to provide deep argument filtering.

Memory Isolation Memory isolation is a well-researched field where proposals can be grouped into OS-, virtualization-, hardware-, language- and runtime-based techniques. While memory isolation can be easily achieved on the OS level by simply placing the necessary parts into separate processes, this does incur a significant overhead. To prevent this significant overhead, recent work provides additional OS abstractions [18, 39, 61] that together with compiler support [13] or runtime analysis tools [8] make it feasible to isolate long-term signing keys in a web server. Other work has proposed to use a hypervisor for isolating memory. Dune [6] allows a user-space process to use the Intel VT-x virtualization extensions to isolate compartments. Other work demonstrated how the *VMFUNC* instruction can be used to switch extended page tables to achieve in-process isolation [54, 62]. SIM [87] uses VT-x for isolating a security monitor in an untrusted VM.

Memory isolation can also be enforced by the hardware, for instance, by Intel SGX or ARM TrustZone. Previous work used SGX to protect internal data structures of just-in-time compilers to prevent code-injection attacks [26]. While the hardware enforces this isolation, the switching overhead is similar to other approaches [54]. Other works proposed additional x86 ISA extensions to add load and store instructions that must be used to access data in a safe region [27, 68]. These approaches additionally require CFI [1] to protect against control-flow hijack attacks [89]. Several works have relied on Intel Memory Protection Keys (MPK) to facilitate more efficient memory isolation [37, 54, 71, 79, 91]. However, Intel MPK is not used in the kernel and not available for most commodity systems. It is only available in a limited subset of shipped processors since the Intel Xeon Skylake microarchitecture. Creative use of page-table entries for in-process isolation is a technique that has been explored [27, 54], but these solutions require modifications of the ISA or the re-purposing of ignored or reserved bits in page tables. Consequently, they can only be applied to future hardware but not to commodity systems.

Memory isolation can also be ensured through static checks in memory-safe languages. In unsafe languages, this isolation can be provided through software fault isolation, where runtime checks are added by the compiler or through binary rewriting [67, 94]. Naturally, this imposes a performance overhead while not protecting against control-flow hijack attacks, making it necessary to combine it with CFI. Recent work has explored the possibility of so-called *zero-cost transitions* between normal and sandboxed code for well-structured code, but still requires CFI [53]. Contrary to previous solutions for memory isolation [27, 37, 54, 68, 71, 79, 91], DPTI does not require ISA extensions or re-purposing of ignored page-table bits. Instead, we only rely on existing functionality and bring our enhanced filtering to commodity systems. DPTI retrofits sandboxing mechanisms with strategies that previous works have explored against microarchitectural attacks [36, 40].

Intel SGX The memory isolation of SGX is asymmetric, *i.e.*, while enclave memory is inaccessible for the operating system and the host application, the enclave has full access to the data and code of the host application. Schwarz et al. [84] showed that untrusted SGX enclaves can rewrite the host application memory to impersonate the host and execute arbitrary syscalls. To mitigate such attacks, Weiser et al. [97] proposed SGXJail, which contains two ways to isolate enclaves, either isolation through another process or through a slightly modified variant of Intel

MPK. Other works have proposed to monitor the I/O behavior of enclaves to detect a potential attack [16, 19]. Another proposed approach is to analyze enclave code before running it, but this is not possible for generic loaders [16]. Hence, Costan and Devadas [16] proposed to force generic loader enclaves to embed malware-analysis code into the enclave, but it is unknown how effective this approach is.

Other approaches improve the security of the enclave itself [55, 85, 101], but such approaches are orthogonal to our approach as they assume a malicious host or OS. An SFI-based approach has been proposed by Ryoan [41], which requires recompilation of the enclave using SFI, which is not necessary in our approach.

7 Conclusion

In this paper, we proposed DPTI, a security-domain isolation mechanism for commodity off-the-shelf CPUs. DPTI provides two mechanisms for time-limited changes to the memory isolation at security-critical points, called DPTI-Freeze and DPTI-Stash. While DPTI-Freeze makes memory temporarily read-only, DPTI-Stash temporarily removes access permissions. We evaluated the versatility of DPTI in two scenarios. In the first scenario, DPTI enables more fine-grained syscall filtering than seccomp while providing stronger memory-safety guarantees that allow supporting deep argument filtering. In the second scenario, DPTI efficiently confines SGX enclaves, outperforming existing solutions by 14.6 %-22 % and removing hardware requirements. Our results show that DPTI is a viable mechanism that can be used to isolate domains within applications without relying on special hardware instructions or extensions.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM, Amazon, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

Table 10.4: The specific commands which were used for evaluating the performance of an unsandboxed, seccomp-, and DPTI-sandboxed version of the respective application.

	Software	Command
busybox	diff	busybox diff cat.sh grep.sh
	true	busybox true
	env	busybox env
	ls	busybox ls
	dmesg	busybox dmesg
	cat	busybox cat test
	head	busybox head -n 100 test
	grep	busybox grep -ir python3 .
	pwd	busybox pwd
git	diff	git diff
	status	git status
ffmpeg	extract	ffmpeg_g -i video.mp4 -r 1 -f image2 image-%2d.png -y -hide_banner
	convert	ffmpeg_g -i video.mp4 video.avi -y -hide_banner
	remove	ffmpeg_g -i video.mp4 -an output.mp4 -y -hide_banner
	crop	ffmpeg_g -i video.mp4 -filter:v 'crop=640:480:200:150' output.mp4 -y -hide_banner
	info	ffmpeg_g -hide_banner -i video.mp4
	change	ffmpeg_g -i video.mp4 -filter:v scale=1280:720 -c:a copy output.mp4 -y -hide_banner

B Real-world Application Commands

In Table 10.4, we show the exact commands that were used for the evaluation of the real-world applications in Section 5.1.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity." In: *CCS*. 2005.

[2] Anurag Acharya and Mandar Raje. "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications." In: *USENIX Security Symposium*. 2000.

[3] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. *Consh: Confined Execution Environment for Internet Computations*. Tech. rep. The University of California, Santa Barbara, 1999.

[4] Android. *Application Sandbox*. 2021. URL: `https://source.android.com/security/app-sandbox`.

[5] Michael Backes and Stefan Nürnberger. "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing." In: *USENIX Security Symposium*. 2014.

[6] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. "Dune: Safe User-level Access to Privileged CPU Features." In: *OSDI*. 2012.

[7] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. "Midas: Systematic Kernel TOCTTOU Protection." In: *USENIX Security Symposium*. 2022.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments." In: *NSDI*. 2008.

[9] Davidlohr Bueso. *tools/perf-bench: Add basic syscall benchmark*. 2019. URL: `https://lore.kernel.org/patchwork/patch/1048777/`.

[10] Claudio Canella, Sebastian Dorn, Daniel Gruss, and Michael Schwarz. "SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems." In: *arXiv:2202.13716* (2022).

[11] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. "Automating Seccomp Filter Generation for Linux Applications." In: *CCSW*. 2021.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns." In: *CCS*. 2010.

[13] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. "Shreds: Fine-Grained Execution Units with Private Memory." In: *S&P*. 2016.

[14] Kernel development community. *Landlock: kernel documentation.* 2020. URL: https://landlock.io/linux-doc/landlock-v21/security/landlock/kernel.html.

[15] Jonathan Corbet. *Memory protection keys for the kernel.* 2020. URL: https://lwn.net/Articles/826554/.

[16] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *Cryptology ePrint Archive, Report 2016/086* (2016).

[17] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitaram. *Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code.* Tech. rep. 1997.

[18] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation." In: *ASPLOS.* 2015.

[19] Shaun Davenport and Richard Ford. *SGX: the good, the bad and the downright ugly.* 2014. URL: https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly.

[20] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. "sysfilter: Automated System Call Filtering for Commodity Software." In: *RAID.* 2020.

[21] Jake Edge. *A seccomp overview.* 2015. URL: https://lwn.net/Articles/656307/.

[22] Jake Edge. *Deep argument inspection for seccomp.* 2019. URL: https://lwn.net/Articles/799557/.

[23] Jake Edge. *Seccomp and deep argument inspection.* 2020. URL: https://lwn.net/Articles/822256/.

[24] Bryan Ford and Russ Cox. "Vx32: Lightweight User-level Sandboxing on the x86." In: *Usenix ATC.* 2008.

[25] T. Fraser, L. Badger, and M. Feldman. "Hardening COTS software with generic software wrappers." In: *DISCEX.* 2000.

[26] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. "JITGuard: Hardening Just-in-Time Compilers with SGX." In: *CCS.* 2017.

[27] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. "IMIX: In-Process Memory Isolation EXtension." In: *USENIX Security Symposium.* 2018.

[28] Tal Garfinkel. “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools.” In: *NDSS*. 2003.

[29] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. “Ostia: A Delegating Architecture for Secure System Call Interposition.” In: *NDSS*. 2004.

[30] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. “Confine: Automated System Call Policy Generation for Container Attack Surface Reduction.” In: *RAID*. 2020.

[31] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. “Temporal System Call Specialization for Attack Surface Reduction.” In: *USENIX Security Symposium*. 2020.

[32] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. “SLIC: An Extensibility System for Commodity Operating Systems.” In: *USENIX ATC*. 1998.

[33] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. “Out of Control: Overcoming Control-Flow Integrity.” In: *S&P*. 2014.

[34] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. “A secure environment for untrusted helper applications: Confining the wily hacker.” In: *USENIX Security Symposium*. 1996.

[35] Gravani, Spyridoula and Hedayati, Mohammad and Criswell, John and Scott, Michael L. “Fast Intra-Kernel Isolation and Security with IskiOS.” In: *RAID*. 2021.

[36] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR.” In: *ESSoS*. 2017.

[37] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. “Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries.” In: *Usenix ATC*. 2019.

[38] Tom Hromatka. *seccomp and libseccomp performance improvements*. 2018.

[39] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. “Enforcing Least Privilege Memory Views for Multithreaded Applications.” In: *CCS*. 2016.

[40] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. "EPTI: efficient defence against meltdown attack for unpatched VMs." In: *USENIX ATC*. 2018.

[41] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data." In: *Usenix OSDI*. 2016.

[42] Google Inc. *Seccomp filter in Android O*. 2017. URL: `https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html`.

[43] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.

[44] isolation:AppArmor. *isolation:AppArmor: Linux kernel security module*. 2021. URL: `https://apparmor.net/`.

