Lukas Maar

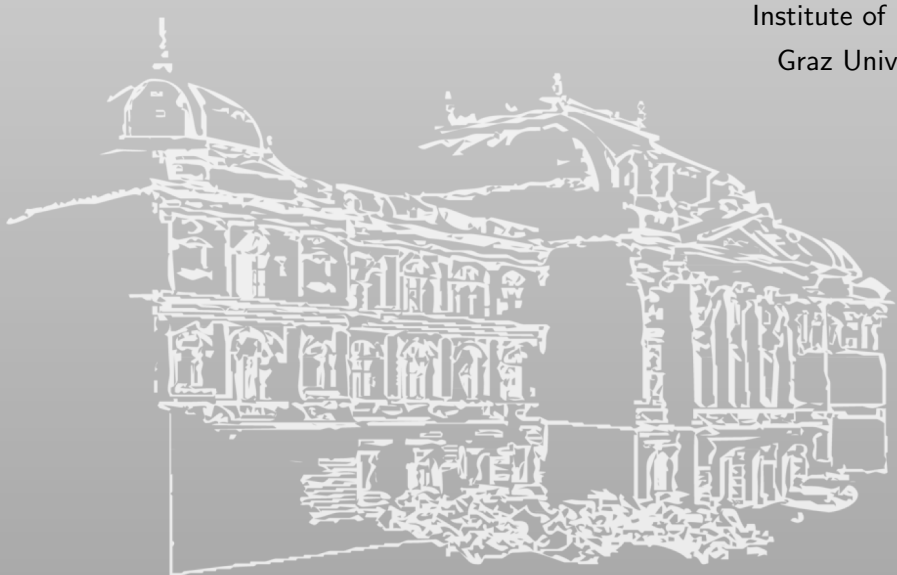
Kernel Security in the Wild:
Side-Channel-Assisted Exploit Techniques, Kernel-Level Defenses, and Real-World Analysis

PhD Thesis

Assessors: Stefan Mangard, Zhiyun Qian

September 2025

Institute of Information Security
Graz University of Technology



SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Abstract

Modern computing systems rely on a layered architecture comprising applications, hardware, and the operating system. At the core of the operating system lies the kernel, a privileged component that manages system resources and enforces isolation, e.g., between processes. Due to its complexity and ever-expanding codebase, the Linux kernel is susceptible to inadvertently introducing vulnerabilities. Exploiting such vulnerabilities typically bypasses the kernel's isolation mechanisms and thereby compromises the entire system. These risks are magnified by the widespread use of the Linux kernel across desktops, servers, and, in particular, mobile devices, of which there are billions in use. This makes the Linux kernel a high-value target for threat actors seeking to compromise systems or install persistent surveillance tools, highlighting the severe impact on society. Therefore, securing the kernel has become an ongoing arms race between offensive and defensive research efforts, with both efforts playing a critical role in countering increasingly sophisticated threats. Yet, despite considerable progress, key challenges remain regarding the reliability of kernel exploits, the effectiveness of existing mitigations, and the opaque deployment of defenses against in-the-wild attacks.

In this thesis, we address all three challenges to advance the state of kernel security. First, we analyze kernel exploits and explore ways to increase their success rate and reliability by exploiting side channels, which are unintentional information channels leaking metadata. We introduce three novel side channels: SLUBStick, a timing side channel on the kernel's memory allocator to infer heap memory reuse; KernelSnitch, a software-induced side channel that leaks the location of kernel heap objects via data structure access timing; and a hardware-induced TLB side channel that leaks fine-grained memory layout information. These side-channel leakages all enable powerful exploit techniques with high reliability that were previously either unreliable or infeasible. Second, we analyze modern kernel defenses against vulnerability exploitation. They limit control-flow hijacking or data-oriented attacks, which target control-related or non-control-related data for privilege escalation. We identify key gaps in these defenses and present HEK-CFI to prevent advanced control-flow hijacking and DOPE to counter data-oriented attacks. Both defenses offer stronger protection than prior work while maintaining reasonable performance overheads. Third, to assess real-world exploitability, we conduct two large-

scale analyses of the Android kernel ecosystem: Defects-in-Depth and The Doom of Device Drivers. Our studies reveal critical shortcomings in defense adoption, widespread exposure to known vulnerabilities in device drivers, and significant patch delays. These findings demonstrate that, despite available upstream mitigations or patches, many Android devices remain susceptible to known kernel attacks.