[45] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks." In: *CCS*. 2018.

[46] Kapil Jain and R Sekar. "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement." In: NDSS. 2000.

[47] Vasileios Kemerlis. "Protecting Commodity Operating Systems through Strong Kernel Isolation." PhD thesis. Columbia University, 2015.

[48] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. "ret2dir: Rethinking kernel isolation." In: *USENIX Security Symposium*. 2014.

[49] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-User Attacks." In: *USENIX Security Symposium*. 2012.

[50] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. "V0LTpwn: Attacking x86 Processor Integrity from Software." In: *USENIX Security Symposium*. 2020.

[51] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors." In: *ISCA*. 2014.

[52] Vladimir Kiriansky and Carl Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses." In: *arXiv:1807.03757* (2018).

[53] Matthew Kolosick, Shravan Narayan, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. "Isolation Without Taxation: Near Zero Cost Transitions for SFI." In: *arXiv:2105.00033* (2021).

[54] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. "No Need to Hide: Protecting Safe Regions on Commodity Hardware." In: *EuroSys*. 2017.

[55] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory Safety for Shielded Execution." In: *EuroSys*. 2017.

[56] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. "Code-Pointer Integrity." In: *OSDI*. 2014.

[57] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. "Loop-oriented programming: a new code reuse attack to bypass modern defenses." In: *IEEE Trustcom/BigDataSE/ISPA*. 2015.

[58] Michael Larabel. *Concurrent TLB Flushing For Linux 5.13 Provide A Small Performance Benefit*. 2021. URL: `https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.13-Concurrent-TLB-Flush`.

[59] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. "Protecting Against Unexpected System Calls." In: *USENIX Security Symposium*. 2005.

[60] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium*. 2018.

[61] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. "Light-Weight Contexts: An OS Abstraction for Safety and Performance." In: *OSDI 16*. 2016.

[62] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. "Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation." In: *CCS*. 2015.

[63] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks." In: *CCS*. 2015.

[64] LWN. *The current state of kernel page-table isolation.* 2017. URL: `https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/`.

[65] Petr Matousek. *CVE-2016-5195 kernel: mm: privilege escalation via MAP_PRIVATE COW breakage.* 2016. URL: `https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c16`.

[66] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. "System V Application Binary Interface." In: *AMD64 Architecture Processor Supplement, Draft v0.99.7* (2014).

[67] Stephen McCamant and Greg Morrisett. "Evaluating SFI for a CISC Architecture." In: *USENIX Security Symposium.* 2006.

[68] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. "MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation." In: *RAID.* 2018.

[69] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX." In: *S&P.* 2020.

[70] Nergal. *The advanced return-into-lib(c) explits: PaX case study.* 2001.

[71] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)." In: *USENIX ATC.* 2019.

[72] David S. Peterson, Matt Bishop, and Raju Pandey. "A Flexible Containment Mechanism for Executing Untrusted Code." In: *USENIX Security Symposium.* 2002.

[73] Vassilis Prevelakis and Diomidis Spinellis. "Sandboxing Applications." In: *USENIX ATC.* 2001.

[74] Niels Provos. "Improving Host Security with System Call Policies." In: *USENIX Security Symposium.* 2003.

[75] Raul Quinonez. *SGXBENCH framework for benchmarking SGX enclaves.* 2018. URL: `https://github.com/isolation:sgxbench/isolation:sgxbench`.

[76] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *USENIX Security Symposium.* 2019.

[77] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses." In: *EuroS&P*. 2017.

[78] Mickaël Salaün. *Landlock: unprivileged access control.* 2016.

[79] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. "Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86." In: *USENIX Security Symposium.* 2020.

[80] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *S&P*. 2015.

[81] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs." In: *arXiv:1905.05725* (2019).

[82] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In: *AsiaCCS*. 2018.

[83] Michael Schwarz, Moritz Lipp, and Claudio Canella. *misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8.* 2018. URL: `https://github.com/misc0110/PTEditor`.

[84] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *DIMVA*. 2019.

[85] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs." In: *NDSS*. 2017.

[86] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In: *CCS*. 2007.

[87] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. "Secure In-VM Monitoring Using Hardware Virtualization." In: *CCS*. 2009.

[88] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. "Draco: Architectural and Operating System Support for System Call Security." In: *MICRO*. 2020.

[89] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *S&P*. 2013.

[90] Tizen. *Security:Seccomp*. 2018. URL: `https://wiki.tizen.org/Security:Seccomp`.

[91] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. "ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys." In: *USENIX Security Symposium*. 2019.

[92] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *S&P*. 2020.

[93] David A. Wagner. *Janus: An Approach for Confinement of Untrusted Applications*. Tech. rep. University of California at Berkeley, 1999.

[94] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. "Efficient software-based fault isolation." In: *ACM SOSP*. 1993.

[95] Pengfei Wang and Jens Krinke. "How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel." In: *USENIX Security Symposium*. 2017.

[96] Robert N. M. Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers." In: *WOOT*. 2007.

[97] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGXJail: Defeating Enclave Malware via Confinement." In: *RAID*. 2019.

[98] Mozilla Wiki. *Project Fission*. 2019. URL: `https://wiki.mozilla.org/Project_Fission`.

[99] Mozilla Wiki. *Security/Sandbox*. 2019. URL: `https://wiki.mozilla.org/Security/Sandbox`.

[100] SELinux Wiki. *FAQ — SELinux Wiki*. 2009. URL: `http://selinuxproject.org/w/?title=FAQ&oldid=729`.

[101] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. "MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX." In: *EuroSys*. 2020.

11 SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems

Publication Data

Claudio Canella, Sebastian Dorn, Daniel Gruss, and Michael Schwarz. "SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems." In: *arXiv:2202.13716* (2022)

Contributions

Main author.

SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems

Claudio Canella[1], Sebastian Dorn[1]
Daniel Gruss[1], Michael Schwarz[2]

[1] Graz University of Technology, Austria [2] CISPA Helmholtz Center for Information Security, Germany

Abstract

Control-Flow Integrity (CFI) is one promising mitigation that is more and more widely deployed and prevents numerous exploits. However, CFI focuses purely on one security domain, and transitions between user space and kernel space are not protected. Furthermore, if user-space CFI is bypassed, the system and kernel interfaces remain unprotected, and an attacker can run arbitrary transitions.

In this paper, we introduce the concept of syscall-flow-integrity protection (SFIP) that complements the concept of CFI with integrity for user-kernel transitions. Our proof-of-concept implementation relies on static analysis during compilation to automatically extract possible syscall transitions. An application can opt-in to SFIP by providing the extracted information to the kernel for runtime enforcement. The concept is built on three fully-automated pillars: First, a syscall state machine, representing possible transitions according to a syscall digraph model. Second, a syscall-origin mapping, which maps syscalls to the locations at which they can occur. Third, an efficient enforcement of syscall-flow integrity in a modified Linux kernel. In our evaluation, we show that SFIP can be applied to large scale applications with minimal slowdowns. In a micro- and a macrobenchmark, it only introduces an overhead of 13.1 % and 7.4 %, respectively. In terms of security, we discuss and demonstrate its effectiveness in preventing control-flow-hijacking attacks in real-world applications. Finally, to highlight the reduction in attack surface, we perform an analysis of the state machines and syscall-origin mappings of several real-world applications. On average, SFIP decreases the number of possible transitions by 41.5 % compared to seccomp and 91.3 % when no protection is applied.

1 Introduction

Vulnerablities in applications can be exploited by an attacker to gain arbitrary code execution within the application [63]. Subsequently, the attacker can exploit further vulnerabilities in the underlying system to elevate privileges [35]. Such attacks can be mitigated in either of these two stages: the stage where the attacker takes over control of a victim application [12, 63], or the stage where the attacker exploits a bug in the system to elevate privileges [34, 36]. Researchers and industry have focused on eliminating the first stage, where an attacker takes over control of a victim application, by reducing the density of vulnerabilities in software, e.g., by enforcing memory safety [12, 63]. The second line of defense, protecting the system, has also been studied extensively [20, 34, 36, 61]. For instance, sandboxing is a technique that tries to limit the available resources of an application, reducing the remaining attack surface. Ideally, an application only has the bare minimum of resources, e.g., syscalls, that are required to work correctly.

Control-flow integrity [1] (CFI) is a mitigation that limits control-flow transfers within an application to a set of pre-determined locations. While CFI has demonstrated that it can prevent attacks, it is not infallible [27]. Once it has been circumvented, the underlying system and its interfaces are once again exposed to an attacker as CFI does not apply protection across security domains.

In the early 2000s, Wagner and Dean [65] proposed an automatic, static analysis approach that generates syscall digraphs, *i.e.*, a k-sequence [17] of consecutive syscalls of length 2. A runtime monitor validates whether a transition is possible from the previous syscall to the current one and raises an alarm if it is not. The Secure Computing interface of Linux [15], seccomp, simplifies the concept by only validating whether a syscall is allowed, but not whether it is allowed in the context of the previous one. Recent work has explored hardware support for Linux seccomp to improve its performance [60]. In contrast to the work by Wagner and Dean [65] and other intrusion detection systems [19, 23, 29, 31, 42, 45, 64, 68, 69], seccomp acts as an enforcement tool instead of a simple monitoring system. Hence, false positives are not acceptable as they would terminate a benign application. Thus, we ask the following questions in this paper:

Can the concept of CFI be applied to the user-kernel boundary? Can prior syscall-transition-based intrusion detection models, e.g., digraph *models [65], be transformed into an enforcement mechanism without breaking modern applications?*

In this paper, we answer both questions in the affirmative. We introduce the concept of syscall-flow-integrity protection (SFIP), complementing the concept of CFI with integrity for user-kernel transitions. Our proof-of-concept implementation relies on static analysis during compilation to automatically extract possible syscall transitions. An application can opt-in to SFIP by providing the extracted information to the kernel for runtime enforcement. SFIP builds on three fully-automated pillars, a syscall state machine, a syscall-origin mapping, and an efficient SFIP enforcement in the kernel.

The **syscall state machine** represents possible transitions according to a syscall digraph model. In contrast to Wagner and Dean's [65] runtime monitor, we rely on an efficient state machine expressed as an $N \times N$ matrix (N is the number of provided syscalls), that scales even to large and complex applications. We provide a compiler-based proof-of-concept implementation, called *SysFlow*[1], that generates such a state machine instead of individual sets of k-sequences. For every available syscall, the state machine indicates to which other syscalls a transition is possible. Our syscall state machine (*i.e.*, the modified digraph) has several advantages including faster lookups ($O(1)$ instead of $O(M)$ with M being the number of possible k-sequences), easier construction, and less and constant memory overhead.

The **syscall-origin mapping** maps syscalls to the locations at which they can occur. Syscall instructions in a program may be used to perform different syscalls, *i.e.*, a bijective mapping between code location and syscall number is not guaranteed. We resolve the challenge of these non-bijective mappings with a mechanism propagating syscall information from the compiler frontend and backend to the linker, enabling the precise enforcement of syscalls and their origin. During the state transition check, we additionally check whether the current syscall originates from a location at which it is allowed to occur. For this purpose, we extend our syscall state machine with a syscall-origin mapping that can be bijective or non-bijective, which we extract from the program. Consequently, our approach eliminates syscall-based shellcode attacks and imposes additional constraints on the construction of ROP chains.

The **efficient enforcement** of syscall-flow integrity is implemented in the Linux kernel. Instead of detection, *i.e.*, logging the intrusion and notifying a user as is the common task for intrusion detection systems [37],

[1]`https://github.com/SFIP/SFIP`

we focus on enforcement. Our proof-of-concept implementation places the syscall state machine and non-bijective syscall-origin mapping inside the Linux kernel. This puts our enforcement on the same level as seccomp, which is also used to enforce the correct behavior of an application. However, detecting the set of allowed syscalls for seccomp is easier. As such, our enforcement is an additional technique to sandbox an application, automatically limiting the post-exploitation impact of attacks. We refer to our enforcement as *coarse-grained syscall-flow-integrity protection*, effectively emulating the concept of control-flow integrity on the syscall level.

We evaluate the performance of SFIP based on our reference implementation. In a microbenchmark, we only observe an overhead on the syscall execution of up to 13.1 %, outperforming seccomp-based protections. In real-world applications, we observe an average overhead of 7.4 %. In long-running applications, such as ffmpeg, nginx, and memcached, this overhead is even more negligible, with less than 1.8 % compared to an unprotected version. We evaluate the one-time overhead of extracting the information from a set of real-world applications. In the worst case, we observe an increase in compilation time by factor 28.

We evaluate the security of the concept of syscall-flow-integrity protection in a security analysis with special focus on control-flow hijacking attacks. We evaluate our approach on real-world applications in terms of number of states (*i.e.*, syscalls with at least one outgoing transition), number of average transitions per state, and other security-relevant metrics. Based on this analysis, SFIP, on average, decreases the number of possible transitions by 41.5 % compared to seccomp and 91.3 % when no protection is applied. Against control-flow hijacking attacks, we find that in nginx, a specific syscall can, on average, only be performed at the location of 3 syscall instructions instead of in 318 locations. We conclude that syscall-flow integrity increases system security substantially while only introducing acceptable overheads.

To summarize, we make the following contributions:

1. We introduce the concept of (coarse-grained) *syscall-flow-integrity protection* (SFIP) to enforce legitimate user-to-kernel transitions based on static analysis of applications.
2. Our proof-of-concept SFIP implementation is based on a syscall state machine and a mechanism to validate a syscall's origin.
3. We evaluate the security of SFIP quantitatively, showing that the number of possible syscall transitions is reduced by 91.3 % on average

in a set of 8 real-world applications, and qualitatively by analyzing the implications of SFIP on a real-world exploit.
4. We evaluate the performance of our SFIP proof-of-concept implementation, showing an overhead of 13.1 % in a microbenchmark and 7.4 % in a macrobenchmark.

2 Background

2.1 Sandboxing

Sandboxing is a technique to constrain the resources of an application to the absolute minimum necessary for an application to still work correctly. For instance, a sandbox might limit an application's access to files, network, or syscalls it can perform. A sandbox is often a last line of defense in an already exploited application, trying to limit the post-exploitation impact. Sandboxes are widely deployed in various applications, including in mobile operating systems [3, 32] and browsers [53, 70, 71]. Linux also provides various methods for sandboxing, including SELinux [72], state:AppArmor [62], or seccomp [15].

2.2 Digraph Model

The behavior of an application can be modeled by the sequence of syscalls it performs. In intrusion detection systems, windows of consecutive syscalls, so-called *k-sequences*, have been used [17]. k-sequences of length $k = 2$ are commonly referred to as digraphs [65]. A model built upon these digraphs allows easier construction and more efficient checking while reducing the accuracy in the detection [65] as only previous and current syscall are considered.

2.3 Linux Seccomp

The syscall interface is a security-critical interface that the Linux kernel exposes to userspace applications. Applications rely on syscalls to request the execution of privileged tasks from the kernel. Hence, securing this interface is crucial to improving the system's overall security.

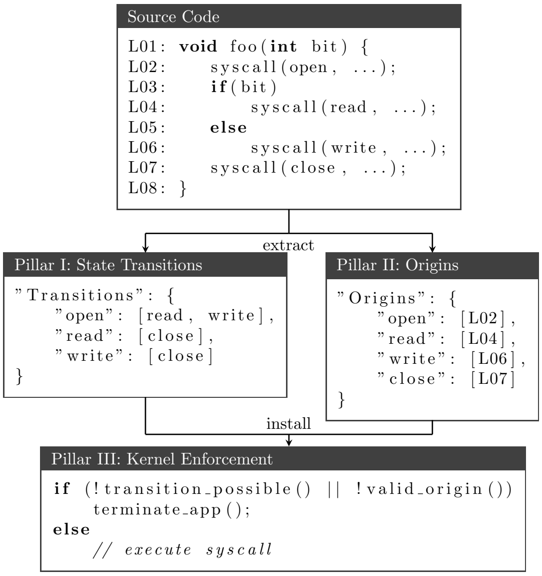
To better secure this interface, the kernel provides Linux Secure Computing (seccomp). A benign application first creates a filter that contains all the syscalls it intends to perform over its lifetime and then passes this filter to the kernel. Upon a syscall, the kernel checks whether the executed syscall is part of the set of syscalls defined in the filter and either allows or denies it. As such, seccomp can be seen as a k-sequence of length 1. In addition to the syscall itself, seccomp can filter static syscall arguments. Hence, seccomp is an essential technique to limit the post-exploitation impact of an exploit, as unrestricted access to the syscall interface allows an attacker to arbitrarily read, write, and execute files. An even worse case is when the syscall interface itself is exploitable, as this can lead to privilege escalation [34–36].

2.4 Runtime Attacks

One of the root causes for successful exploits are memory safety violations. One typical variant of such a violation are buffer overflows, enabling an attacker to modify the application in a malicious way [63]. An attacker tries to use such a buffer overflow to overwrite a code pointer, such that the control flow can be diverted to an attacker-chosen location, e.g., to previously injected *shellcode.* Attacks relying on shellcode have become harder to execute on modern systems due to data normally not being executable [47, 63]. Therefore, attacks have to rely on already present, executable code parts, so-called *gadgets.* These gadgets are chained together to perform an arbitrary attacker-chosen task [49]. Shacham further generalized this attack technique as return-oriented programming (ROP) [59]. Similar to control-flow-hijacking attacks that overwrite pointers [10, 27, 41, 56, 59], memory safety violations can also be abused in data-only attacks [33, 54].

2.5 Control-Flow Integrity

Control-flow integrity [1] (CFI) is a concept that restricts an application's control flow to valid execution traces, *i.e.*, it restricts the targets of control-flow transfer instructions. This is enforced at runtime by comparing the current state of the application to a set of pre-computed states. Control-flow transfers can be divided into forward-edge and backward-edge transfers [6]. Forward-edge transfers transfer control flow to a new destination, such as the target of an (indirect) jump or call. Backward-edge transfers transfer the control flow back to a location that was previously used in a forward edge, e.g., a return from a call. Furthermore, CFI can be subdivided into coarse-grained and fine-grained CFI. In contrast to fine-grained CFI, coarse-grained CFI allows for a more relaxed control-flow graph, allowing more targets than necessary [13].



```
L01: void foo(int bit) {
L02:     syscall(open, ...);
L03:     if(bit)
L04:         syscall(read, ...);
L05:     else
L06:         syscall(write, ...);
L07:     syscall(close, ...);
L08: }
```

```
"Transitions": {
    "open": [read, write],
    "read": [close],
    "write": [close]
}
```

```
"Origins": {
    "open": [L02],
    "read": [L04],
    "write": [L06],
    "close": [L07]
}
```

```
if (!transition_possible() || !valid_origin())
    terminate_app();
else
    // execute syscall
```

Figure 11.1: The three pillars of SFIP on the example of a function. The first pillar models possible syscall transitions, the second maps syscalls to their origin, and the third enforces them.

3 Design of Syscall-Flow-Integrity Protection

3.1 Threat Model

SFIP is applied to a benign userspace application that potentially contains a vulnerability allowing an attacker to execute arbitrary code within the application. The post-exploitation targets the operating system through the syscall interface to gain kernel privileges. With SFIP, a syscall is only allowed if the state machine contains a valid transition from the previous syscall to the current one and if it originates from a pre-determined location. If either one is violated, the application is terminated by the kernel. Similar to prior work [8, 14, 21, 22], our protection is orthogonal but fully compatible with defenses such as CFI, ASLR, NX, or canary-based protections. Therefore, the security it provides to the system remains even if these other protections have been circumvented. Side-channel and fault attacks [38, 39, 44, 55, 57, 73] on the state machine or syscall-origin mapping are out of scope.

3.2 High-Level Design

In this section, we discuss the high-level design behind SFIP. Our approach is based on three pillars: a digraph model for syscall sequences, a per-syscall model of syscall origin, and the strict enforcement of these models (cf. Figure 11.1).

For our first pillar, we rely on the idea of a digraph model from Wagner and Dean [65]. For our sycall-flow-integrity protection, we rely on a more efficient construction and in-memory representation. In contrast to their approach, we express the set of possible transitions not as individual k-sequences, but as a global syscall matrix of size $N \times N$, with N being the number of available syscalls. We refer to the matrix as our *syscall state machine*. With this representation, verifying whether a transition is possible is a simple lookup in the row indicated by the previous syscall and the column indicated by the currently executing syscall. Even though the representation of the sequences differs, the set of valid transitions remains the same: every transition that is marked as valid in our syscall state machine must also be a valid transition if expressed in the way discussed by Wagner and Dean. Our representation has several advantages though, that we explore in this paper, namely faster lookups ($O(1)$), less memory overhead, and easier construction.