Together, the contributions of this thesis address key gaps in exploitation reliability, kernel defenses, and real-world analysis. By uncovering novel insights, presenting mitigations, and reporting findings, this thesis enhances the security and resilience of the Linux kernel.

This thesis is split into two parts. The first part outlines its contributions, provides background, summarizes state-of-the-art, and concludes its findings. The second part presents the first-authored papers in their original form[1], comprising seven publications. Five of these were accepted at renowned international tier-1 security conferences, while the other two were accepted at tier-2 ones.

[1]The content of the included papers remains unmodified from the camera-ready versions. However, their format and color style were adjusted to fit the layout of this thesis.

Acknowledgements

First and foremost, I want to thank my supervisor, Stefan Mangard. You gave me the opportunity to pursue a PhD, for which I am deeply grateful. Thank you for your continuous support throughout my PhD, for the inspiring discussions we had, for helping me refine my writing and structure ideas, for the freedom you gave me in conducting my own research, for guiding me in difficult decisions, and for being generous in allowing me to work from home occasionally, which made the travels between Innsbruck and Graz much easier.

I want to thank Zhiyun Qian for taking the time and effort to review my PhD thesis and provide feedback on it, as well as for the enlightening discussions we shared at conferences.

I am also grateful to Daniel Gruss, who had an important influence on my path toward cybersecurity. From giving me the opportunity to work as an operating systems tutor, to supporting me through inspiring discussions, helping me refine my writing, and connecting me to other brilliant minds, your support was an important step in shaping my journey. Thank you.

I want to thank Mathias Oberhuber, with whom I had the pleasure of sharing an office for about two years. We had countless discussions, both work-related and personal, and we always supported each other in every kind of situation. It was a real joy to share an office with you. I also wish to thank my other colleagues, especially those from my team: Lukas Lamster, Martin Unterguggenberger, Paul Gollob, Ernesto García, Lorenz Schumm, Rishub Nagpal, Moritz Waser and Martin Wistauder, as well as Pascal Nasahl, Gaëtan Cassiers and all the former members. From other teams I collaborated closely with (academic or non-academic), I want to thank Jonas Juffinger, Florian Draschbacher, Martin Schwarzl, Lukas Giner, Stefan Gast, Fabian Rauscher, Andreas Kogler, Vedad Hadzic, Thomas Steinbauer and Sudheendra Neela. It was a pleasure to exchange ideas, tackle challenges, and find solutions together. My thanks also go to everyone at the institute with whom I discussed paper ideas and other topics, as well as to all the people I met and enjoyed spending time with at conferences. A special thank you goes to Theresa Dachauer for creating the beautiful graphics for the papers and the cover of this thesis, which greatly enhanced my work and received very positive feedback. I also want to thank the ISEC administration team, especially Ursula Urwanisch, for

all their help and support. I wish you all the very best for the future. For those pursuing their PhDs, I wish you perseverance and joy on your journey.

One of the most memorable aspects of my PhD was the opportunity to travel to places I might not have traveled to otherwise. From camping in Joshua Tree National Park under a sky full of stars to hiking through the jungles of Indonesia and observing orangutans in the wild, these experiences will stay with me for the rest of my life. I am grateful to all the people I shared these adventures with and to everyone I met along the way.

I also want to thank those who took the time to provide feedback or review the structure of this thesis, either partially or fully: Martin Schwarzl, Lukas Lamster, Martin Unterguggenberger, Ernesto García, Lorenz Schumm and Andrey Konovalov.

Finally, I want to thank Ziming Zhao, whom I first met at AsiaCCS 2024, where you served as my session chair. In an academic world where negativity often surrounds the acceptance of system defense research in particular, you brought positivity and encouragement to the system security community. Thank you for that.