Our syscall state machine can already be used for coarse-grained SFIP to improve the system's security (cf. Section 5.2). However, the second pillar, the validation of the origin of a specific syscall, further improves the provided security guarantees by adding additional, enforceable information. The basis for this augmentation is the ability to map syscalls to the location at which they can be invoked, independent of whether it is a bijective or non-bijective mapping. We refer to the resulting mapping as our *syscall-origin mapping*. For instance, our mapping might contain the information that the syscall instruction located at address `0x7ffff7ecbc10` can only execute the syscalls *write* and *read*. Neither unaligned execution (e.g., in a ROP chain) nor code inserted at runtime is in our syscall-origin mapping. Thus, syscalls can only be executed at already existing syscall instructions.

The third pillar is the enforcement of the syscall state machine and the syscall-origin mapping. Wagner and Dean [65] proposed their runtime monitoring as a concept for intrusion detection systems. There is still a domain expert involved to decide on any further action [37]. In contrast to monitoring, enforcement cannot afford false positives as this immediately leads to the termination of the application in benign scenarios. However, enforcement provides better security than monitoring as immediate action is undertaken, completely eliminating the time window for a possible

```c
void foo(int bit, int nr) {
  syscall(open, ...);
  if(bit)
    syscall(read, ...);
  else
    syscall(nr, ...);
  bar(...);
  syscall(close, ...);
}
```

Listing 3.1: Example of a dummy program with multiple syscall-flow paths.

exploit. Thus, by the use case of SFIP, namely enforcement of syscall-flow integrity, our concept is more closely related to seccomp but harder to realize than seccomp-based enforcement of syscalls.

3.3 Challenges

Previous automation work for seccomp filters outlined several challenges for automatically detecting an application's syscalls [8]. While several works [8, 14, 21] solve these challenges, none provides the full information required for SFIP. The challenges of getting this missing information focus on precise syscall information and inter- and intra-procedural control-flow transfer information. We illustrate the challenges using a simple dummy program in Listing 3.1.

$\mathcal{C}$1: Precise Per-Function Syscall Information The first challenge focuses on precise per-function syscall information. This challenge must be solved for the generation of the syscall state machine as well as the sycall-origin map. For seccomp-based approaches, *i.e.*, k-sequence of length 1, an automatic approach only needs to identify the set of syscalls within a function, *i.e.*, the exact location of the syscalls is irrelevant. This does not hold for SFIP, which requires precise information at which location a specific syscall is executed. Thus, we have to detect that the first syscall instruction always executes the *open* syscall, the second executes *read*, and the third syscall instruction can execute any syscall that can be specified via *nr*. For the state machine generation, the precise information of syscall locations provides parts of the information required to correctly generate the sequence of syscalls. For the syscall-origin map, the precise information allows generating the mapping of syscall instructions to actual syscalls in the case where syscall numbers are specified as a constant at the time of invocation.

$\mathcal{C}$2: Argument-based Syscall Invocations The second challenge extends upon **$\mathcal{C}$1** as it concerns syscall locations where the actual syscall executed cannot be easily determined at the time of compilation. When parsing the function `foo`, we can identify the syscall number for all invocations of the `syscall` function where the number is specified as a constant. The exception is the third invocation, as the number is provided by the caller of the `foo` function. As the call to the function, and hence the actual syscall number, is in a different translation unit than the actual syscall invocation, the possibility for a non-bijective mapping exists. Still, an automated approach must determine all possible syscalls that can be invoked at each syscall instruction.

$\mathcal{C}$3: Correct Inter- and Intra-Procedural Control-Flow Graph Precise per-function syscall information on its own is not sufficient to generate syscall state machines due to the non-linearity of typical code. Solving **$\mathcal{C}$1** and **$\mathcal{C}$2** provides the information which syscalls occur at which syscall location, but does not provide the information on the execution order. A trivial construction algorithm can assume that each syscall within a function can follow each other syscall, but this overapproximation leads to imprecise state machines. Such an approach accepts a transition from *read* to the syscall identified by *nr* as valid, even though it cannot occur within our example function.

Therefore, we need to determine the correct inter- and intra-procedural control-flow transfers in an application. The correct intra-procedural control-flow graph allows determining the possible sequences within a function. In our example, and if function `bar` does not contain any syscalls, it provides the information that the sequence of syscalls *open* → *read* → *close* is valid, while *open* → *nr* → *close* (where $nr \neq read$) is not.

Even in the presence of a correct intra-procedural control-flow graph, we cannot reconstruct the syscall state machine of an application as information is missing on the sequence of syscalls from other called functions. For instance, if function `bar` contains at least one syscall, the sequence of *open* → *read* → *close* is no longer valid. Hence, we additionally need to recover the precise location where control flow is transferred to another function and the target of this control-flow transfer. By combining the inter- and intra-procedural control-flow graph, the correct syscall sequences of an application can be modeled.

Constructing a precise control-flow graph is known to be a challenging task to solve efficiently [2, 28], especially in the presence of indirect control-flow transfers. These algorithms are often cubic in the size of the application, which makes them infeasible for large-scale applications. In the construction of the control-flow graph and, by extension, the generation of the syscall state machine and syscall-origin mapping, other factors, such as aliased and referenced functions, must be considered as well as functions that are passed as arguments to other functions, e.g., the entry function for a new thread created with `pthread_create`. Any form of imprecision can lead to the termination of the application by the runtime enforcement.

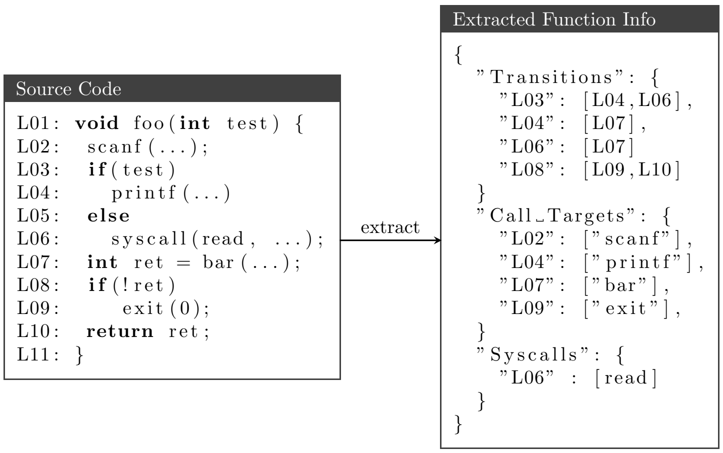
4 Implementation

In this section, we discuss our proof-of-concept implementation SysFlow and how we systematically solve the challenges outlined in Section 3.3 to provide fully-automated SFIP.

SysFlow SysFlow automatically generates the state machine and the syscall-origin mapping while compiling an application. As the basis of SysFlow we considered the works by Ghavamnia et al. [21] and Canella et al. [8].

4.1 State-Machine Extraction

In SysFlow, the linker is responsible for creating the final state machine. The construction works as follows: The linker starts at the main function, *i.e.*, the user-defined entry point of an application, and recursively follows the ordered set of control-flow transfers. Upon encountering a syscall location, the linker adds a transition from the previous syscall(s) to the newly encountered syscall. If control flow continues at a different function, the set of last valid syscall states is passed to the recursive visit of the encountered function. Upon returning from a recursive visit, the linker updates the set of last valid syscall states and continues processing the function. During the recursive processing, it also considers aliased and referenced functions. A special case, and source of overapproximation, are indirect calls, which we address with appropriate techniques from previous works [8, 14, 22]. The resulting syscall state machine and our support



```
L01: void foo(int test) {
L02:   scanf(...);
L03:   if(test)
L04:     printf(...)
L05:   else
L06:     syscall(read, ...);
L07:   int ret = bar(...);
L08:   if(!ret)
L09:     exit(0);
L10:   return ret;
L11: }
```

```
{
  "Transitions": {
    "L03": [L04,L06],
    "L04": [L07],
    "L06": [L07]
    "L08": [L09,L10]
  }
  "Call_Targets": {
    "L02": ["scanf"],
    "L04": ["printf"],
    "L07": ["bar"],
    "L09": ["exit"],
  }
  "Syscalls": {
    "L06" : [read]
  }
}
```

Figure 11.2: A simplified example of the information that is extracted from a function. *Transitions* identifies control-flow transfers between basic blocks, *Call Targets* the location of a call to another function and the targets name, *Syscalls* the location of the syscall and the corresponding syscall number.

libarary are embedded in the static binary. We discuss the support library in more detail in Section 4.3.

Building the state machine requires that precise information of the syscalls a function executes (***C*1**) and a control-flow graph of the application (***C*3**) is available to the linker. Both the front- and backend are involved in collecting this information. The frontend extracts the information from the LLVM IR generated from C source code, while the backend extracts the information from assembly files. Figure 11.2 illustrates the information that is extracted from a function.

Extracting Precise Syscall Information In the frontend, we iterate over every IR instruction of a function and determine the used syscalls. In the backend, we iterate over every assembly instruction to extract the syscalls. Extracting the information in the front- and backend successfully solves ***C*1**.

Extracting Precise Control-Flow Information Recovering the control-flow graph (***C*3**) in the frontend requires two different sources of information: IR call instructions and successors of basic blocks. The former allows tracking inter-procedural control-flow transfers while the latter allows tracking intra-procedural transfers. For inter-procedural transfers, we iterate over every IR instruction and determine whether it is a call to an external function. For direct calls, we store the target of the call; for indirect calls, we store the function signature of the target function. In addition, we also gather information on referenced and aliased functions, as well as functions that are passed as arguments to other functions. For the intra-procedural transfers, we track the successors of each basic block. In the backend, we perform similar steps, although on a platform-specific assembly level. Extracting this information in the front- and backend successfully solves $\boldsymbol{\mathcal{C}3}$.