I owe my deepest gratitude to my parents and my twin brother. Elisabeth, Edgar, and Gabriel, you are the best family I could ever wish for. Thank you for your unconditional support throughout my studies and life, as well as for giving me the freedom and opportunity to pursue my dreams. Without you, I would not have become the person I am today. I am also grateful to my close friends, who have supported me through both good and difficult times. In particular, I want to thank Christian, Valentin, Daniel, Hannes, Josef, Viktoria, and Andreas. Without you, too, I would not be the person I am today. My thanks also go to the many other friends and companions, past and present, who have been an invaluable part of my life and journey.

Contents

Part I.

Kernel Security in the Wild

1

Introduction

Modern computing systems typically comprise three layers: hardware resources, the operating system, and software applications. Applications provide the functionality with which users interact, while hardware performs the actual computations. The operating system acts as an intermediary between applications and hardware, managing access to physical resources. At the core of the operating system lies the kernel, a privileged component responsible for abstracting and managing critical hardware resources such as the CPU, memory, and I/O devices. To fulfill these responsibilities, the kernel typically operates with the highest privilege level on non-virtualized systems, granting it unrestricted access to the machine. It performs essential low-level tasks ranging from memory management and device driver handling to process scheduling and system calls.

One of the kernel's key responsibilities is enforcing isolation and privilege separation. It ensures that software applications are isolated from one another, preventing one process from affecting others if it is faulty or malicious. The kernel also maintains a strict boundary between itself and user space to preserve the distinction between unprivileged and privileged execution contexts. Modern processors facilitate these separations by providing hardware-enforced paging and privilege levels, which the operating system leverages to enforce security boundaries. By controlling access to hardware and shared system resources, the kernel plays a vital role in maintaining both system correctness and security.

Given the foundational role of the kernel in modern systems, fulfilling its responsibilities requires immense complexity and an extensive, ever-evolving codebase. The Linux kernel, which is the focus of this thesis, had approximately 40 million lines of code in January 2025 [246] and is actively developed by thousands of contributors worldwide [142]. For context, in 2022 the Linux kernel comprised approximately 30 million lines [246], reflecting a growth of about 10 million lines in three years.

This ever-evolving landscape introduces inherent risks: vulnerabilities are unintentionally introduced, some of which remain undetected or unpatched for years [6, 45, 47, 270, 320, 343]. A notorious example is the kernel vulnerability CVE-2019-2215, which was actively exploited in the wild by the BadBinder attack [270]. Although the vulnerability was discovered in 2017, it took nearly two years before patches reached the affected end-user systems. Moreover, it remains unclear how long the vulnerability had been actively exploited by threat actors [273]. Other research [6, 45, 47] has generally shown that kernel vulnerabilities have an average lifetime of five years between introduction and patching. Such flaws pose serious threats to system integrity, as they allow actors to bypass isolation mechanisms, compromise the kernel, and gain full control over affected systems.

The consequences are particularly severe given the widespread use of the Linux kernel. It powers a wide range of systems, from desktop and server environments to mobile devices and embedded systems. Android, for example, uses a downstream version of the Linux kernel and is deployed on billions of mobile devices worldwide. Similarly, the Linux kernel underpins multiple desktop distributions and enterprise-grade server systems. When accounting for all upstream and downstream versions, the Linux kernel was used by over 47 % of operating systems worldwide as of April 2025 [79], making it one of the most widely deployed kernel. As a result, vulnerabilities in the Linux kernel pose systematic risks with a wide-reaching impact on society. One noteworthy example is the compromise of mobile devices to install commercial spyware such as Pegasus [123], Predator [206], or NoviSpy [120], enabling the covert surveillance of end users.

Given the Linux kernel's high privilege level, immense complexity, susceptibility to vulnerabilities, and widespread deployment across critical computing infrastructure, securing it against malicious attacks remains a pressing challenge. As a result, kernel security has become an ongoing arms race between defensive countermeasures and offensive techniques. To meaningfully strengthen the kernel's resilience, it is essential to study both sides of this race. Defenses have a direct impact by making exploitation more difficult, while offensive research contributes indirectly by uncovering previously unknown weaknesses and driving for more effective defenses. However, despite considerable progress, three key areas remain under-explored in terms of evolving threats:

(1) Exploit Reliability: The success rate of kernel exploits is a critical factor, where failure potentially results in crashes. A crash is not merely a technical error. It breaks stealth, increases the risk of detection,

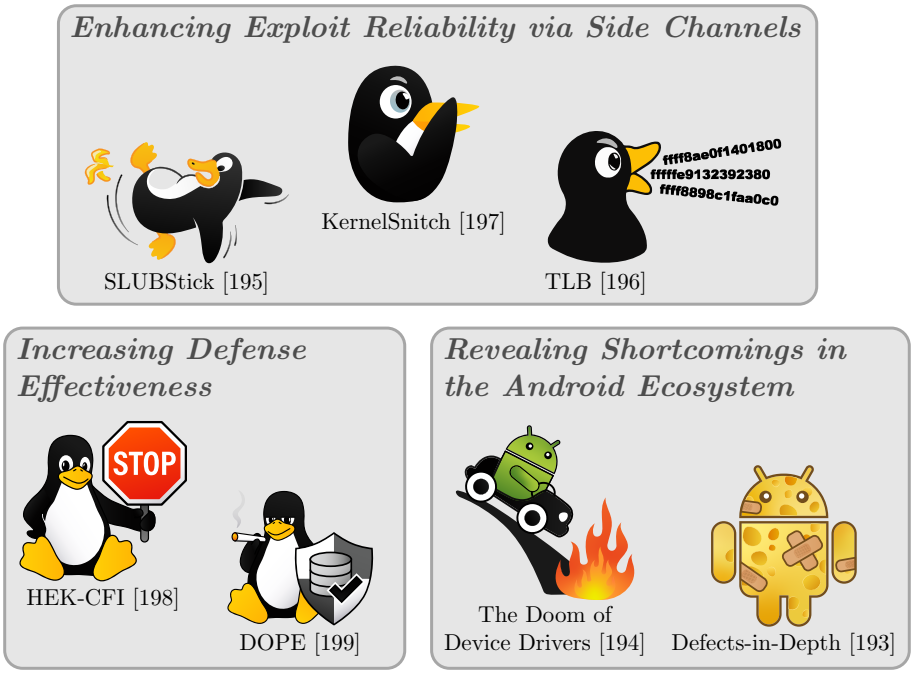


Figure 1.1: Three main contributions of this thesis.

and may trigger forensic investigations [123, 130, 204]. Hence, novel approaches are needed to enhance the reliability of kernel exploits.

(2) Defense Effectiveness: With the rise of novel exploit techniques that bypass defenses [19, 126, 128, 156, 211, 230, 310], it remains uncertain whether current kernel exploitation mitigations and defenses [53, 166, 220, 248, 265] are sufficient to withstand modern, sophisticated attacks.

(3) Android Ecosystem: Threat actors continue to target the Android kernel to, e.g., install spyware for surveillance [32, 231, 267]. Yet, the large-scale deployment of effective defenses to limit in-the-wild exploitation remains largely unknown.

This thesis addresses gaps in all three key areas through its main contributions (see Figure 1.1 and Section 1.1), while also tackling other contributions (see Section 1.2).

First, to enhance the exploit's reliability, we investigate how to increase exploitation success using side channels. These are unintended information channels that leak runtime metadata and deduces security-relevant

information. While prior work [89, 114, 128, 172, 189] has demonstrated that side channels can assist in kernel exploitation, these techniques only leak coarse-grained information [89, 114, 128, 189] or lacked general applicability [172]. To overcome these limitations, we present three novel side channels [195, 196, 197] that significantly improve the success rate of kernel-level exploitation. SLUBStick [195] is a generic timing side channel in the kernel allocator to infer kernel heap memory reuse. KernelSnitch [197] is the first side-channel attack to leak the precise location of kernel heap objects. While KernelSnitch is software-based, we also present a hardware-induced side-channel attack on the Translation Lookaside Buffer (TLB) [196], which caches recent virtual-to-physical address translations. This TLB side channel reveals the locations of kernel heap objects, kernel stacks, and page tables. Together, these side channels enable new exploit techniques with high reliability, while significantly improving the reliability of existing techniques. By advancing offensive capabilities, our work drives defense development, contributing to a more resilient kernel.