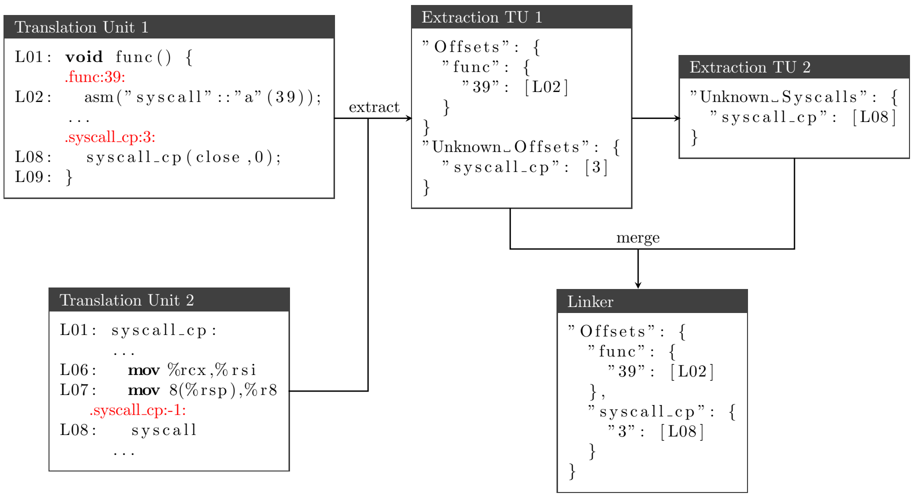
4.2 Syscall-Origin Extraction

In SysFlow, the linker also generates the final syscall-origin mapping. The mapping maps all reachable syscalls to the locations where they can occur. We extract the information as an offset instead of an absolute position to facilitate compatibility with ASLR. The linker requires precise information of syscalls, *i.e.*, their offset relative to the start of the encapsulating function, and a precise call graph of the application. Both the front- and backend are responsible for providing this information. Figure 11.3 illustrates the extraction. From the frontend, the syscall information generated by the state machine extraction is re-used ($\boldsymbol{\mathcal{C}1}$). A challenge is the possibility of non-bijective syscall mappings ($\boldsymbol{\mathcal{C}2}$).

Non-Bijective Syscall Mappings If the syscall number cannot be determined at the location of a syscall instruction, a non-bijective mapping exists for the instruction, *i.e.*, multiple syscalls can be executed through it. An example of such a case is shown in Listing 3.1. In such cases, the backend itself cannot create a mapping of a syscall to the syscall instruction. Hence, it must propagate the syscall number and the syscall offset from their respective translation unit to the linker, which can then merge it, solving $\boldsymbol{\mathcal{C}2}$.

4.3 Installation

For each syscall, the binary contains a list of all other reachable syscalls as an $N \times N$ matrix, *i.e.*, the state machine, with N being the number of syscalls available. Valid transitions are indicated by a 1 in the matrix, invalid ones with a 0 to allow fast checks and constant memory overhead. If a function contains a syscall, the offset of the syscall is added to the load address of the function. The state machine and the syscall-origin mapping are sent to the kernel and installed.



```
L01: void func() {
     .func:39:
L02:    asm("syscall"::"a"(39));
     ...
     .syscall_cp:3:
L08:    syscall_cp(close,0);
L09: }
```

```
"Offsets": {
  "func": {
    "39": [L02]
  }
}
"Unknown_Offsets": {
  "syscall_cp": [3]
}
```

```
"Unknown_Syscalls": {
  "syscall_cp": [L08]
}
```

```
L01: syscall_cp:
     ...
L06:    mov %rcx,%rsi
L07:    mov 8(%rsp),%r8
     .syscall_cp:-1:
L08:    syscall
     ...
```

```
"Offsets": {
  "func": {
    "39": [L02]
  },
  "syscall_cp": {
    "3": [L08]
  }
}
```

Figure 11.3: A simplified example of the syscall-origin extraction. Inserted red labels mark the location of a syscall and encode available information. The extraction deconstructs the label and calculates the offset using the label's address from the symbol table. The linker combines the information from each translation unit and generates the final syscall-origin mapping.

4.4 Kernel Enforcement

In this section, we discuss the third and final pillar of SFIP: enforcement of the syscall flow and origin where every violation leads to immediate process termination. Our Linux kernel is based on version 5.13 configured for Ubuntu 21.04 with the following modifications.

First, a new syscall, *SYS_syscall_sequence*, which takes as arguments the state machine, the syscall-origin mapping, and a flag that identifies the requested mode, *i.e.*, is state-machine enforcement requested, syscall-origin enforcement, or both. The kernel rejects updates to already installed syscall-flow information. Consequently, an unprivileged process cannot apply a malicious state machine or syscall origins before invoking a setuid binary or other privileged programs using the *exec* syscall [16].

Second, our syscall-flow-integrity checks are executed before every syscall. We create a new *syscall_work_bit* entry, which determines whether or not the kernel uses the slow syscall entry path, like in seccomp, to ensure that our checks are executed. Upon installation, we set the respective bit in the *syscall_work* flag in the *thread_info* struct of the requesting task.

Third, the syscall-flow information has to be stored and cleaned up properly. As it is never modified after installation, it can be shared between the parent and child processes and threads. Upon task cleanup, we decrease the reference counter, and if it reaches 0, we free the respective memory. The current state, *i.e.*, the previously executed syscall, is not shared between threads or processes and is thus part of every thread.

Enforcing State Machine Transitions Each thread and process tracks its own current state in the state machine. As we enforce sequence lengths of size 2, storing the previously executed syscall as the current state is sufficient for the enforcement. Due to the design of our state machine, verifying whether a syscall is allowed is a single lookup in the matrix at the location indicated by the previous and current syscall. If the entry indicates a valid transition, we update our current state to the currently executing syscall and continue with the syscall execution. Otherwise, the kernel immediately terminates the offending application. The simple state machine lookup, with a complexity of $O(1)$, ensures that only a small overhead is introduced to the syscall (cf. Section 5.1).

Enforcing Syscall Origins The enforcement of the syscall origins is very efficient due to its design. Our modified kernel uses the current syscall to retrieve the set of possible locations from the mapping to check whether the current RIP, minus the size of the syscall instruction itself, is a part of the retrieved set. If not, the application requested the syscall from an unknown location, which results in the kernel immediately terminating it. By design, the complexity of this lookup is $O(N)$, with N being the number of valid offsets for that syscall. We evaluate typical values of N in Section 5.2.

5 Evaluation

In this section, we evaluate the general idea of SFIP and our proof-of-concept implementation SysFlow. In the evaluation, we focus on the performance and security of the syscall state machines and syscall-origins individually and combined. We evaluate the overhead introduced on syscall executions in both a micro- and macrobenchmark. We also evaluate the time required to extract the required information from a selection of real-world applications.

Our second focus is the security provided by SFIP. We first consider the protection SFIP provides against control-flow hijacking attacks. We

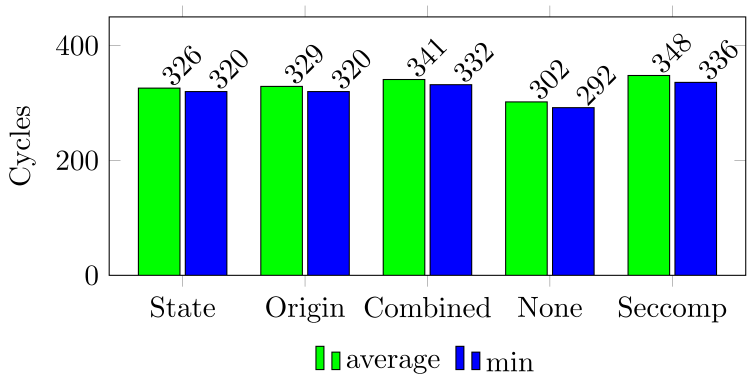


Figure 11.4: Microbenchmark of the *getppid* syscall over 100 million executions. We evaluate SFIP with only state machine, only syscall origin, both, and no enforcement active. For comparison, we also benchmark the overhead of seccomp.

evaluate the security of pure syscall-flow protection, pure syscall-origin protection, and combined protection. We discuss mimicry attacks and how SFIP makes such attacks harder. We also consider the security of the stored information in the kernel and discuss the possibility of an attacker manipulating it. Finally, we extract the state machines and syscall origins from several real-world applications and analyze them. We evaluate several security-relevant metrics such as the number of states in the state machine, average possible transitions per state, and the average number of allowed syscalls per syscall location.

5.1 Performance

Setup

All performance evaluations are performed on an i7-4790K running Ubuntu 21.04 and our modified Linux 5.13 kernel. For all evaluations, we ensure a stable frequency.

Microbenchmark

We perform a microbenchmark to determine the overhead our protection introduces on syscall executions. Our benchmark evaluates the latency of the *getppid* syscall, a syscall without side effects that is also used by kernel developers and previous works [5, 8, 30]. SysFlow first extracts the state machine and the syscall-origin information from our benchmark program,

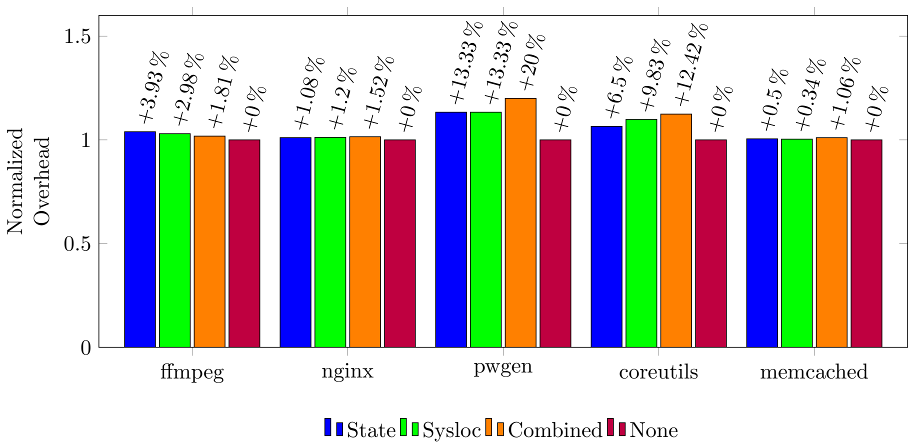


Figure 11.5: We perform a macrobenchmark using 5 real-world applications. For nginx, we measure the time it takes to handle 100 000 requests using `ab`. For ffmpeg, we convert a video (21 MB) from one file format to another. pwgen generates a set of passwords while coreutils and memcached are benchmarked using their respective testsuites. Each benchmark measures the average execution time over 100 repetitions of each mode of SFIP.

which we then execute once for every mode of SFIP, *i.e.*, state machine, syscall origins, and combined. Each execution measures the latency of 100 million syscall invocations. For comparison, we also benchmark the execution with no active protection. As with seccomp, syscalls performed while our protection is active require the slow syscall enter path to be taken. As the slow path introduces part of the overhead, we additionally measure the performance of seccomp in the same experiment setup.