Second, despite advances in kernel defenses, many approaches remain vulnerable to bypass techniques [19, 126, 128, 156, 211, 230, 310]. To address these shortcomings, we present two defense mechanisms [198, 199]. These defenses improve protection against control-flow hijacking and data-oriented attacks that target control-related and non-control-related data for privilege escalation. HEK-CFI [198] addresses advanced control-flow hijacking techniques that previous defenses could not fully prevent. DOPE [199] protects multiple sensitive kernel objects against data manipulation, achieving a stronger security-performance trade-off than prior work.

Third, we analyze the Android kernel ecosystem in the context of in-the-wild exploitation through two studies [193, 194], revealing multiple shortcomings. Defects-in-Depth [193] examines how upstream kernel defenses are incorporated into downstream Android devices. The study reveals significant deviations from the security levels that could have been achieved. The Doom of Device Drivers [194] shows that threat actors can exploit publicly known n-day vulnerabilities in device drivers exposed to untrusted execution contexts, eliminating the need to invest in costly zero-day development. We reported the findings of both works and encouraged multiple vendors to enhance the security of their devices against kernel-level exploitation.

The goal of kernel security research is to raise the cost of in-the-wild exploitation, thereby preventing even well-resourced actors from carrying

out attacks [32, 268, 272, 273, 274, 276, 293, 294]. The recent Paragon campaign with Graphite spyware [122, 205] may reflect early success in this direction, with threat actors shifting focus from kernel-level compromises to targeting legitimate applications instead. While not a definitive victory, this change of direction signals meaningful progress. This thesis supports this trajectory by enhancing security directly through advanced kernel mitigations, as well as by encouraging defensive adaptation through improving exploit reliability via novel side channels and uncovering shortcomings in real-world systems. Ultimately, it contributes to making modern systems more resilient against kernel-level threats.

1.1. Main Contributions

During my PhD, I first-authored seven papers that advance the state of the art in kernel security. Five of these were published at tier-1 security conferences, while the others were published at tier-2 ones. My work covers a wide spectrum, including attacks, defenses, and in-depth analyses of kernel-level vulnerability exploitation, as well as side-channel-assisted exploitation techniques. The research targets both Linux-based desktop and Android operating systems, reflecting a comprehensive investigation of modern Linux kernel security. To structure these contributions (see Figure 1.1), I group them into: *Enhancing Exploit Reliability via Side Channels*, *Increasing Defense Effectiveness*, and *Revealing Shortcomings in the Android Ecosystem*.

1.1.1. Enhancing Exploit Reliability via Side Channels

Kernel exploits typically face two major challenges. First, real-world vulnerabilities often provide limited capabilities. For instance, such vulnerabilities include out-of-bounds [187] or use-after-free [242] writes at specific offsets with partially controlled data, or heap out-of-bounds writes restricted to two zero bytes [227]. Using such constrained vulnerabilities for full kernel compromise is often non-trivial and requires carefully crafted, multi-stage exploitation strategies.

Second, modern kernels deploy robust defenses that significantly limit kernel-level vulnerability exploitation [294]. Two key defenses are heap separation of kernel objects and randomization of their memory locations. With heap separation, the kernel places heap objects into distinct allocator

Table 1.1: Systematization of our kernel side-channel contributions, categorized by side-channel type, target, and exploitation goal.

Side Channel	Type	Target	Goal
SLUBStick [195]	Software	Slab allocator	Cross-cache reuse (heap manipulation)
KernelSnitch [197]	Software	Kernel data structures	Leakage of heap object locations Covert channel Website fingerprinting
TLB disclosure [196]	Hardware	TLB	Leakage of heap object locations Leakage of stack locations Leakage of page table locations

caches based on their security context [46, 109, 191]. This separation helps ensure that vulnerabilities in one cache do not compromise sensitive data in other, security-critical caches. As a result, it serves as a defense mechanism that hinders attackers from establishing reliable exploit primitives for kernel-level exploitation. With object randomization, the kernel randomizes the memory location of kernel objects [65], requiring attackers to first leak the location of a target object before overwriting it. Such leaks are typically achieved either through kernel vulnerabilities or via side-channel attacks. The former risks system crashes and reduces exploit reliability and stealth, undermining the viability of kernel exploitation. The latter has so far been limited to only leaking coarse-grained information [27, 89, 114, 159, 186, 189], such as the base address of the kernel code or the physical map. While these coarse-grained side-channel attacks are useful and widely employed in practice [82, 128, 188], it remained unclear whether finer-grained side-channel leakages were possible.