Results Figure 11.4 shows the results of the microbenchmark. Our results indicate a low overhead for the syscall execution for all SFIP modes. Transition checks show an overhead of 8.15 %, syscall origin 9.13 %, and combined 13.1 %. Seccomp introduces an overhead of 15.23 %. The improved seccomp has a complexity of $O(1)$ for simple allow/deny filters [11], the same as our state machine. The syscall-origin check has a complexity of $O(N)$, with typically small numbers for N, *i.e.*, $N = 1$ for the *getppid* syscall in the microbenchmark. Section 5.2 provides a more thorough evaluation of N in real-world applications. The additional overhead in seccomp is due to its filters being written in cBPF and converted to and executed as eBPF.

Macrobenchmark

To demonstrate that SFIP can be applied to large-scale, real-world applications with a minimal performance overhead, we perform a macrobenchmark using applications used in previous work [8, 21, 60]. We measure the performance over 100 executions with only state machine, only syscall origin, both, and no enforcement active. For nginx, we measure the time it takes to process 100 000 requests. For ffmpeg, we convert a video (21 MB) from one file format to another. With pwgen, we generate a set of passwords while coreutils and memcached are benchmarked using their respective testsuites. In all cases, we verified that syscalls are being executed, e.g., each request for nginx executes at least 13 syscalls.

Results Figure 11.5 shows the results of the macrobenchmark. In nginx, we observe a small increase in execution time when any mode of SFIP is active. If both checks are performed, the average increase from 24.96 s to 25.34 s (+1.52 %) is negligible. We observe similar overheads in the ffmpeg benchmark. For the combined checks, we only observe an increase from 9.41 s to 9.58 s (+1.52 %). pwgen and coreutils show the highest overhead. pwgen is a small application that performs its task in under a second; hence any increase appears large. The absolute change in runtime is an increase of 0.05 s. For the coreutils benchmark, we execute the testsuite that involves all 103 utilities. Each utility requires that the SFIP information is copied to the kernel, which introduces a majority of the overhead. As the long-running applications show, the actual runtime overhead is less than 1.8 %. Our results demonstrate that SFIP is a feasible concept for modern, large-scale applications.

Extraction-Time Benchmark

We evaluate the time it takes to extract the information required for the state machine and syscall origins. As targets, we use several real-world applications (cf. Table 11.1) used in previous works on automated seccomp sandboxing [8, 14, 21]. These range from smaller utility applications such as busybox and coreutils to applications with a larger and more complex codebase such as ffmpeg, mupdf, and nginx. For the benchmark, we compile each application 10 times using our modified compiler with and without our extraction active.

Results Table 11.1 shows the result of the extraction-time benchmark. We present the average compilation time and the standard error for com-

Table 11.1: The results of our extraction time evaluation in real world applications. We present both the compilation time of the respective application with and without our extraction active.

Application	**Unmodified**	**Modified**
	Average / SEM	**Average / SEM**
ffmpeg	162.12 s / 0.78	1783.15 s / 10.61
mupdf	58.01 s / 0.71	489.85 s / 0.68
nginx	8.22 s / 0.03	226.64 s / 1.67
busybox	16.09 s / 0.08	81.33 s / 0.14
coreutils	5.50 s / 0.02	14.39 s / 0.41
memcached	2.90 s / 0.03	4.59 s / 0.01
pwgen	0.07 s / 0.00	0.12 s / 0.00

piling each application 10 times. The results indicate that the extraction introduces a significant overhead. For instance, for the coreutils applications, we observe an increase in compilation time from approximately 6 s to 15 s. We observe the largest increase in nginx from approximately 8 s to 227 s. Most of the overhead is in the linker, while the extraction in the frontend and backend is fast. We expect that a full implementation can significantly improve upon the extraction time by employing more efficient caching and by potentially applying other construction algorithms.

Similar to previous work [21], we consider the increase in compilation time not to be prohibitive as it is a one-time cost. Hence, the security improvement outweighs the increase in compilation time.

5.2 Security

In this section, we evaluate the security provided by SFIP. We discuss the theoretical security benefit of each mode of SFIP in the context of control-flow-hijacking attacks. We then evaluate a real vulnerability in BusyBox version 1.4.0 and later[1]. We also consider mimicry attacks [65, 66] and perform an analysis of real-world state machines and syscall origins.

[1] `https://ssd-disclosure.com/ssd-advisory-busybox-local-cmdline-stack-buffer-overwrite/`

Syscall-Flow Integrity in the Context of Control-flow Hijacking

In the threat model of SFIP (cf. Section 3.1), an attacker has control over the program-counter value of an unprivileged application. In such a situation, an attacker can either inject code, so-called shellcode, that is then executed, or reuse existing code in a so-called code-reuse attack. In a shellcode attack, an attacker manages to inject their own custom code. With control over the program-counter value, an attacker can redirect the control flow to the injected code. On modern systems, these types of attacks are by now harder to execute due to data execution prevention [47, 63], *i.e.*, data is no longer executable. As a result, an attacker must first make the injected code executable, which requires syscalls, e.g., the *mprotect* syscall. For this, an attacker has to rely on existing code (gadgets) in the exploited application to execute such a syscall. An attacker might be lucky, and the correct parameters are already present in the respective registers, resulting in a straightforward code-reuse attack commonly known as ret2libc [49]. Realistically, however, an attacker first has to get the location and size of the shellcode area into the corresponding registers using existing code gadgets. Depending on the type of gadgets, such attacks are known as return-oriented-programming [59] or jump-oriented-programming attacks [4].

On an unprotected system, every application can execute the *mprotect* syscall. Depending on the application, the *mprotect* syscall cannot be blocked by seccomp if the respective application requires it. With SFIP, attacks that rely on *mprotect* can potentially be prevented even if the application requires the syscall. First, we consider a system where only the state machine is verified on every syscall execution. *mprotect* is mainly used in the initialization phase of an application [8, 21]. Hence, we expect very few other syscalls to have a transition to it, if any. This leaves a tiny window for an attacker to execute the syscall to make the shellcode executable, *i.e.*, it is unlikely that the attempt succeeds in the presence of state-machine SFIP. Still, with only state-machine checks, the syscall can originate from any syscall instruction within the application.

Contrary, if only the syscall origin is enforced, the *mprotect* syscall is only allowed at certain syscall instructions. Hence, an attacker needs to construct a ROP chain that sets up the necessary registers for the syscall and then returns to such a location. In most cases, the only instance where *mprotect* is allowed is within the libc `mprotect` function. If executed from there, the syscall succeeds. If the syscall originates from another location, the check fails, and the application is terminated. Still, with only syscall origins being enforced, the previous syscall is not considered, allowing an attacker to perform the attack at any point in time.

With both active, *i.e.*, full SFIP, several restrictions are applied to a potential attack. The attacker must construct a ROP chain that either starts after a syscall with a valid transition to *mprotect* was executed, or the ROP chain must contain a valid sequence of syscalls that lead to such a state, *i.e.*, a mimicry attack (cf. Section 5.2). Additionally, all syscalls must originate from a location where they can legally occur. These additional constraints significantly increase the security of the system.

Real-world Exploit

For a real-world application, we evaluate a stack-based buffer overflow in the BusyBox arp applet from version 1.4.0 to version 1.23.1. In line with our threat model, we assume that all software-based security mechanisms, such as ASLR and stack protector, have already been circumvented. The vulnerable code is in the `arp_getdevhw` function, which copies a user-provided command-line parameter to a stack-allocated structure using `strcpy`. By providing a device name longer than `IFNAMSIZ` (default 16 characters), this overflow overwrites the stack content, including the stored program counter.

The simplest exploit we found is to mount a return2libc attack using a *one gadget RCE*, *i.e.*, a gadget that directly spawns a shell. In libc version 2.23, we discovered such a gadget at offset `0xf0897`, with the only requirement that offset `0x70` on the stack is zero, which is luckily the case. Hence, by overwriting the stored program counter with that offset, we can successfully replace the application with an interactive shell. With SFIP, this exploit is prevented. Running the exploit executes the *socket* syscall right before the *execve* syscall that opens the shell. While the *execve* syscall is at the correct location, the state machine does not allow a transition from the *socket* to the *execve* syscall. Hence, exploits that directly open a shell are prevented. We also verified that there is no possible transition from *socket* to *mprotect*,; hence loaded shellcode cannot be marked as executable. There are only 21 syscalls after a *socket* syscall allowed by the state machine. Especially as neither the *mprotect* nor the *execve* syscall are available, possible exploits are drastically reduced. To circumvent the protection, an attacker would need to find gadgets allowing a valid transition chain from the *socket* to the *execve* (or *mprotect*) syscall. We also note that the buffer overflow itself is also a limiting factor. As the overflow is caused by a `strcpy` function, the exploit payload, *i.e.*, the

ROP chain, cannot contain any null byte. Thus, given that user-space addresses on 64-bit systems always have the 2 most-significant address bits set to 0, a longer chain is extremely difficult to craft.

Syscall-Flow-Integrity Protection and Mimicry Attacks

We consider the possibility of mimicry attacks [65, 66] where an attacker tries to circumvent a detection system by evading the policy. For instance, if an intrusion detection system is trained to detect a specific sequence of syscalls as malicious, an attacker can add arbitrary, for the attack unneeded, syscalls that hide the actual attack. With SFIP, such attacks become significantly more complicated. An attacker needs to identify the last executed syscall and knowledge of the valid transitions for all syscalls. With this knowledge, the attacker needs to perform a sequence of syscalls that forces the state machine into a state where the malicious syscall is a valid transition. Additionally, as syscall origins are enforced, the attacker has to do this in a ROP attack and is limited to syscall locations where the specific syscalls are valid. While this does not make mimicry attacks impossible, it adds several constraints that make the attack significantly harder.

Security of Syscall-Flow Information in the Kernel

The security of the syscall-flow information stored in the kernel is crucial for effective enforcement. Once the application has sent the information to the kernel for enforcement, it is the responsibility of the kernel to prevent malicious changes to the information. The case where the initial information sent to the kernel is malicious is outside of the threat model (cf. Section 3.1).