This part addresses both challenges by advancing side-channel-assisted, kernel-level exploit techniques. These techniques enhance the reliability and stability of end-to-end kernel compromise. To provide a systematic view, we map our contributions into the space of side-channel types (i.e., software-induced and hardware-induced) and attacker goals (e.g., heap manipulation, memory disclosure, and covert channels). Table 1.1 summarizes this systematization, situating each of our works in this space.

Circumventing Heap Separation and Limited Capabilities. To bypass heap separation, prior work [13, 68, 97, 107, 125, 182, 184, 185, 310, 319] has demonstrated and performed cross-cache reuse, a technique that exploits the kernel's page allocator to reuse memory across separated allocator caches. These approaches were powerful but often restricted to dedicated caches or had low success rates. Failure can result in crashes that undermine stealth, a key factor since it increases the risk of detection and

forensic investigations. To address these challenges, we introduce SLUBStick [195], a reliable cross-cache reuse exploitation technique leveraging limited capabilities from heap vulnerabilities. SLUBStick first leverages a timing side channel in the slab allocator to reliably detect memory reuse, thereby significantly improving the success rate of cross-cache reuses. It then leverages common kernel code patterns to transform weak heap write primitives into powerful page-table manipulations, enabling arbitrary kernel memory access. We show SLUBStick's efficacy on one synthetic vulnerability and nine real-world CVEs, achieving privilege escalation on modern Linux kernels with state-of-the-art defenses. This paper was published at USENIX Security 2024 [195] in collaboration with Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard.

Leaking Locations via Software-Induced Side Channels. While prior work [88, 92, 131, 172, 195, 236, 263, 304, 340, 341] has introduced multiple software-induced side channels, most of these have focused on leaking data or behavioral patterns from user-space processes. Only SLUBStick [195] and PSPRAY [172] have demonstrated side channels that target the kernel, are induced by kernel software, and improve heap manipulation attacks. Building on this line of work, we introduce KernelSnitch [197], a novel software-induced side channel that exploits timing differences when accessing kernel data structures such as hash tables and trees. By amplifying this timing leakage, KernelSnitch enables the leakage of user data through covert channels and website fingerprinting. Leveraging the specific indexing mechanisms of Linux hash tables, it also reveals the location of kernel heap objects, making it the first side-channel attack to achieve this. This paper was published at NDSS 2025 [197], in collaboration with Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard.

Leaking Locations via Hardware-Induced Side Channels. When considering hardware-induced side channels for kernel information leakage, attacks often exploit the TLB, which caches recent virtual-to-physical address translations. This TLB side channel [114] allows the distinction between mapped and unmapped memory regions. It enables unprivileged users to infer kernel memory mappings, such as the base address of the kernel code or the physical map. Several follow-up works extended this technique [85, 89, 159, 186, 188, 286, 297, 347], while still being limited to only leak the coarse-grained location of memory sections. Expanding on this foundation, we show that certain design decisions in defenses and kernel memory allocators can amplify TLB-based leakage [196]. We then

use kernel allocator massaging to craft the internal allocator state, and combine it with an Evict+Reload TLB side-channel attack to leak the locations of kernel objects. Specifically, on recent Intel CPUs, we leak the base addresses of all major memory sections and the fine-grained locations of kernel heap objects, page tables, and kernel stacks. These disclosure attacks enable successful kernel-level exploitation on modern Linux systems, allowing novel exploit techniques and improving the reliability and stability of existing techniques. This paper was published at USENIX Security 2025 [196] in collaboration with Lukas Giner, Daniel Gruss, and Stefan Mangard.

1.1.2. Increasing Defense Effectiveness

When attackers exploit kernel vulnerabilities to achieve full system compromise, they typically follow one of two attack strategies: hijacking control flow to execute arbitrary code [21, 25, 29, 113, 141], or corrupting critical data to escalate privileges [36, 110, 314]. Although prior work has proposed various defense mechanisms, it remains uncertain whether these are sufficient to counter modern kernel exploit techniques. This part addresses both strategies by introducing mitigations that advance the state of the art in defending against control-flow hijacking and data-oriented attacks.