The kernel stores the information in kernel memory; hence direct access and manipulation is not possible. The only way to modify the information is through our new syscall. Our implementation currently does not allow for any changes to the installed information, *i.e.*, no updates are allowed. An attacker using our syscall and a ROP attack to manipulate the information is also not possible as the syscall itself needs to pass SFIP checks before being executed. As the application contains no valid transition nor location for the syscall, the kernel terminates the application.

Still, as allowing no updates is a design decision, another implementation might consider allowing updates. In this case, the application needs to perform our new syscall to update the filters. Before our syscall is

Table 11.2: We evaluate various properties of applications state machines, including the average number of transitions per state, number of states in the state machine, min and max transitions. Busybox and coreutils show the averages over all contained utilites (398 and 103 utilities, respectively).

Application	Average Transitions	#States	Min Transitions	Max Transitions
busybox	15.73	24.51	1.0	21.09
pwgen	12.42	19	1	16
muraster	17.51	41	1	33
nginx	65.55	108	1	80
coreutils	15.75	27.11	1.0	23.0
ffmpeg	48.48	56	1	51
memcached	40.6	87	1	71
mutool	32.0	61	1	46

executed, SFIP is applied to the syscall, *i.e.*, it is verified whether there is a valid transition to it and whether it originates at the correct location. If not, the kernel terminates the application; otherwise, the update is applied. In this case, if timed correctly, an attacker is able to maliciously modify the stored information.

State Machine Reachability Anaysis

We analyse the state machine of several real-world applications in more detail. We define a state in our state machine as a syscall with at least one outgoing transition. While Wagner and Dean [65] only provide information on the *average branching factor*, *i.e.*, the number of average transitions per state, we extend upon this to provide additional insights into automatically generated syscall state machines. We focus on several key factors: the overall number of states in the application and the minimum, maximum, and average number of transitions across these states. These are key factors that determine the effectiveness of SFIP. We do not consider additional protection provided by enforcing syscall origins. We again rely on real-world applications that have been used in previous work [8, 14, 21, 60]. For busybox and coreutils, we do not provide the data for every utility individually, but instead present the average of all contained utilities, *i.e.*, 398 and 103, respectively. To determine the improvement in security, we consider an unprotected version of the respective application, *i.e.*, every syscall can follow the previously executed syscall. Additionally, we compare our results to a seccomp-based version.

Results Table 11.2 shows the results of this evaluation. nginx shows the highest number of states with 108, followed by memcached, mutool, and ffmpeg with 87, 61, and 56 states, respectively. coreutils and busybox also provide multiple functionalities but split across various utilities. Hence, their number of states is comparatively low.

Interestingly, each application has at least one state with only one valid transition. We manually verified this transition, and in every case, it is a transition from the *exit_group* syscall to the *exit* syscall, which is indeed the only valid transition for this syscall.

The combination of the average and maximum number of transitions together with the number of states provides some interesting insight. We observe that in most cases, the number of average transitions is relatively close to the maximum number of transitions, while the difference to the number of states can be larger. This indicates that our state machine is heavily interconnected. Modern applications delegate many tasks via syscalls to the kernel, such as allocating memory, sending data over the network, or writing to a file. As syscalls can fail, they are often followed by error checking code that performs application-specific error handling, logs the error, or terminates the application. Hence, a potential transition to these syscalls is automatically detected, leading to larger state machines. Another source is locking, as the involved syscalls can be preceded and followed by a wide variety of other syscalls. Additionally, the overapproximation of indirect calls also increases the number of transitions.

Even with such interconnected state machines, the security improvement is still large compared to an unprotected version of the application or even a seccomp-based version. In the case of an unprotected version, all syscalls are valid successors to a previously executed syscall. An unmodified Linux kernel 5.13 provides 357 syscalls. Compared to nginx, which has the highest number of average transitions with 66, this is an increase of factor 5.4 in terms of available transitions. In our state machine, the number of states corresponds to the number of syscalls an automated approach needs to allow for seccomp-based protection. These numbers also match the numbers provided in previous work on automated seccomp filter generation. For instance, Canella et al. [8] reported 105 syscalls in nginx and 63 in ffmpeg. Ghavamnia et al. [21] reported 104 in nginx. Each such syscall can follow any of the other syscalls that are part of the set. In the case of nginx, this is around factor 1.6 more than in the average state when SFIP is applied. Hence, we conclude that even coarse-grained SFIP can drastically increase the system's security.

Table 11.3: We evaluate various metrics for our syscall location enforcement, including the total number of functions containing syscalls, min, max and average number of syscalls per function, total syscall offsets found, average offsets per syscall, and the number of syscalls in the used musl syscall wrapper functions. Busybox and coreutils show the averages over all contained utilites (398 and 103 utilities, respectively).

Application	#Functions	Min Syscalls	Max Syscalls	Avg. Syscalls per Function	Total #Offsets	Avg #Offsets	#syscall()	#syscall_cp()	#syscall_cp_asm()
busybox	30.57	1.0	9.83	1.48	102.64	3.75	1.71	9.79	0
pwgen	28	1	3	1.25	84	4.42	0	2	0
muraster	55	1	12	1.62	193	4.6	0	4	0
nginx	105	1	24	1.53	318	3.0	7	24	0
coreutils	36.86	1.0	4.21	1.38	116.71	4.42	1.0	3.41	0
ffmpeg	89	1	13	1.55	279	4.98	0	13	13
memcached	101	1	20	1.5	317	3.69	0	20	0
mutool	81	1	14	1.67	278	4.15	6	14	0

Syscall Origins Analysis

We perform a similar analysis for our syscall origins in real-world applications. We focus on analyzing the number of syscall locations per application and for each such location, the number of syscalls that can be executed. Special focus is put on the number of syscalls that can be invoked through the syscall wrapper functions as they can allow a wide variety of syscalls. Hence, the fewer syscalls are available through these functions, the better the security of the system.

Results We show the results of this evaluation in Table 11.3. The average number of offsets per syscall indicates that many syscalls are available at multiple locations. This is most likely due to the inlining of the syscall. This number is largely driven by the *futex* syscall, as locking is required in many places of applications. Error handling is a less driving factor in this case as these are predominantly printed using dedicated, non-inlined functions.

The last two columns analyze the number of syscalls that can be invoked by the respective syscall wrapper function and demonstrate a non-bijective mapping of syscalls to syscall locations. Relatively few syscalls are available through the `syscall()` function as it can be more easily inlined, *i.e.*, it is almost always inlined within libc itself. On the other hand, `syscall_cp()` cannot be inlined as it is a wrapper around an aliased function that performs the actual syscall.

Our results also indicate that, on average, every function that contains a syscall contains more than one syscall. nginx contains the most functions with a syscall and the highest number of total syscall offsets. Without syscall-origin enforcement, an attacker can choose from 318 syscall locations to execute any of the 357 syscalls provided by Linux 5.13 during a ROP attack. With our enforcement, the number is drastically reduced as each one of these locations can, on average, perform only 3 syscalls instead of 357.

6 Discussion

Limitations and Future Work Our proof-of-concept implementation currently does not handle signals and syscalls invoked in a signal handler. However, this is not a conceptual limitation. The compiler can identify all functions that serve as a signal handler and the functions that are reachable through it. Hence, it can extract a per-signal state machine to which the kernel switches when it sets up the signal stack frame. This allows for small per-signal state machines, which further improve security. As this requires significant engineering work, we leave the implementation and evaluation for future work.

Our state-machine construction leads to coarse-grained state machines, which can be improved by the fact that we can statically identify syscall origins. Future work can intertwine this information on a deeper level with the generated state machine. By doing so, a transition to another state is then not only dependent on the previous and the current syscall number but also on the virtual address of the previous and current syscall instruction. This allows to better represent the syscall-flow graph of the application without relying on context-sensitivity or call stack information [26, 58, 65]. As this requires significant changes to the compiler and the enforcement in the kernel and thorough evaluation, we leave this for future work.

Recent work has proposed hardware support for seccomp [60]. In future work, we intend to investigate whether similar approaches are possible to improve the performance of SFIP.

Related Work In 2001, the seminal work by Wagner and Dean [65] introduced automatically-generated syscall NDFAs, NDPDAs, and digraphs for sequence checks in intrusion detection systems. SFIP builds upon digraphs but modifies their construction and representation to increase performance. We further extend upon their work by additionally verifying the origin of a syscall. The accuracy and performance of SFIP allows real-time enforcement in large-scale applications.

Several papers have focused on extracting and modeling an applications control flow based on the work by Forrest et al. [17]. Frequently, such approaches rely on dynamic analysis [19, 23, 29, 31, 42, 45, 64, 68, 69]. Other approaches rely on machine-learning techniques to learn syscall sequences or detect intrusions [7, 24, 46, 51, 67, 74]. Giffin et al. [25] proposed incorporating environment information in the static analysis to generate more precise models. The Dyck model [26] is a prominent approach for learning syscall sequences that rely on stack information and context-sensitive models. Other works disregard control flow and focus instead on detecting intrusions based on syscall arguments [40, 48]. Forrest et al. [18] provide an analysis on the evolution of system-call monitoring. Our work differs as we do not require stack information, context-sensitive models, dynamic tracing of an application, or code instrumentation. The only additional information we consider is the mapping of syscalls to syscall instructions.

Recent work has investigated the possibility of automatically generating seccomp filters from source or existing binaries [8, 14, 21, 22, 50]. SysFlow can be extended to generate the required information from binaries as well. More recent work proposed a faster alternative to seccomp while also enabling complex argument checks [9]. In contrast to these works, we consider syscall sequences and origins, which requires additional challenges to be solved (cf. Section 3.3).

A similar approach to our syscall-origin enforcement has been proposed by Linn et al. [43] and de Raadt [52]. The former extracts the syscall locations and numbers from a binary and enforces them on the kernel level but fails in the presence of ASLR. The latter restricts the execution of syscalls to entire regions, but not precise locations, *i.e.*, the entire text segment of a static binary is a valid origin. Additionally, in the entire region, any syscall is valid at any syscall location. Our work improves upon them in several ways as we (1) present a way to enforce syscall origins in the presence of ASLR, (2) limit the execution of specific syscalls to precise locations, (3) combine syscall origins with state machines which lead to a significant increase in security.

7 Conclusion

In this paper, we introduced the concept of syscall-flow-integrity protection (SFIP), complementing the concept of CFI with integrity for user-kernel transitions. In our evaluation, we showed that SFIP can be applied to large-scale applications with minimal slowdowns. In a micro- and a macrobenchmark, we observed an overhead of only 13.1 % and 7.4 %, respectively. In terms of security, we discussed and demonstrated its effectiveness in preventing control-flow-hijacking attacks in real-world applications. Finally, to highlight the reduction in attack surface, we performed an analysis of the state machines and syscall-origin mappings of several real-world applications. On average, we showed that SFIP decreases the number of possible transitions by 41.5 % compared to seccomp and 91.3 % when no protection is applied.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM and from Amazon. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. “Control-Flow Integrity.” In: *CCS*. 2005.

[2] Lars Ole Andersen. “Program Analysis and Specialization for the C Programming Language.” PhD thesis. University of Copenhagen, 1994.

[3] Android. *Application Sandbox*. 2021. URL: `https://source.android.com/security/app-sandbox`.

[4] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. “Jump-oriented programming: a new class of code-reuse attack.” In: *AsiaCCS*. 2011.

[5] Davidlohr Bueso. *tools/perf-bench: Add basic syscall benchmark*. 2019. URL: `https://lore.kernel.org/patchwork/patch/1048777/`.

[6] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. “Control-Flow Integrity: Precision, Security, and Performance.” In: *ACM Computing Surveys* (2017).

[7] Byrnes, Jeffrey and Hoang, Thomas and Mehta, Nihal Nitin and Cheng, Yuan. “A Modern Implementation of System Call Sequence Based Host-based Intrusion Detection Systems.” In: *TPS-ISA*. 2020.

[8] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. “Automating Seccomp Filter Generation for Linux Applications.” In: *CCSW*. 2021.

[9] Canella, Claudio and Kogler, Andreas and Giner, Lukas and Gruss, Daniel and Schwarz, Michael. “Domain Page-Table Isolation.” In: *arXiv:2111.10876* (2021).

[10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. “Return-oriented programming without returns.” In: *CCS*. 2010.

[11] Jonathan Corbet. *Constant-action bitmaps for seccomp()*. 2020.

[12] Crispan Cowan et al. “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” In: *USENIX Security*. 1998.

[13] Davi, Lucas and Sadeghi, Ahmad-Reza and Lehmann, Daniel and Monrose, Fabian. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection." In: *USENIX Security Symposium*. 2014.

[14] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. "sysfilter: Automated System Call Filtering for Commodity Software." In: *RAID*. 2020.

[15] Jake Edge. *A seccomp overview*. 2015. URL: `https://lwn.net/Articles/656307/`.

[16] Jake Edge. *System call filtering and no_new_privs*. 2012. URL: `https://lwn.net/Articles/475678/`.

[17] Forrest, S. and Hofmeyr, S.A. and Somayaji, A. and Longstaff, T.A. "A sense of self for Unix processes." In: *S&P*. 1996.

[18] Forrest, Stephanie and Hofmeyr, Steven and Somayaji, Anil. "The Evolution of System-Call Monitoring." In: *ACSAC*. 2008.

[19] Garvey, Thomas D. and Lunt, Teresa F. "Model-based intrusion detection." In: *NCSC*. 1991.

[20] Ge, Xinyang and Talele, Nirupama and Payer, Mathias and Jaeger, Trent. "Fine-Grained Control-Flow Integrity for Kernel Software." In: *Euro S&P*. 2016.

[21] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. "Temporal System Call Specialization for Attack Surface Reduction." In: *USENIX Security Symposium*. 2020.

[22] Ghavamnia, Seyedhamed and Palit, Tapti and Mishra, Shachee and Polychronakis, Michalis. "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction." In: *RAID*. 2020.

[23] Ghosh, Anup and Schwartzbard, Aaron and Schatz, Michael. "Learning Program Behavior Profiles for Intrusion Detection." In: *ID*. 1999.

[24] Ghosh, Anup K. and Schwartzbard, Aaron. "A Study in Using Neural Networks for Anomaly and Misuse Detection." In: *USENIX Security Symposium*. 1999.

[25] Giffin, Jonathon and Dagon, David and Jha, Somesh and Lee, Wenke and Miller, Barton. "Environment-Sensitive Intrusion Detection." In: *RAID*. 2005.

[26] Giffin, Jonathon T and Jha, Somesh and Miller, Barton P. “Efficient Context-Sensitive Intrusion Detection.” In: *NDSS*. 2004.

[27] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. “Out of Control: Overcoming Control-Flow Integrity.” In: *S&P*. 2014.

[28] Michael Hind. “Pointer analysis: Haven’t we solved this problem yet?” In: *PASTE*. 2001.

[29] Hofmeyr, Steven A. and Forrest, Stephanie and Somayaji, Anil. “Intrusion Detection Using Sequences of System Calls.” In: *J. Comput. Secur.* (1998).

[30] Tom Hromatka. *seccomp and libseccomp performance improvements*. 2018.

[31] Ilgun, K. and Kemmerer, R.A. and Porras, P.A. “State transition analysis: a rule-based intrusion detection approach.” In: *TSE* (1995).

[32] Google Inc. *Seccomp filter in Android O*. 2017. URL: `https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html`.

[33] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. “Block Oriented Programming: Automating Data-Only Attacks.” In: *CCS*. 2018.

[34] Vasileios Kemerlis. “Protecting Commodity Operating Systems through Strong Kernel Isolation.” PhD thesis. Columbia University, 2015.

[35] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. “ret2dir: Rethinking kernel isolation.” In: *USENIX Security Symposium*. 2014.

[36] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. “kGuard: Lightweight Kernel Protection against Return-to-User Attacks.” In: *USENIX Security Symposium*. 2012.

[37] Kemmerer, Richard A and Vigna, Giovanni. “Intrusion detection: a brief history and overview.” In: *Computer* (2002).

[38] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors.” In: *ISCA*. 2014.

[39] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution.” In: *S&P*. 2019.

[40] Kruegel, Christopher and Mutz, Darren and Valeur, Fredrik and Vigna, Giovanni. “On the Detection of Anomalous System Call Arguments.” In: *ESORICS*. 2003.

[41] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. “Loop-oriented programming: a new code reuse attack to bypass modern defenses.” In: *IEEE Trustcom/BigDataSE/ISPA*. 2015.

[42] Lane, Terran and Brodley, Carla E. “Temporal Sequence Learning and Data Reduction for Anomaly Detection.” In: *TOPS* (1999).

[43] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. “Protecting Against Unexpected System Calls.” In: *USENIX Security Symposium*. 2005.

[44] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space.” In: *USENIX Security Symposium*. 2018.

[45] Lunt, Teresa F. “Automated Audit Trail Analysis and Intrusion Detection: A Survey.” In: *NCSC*. 1988.

[46] Lv, Shaohua and Wang, Jian and Yang, Yinqi and Liu, Jiqiang. “Intrusion Prediction With System-Call Sequence-to-Sequence Model.” In: *IEEE Access* (2018).

[47] Microsoft. *Data Execution Prevention*. 2021. URL: `https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`.

[48] Mutz, Darren and Valeur, Fredrik and Vigna, Giovanni and Kruegel, Christopher. “Anomalous System Call Detection.” In: *TOPS* (2006).

[49] Nergal. *The advanced return-into-lib(c) explits: PaX case study*. 2001.

[50] Pailoor, Shankara and Wang, Xinyu and Shacham, Hovav and Dillig, Isil. “Automated Policy Synthesis for System Call Sandboxing.” In: *PACMPL* (2020).

[51] Qiao, Y. and Xin, X.W. and Bin, Y. and Ge, S. “Anomaly intrusion detection method based on HMM.” In: *Electronics Letters* (2002).

[52] Theo de Raadt. *syscall call-from verification*. 2019. URL: `https://lwn.net/Articles/806863/`.

[53] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *USENIX Security Symposium*. 2019.

[54] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses." In: *EuroS&P*. 2017.

[55] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load." In: *S&P*. 2019.

[56] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *S&P*. 2015.

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *CCS*. 2019.

[58] Sekar, R. and Bendre, M. and Dhurjati, D. and Bollineni, P. "A fast automaton-based method for detecting anomalous program behaviors." In: *S&P*. 2001.

[59] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In: *CCS*. 2007.

[60] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. "Draco: Architectural and Operating System Support for System Call Security." In: *MICRO*. 2020.

[61] Spengler, Brad. *Recent ARM Security Improvements*. 2013.

[62] state:AppArmor. *state:AppArmor: Linux kernel security module*. 2021. URL: `https://apparmor.net/`.

[63] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *S&P*. 2013.

[64] Teng, H.S. and Chen, K. and Lu, S.C. "Adaptive real-time anomaly detection using inductively generated sequential patterns." In: *S&P*. 1990.

[65] Wagner, D. and Dean, R. "Intrusion detection via static analysis." In: *S&P*. 2001.

[66] Wagner, David and Soto, Paolo. “Mimicry Attacks on Host-Based Intrusion Detection Systems.” In: *CCS*. 2002.

[67] C. Warrender, S. Forrest, and B. Pearlmutter. “Detecting intrusions using system calls: alternative data models.” In: *S&P*. 1999.

[68] Wenke, Lee and Stolfo, S.J. and Mok, K.W. “A data mining framework for building intrusion detection models.” In: *S&P*. 1999.

[69] Wespi, Andreas and Dacier, Marc and Debar, Hervé. “Intrusion Detection Using Variable-Length Audit Trail Patterns.” In: *RAID*. 2000.

[70] Mozilla Wiki. *Project Fission*. 2019. URL: `https://wiki.mozilla.org/Project_Fission`.

[71] Mozilla Wiki. *Security/Sandbox*. 2019. URL: `https://wiki.mozilla.org/Security/Sandbox`.

[72] SELinux Wiki. *FAQ — SELinux Wiki*. 2009. URL: `http://selinuxproject.org/w/?title=FAQ&oldid=729`.

[73] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.” In: *USENIX Security Symposium*. 2014.

[74] Zhengdao, Zhang and Zhumiao, Peng and Zhiping, Zhou. “The Study of Intrusion Prediction Based on HsMM.” In: *APSCC*. 2008.