Mitigating Control-Flow Hijacking Attacks. Prior work [9, 53, 56, 77, 178, 220, 280, 288, 323] has proposed various kernel Control-Flow Integrity (CFI) defenses that limit control-flow transfers to predefined, valid targets. These include KCoFI [53], which restricts transfers to function entries, and FineIBT [220], which enforces CFI at function-signature granularity. Among these, only KCoFI fully protects against recent CFI-bypass attacks [128, 156, 170, 210, 211], which target the thread state. The thread state refers to the CPU register contents saved to memory during system events such as system calls, exceptions, or interrupts. Other defense approaches [77, 178, 280] provide only partial thread state protection [198], and only few [77, 178, 280] address signature-based CFI bypasses seen in the wild [19, 126]. To close this gap, we introduce HEK-CFI [198], which protects control-flow-relevant data during system events while offering fine-grained CFI. Specifically, HEK-CFI extends function-signature-based CFI by retrofitting Intel's new hardware isolation primitive (i.e., shadow stack) to fully protect backward- (e.g., return addresses) and additional, critical forward-edge (e.g., function pointers) control-flow transfers. Our

prototype eliminates as first kernel defense all illegal backward-edge targets, reduces the numbers of forward-edge targets compared to prior work, and protects thread state during system events, with a reasonable performance overhead. This work was published at AsiaCCS 2024 [198] in collaboration with Pascal Nasahl and Stefan Mangard.

Mitigating Data-Oriented Attacks. Prior work [14, 34, 35, 55, 59, 164, 166, 248, 250, 265, 305, 321] has protected critical kernel data objects from tampering. A notable example is xMP [248], which secures credentials, page tables, and address-space structures, though with considerable performance overhead. While these efforts represent important progress, they fall short of offering strong security guarantees for multiple sensitive kernel objects without incurring significant performance costs. To address this, we introduce DOmain Protection Enforcement (DOPE) [199]. DOPE leverages Intel's Protection Keys for Supervisor (PKS) to enforce domain-based memory protection. This restricts access to sensitive kernel data according to the principle of least privilege. Through a prototype implementation, we demonstrate that DOPE effectively protects multiple selected sensitive data objects while maintaining reasonable performance overhead, advancing in security relative to performance. This paper was published at ACSAC 2023 [199] in collaboration with Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard.

1.1.3. Revealing Shortcomings in the Android Ecosystem

Full-chain, in-the-wild exploits that target Android devices usually obtain initial code execution by exploiting vulnerabilities in applications with an interface to the outside world. These apps include browsers [126], messengers [122, 206, 231], and other exposed interfaces [120, 121]. Due to Android's strict security model, initial code execution occurs in an untrusted security context with a limited kernel attack surface. To fully compromise a device, threat actors typically either target the limited kernel surface directly or first exploit more privileged services to broaden the kernel attack surface. These exploit chains are often used to install spyware [276] such as Pegasus [123], Predator [206], or NoviSpy [120]. From the actor's perspective, compromising the kernel presents two major challenges: First, many components contributing to the kernel attack surface have undergone extensive security reviews, vulnerability discovery efforts, and patching to reduce exploitability. Second, modern kernels implement strong mitigations to limit exploitation.

This part addresses both challenges and analyses the real-world kernel attack targets and integration of defenses on Android devices. In doing so, it exposes a significantly more fragile state than previously assumed, revealing critical shortcomings in the Android ecosystem.

Analysing the Android Kernel Attack Targets. In 2023, a notable shift occurred in kernel exploitation: threat actors moved away from targeting the broader kernel surface via privileged services and instead began exploiting the GPU, which is accessible even from untrusted contexts. In fact, four out of five observed in-the-wild kernel compromises were attributed to GPU vulnerabilities [276]. In response, Google, Android's Red Team, and ARM prioritized GPU security through vulnerability detection and hardening efforts [317]. This shift raises a critical research question: Are GPUs the sole attractive target for threat actors, or do other kernel components pose similar or even greater risks that remain insufficiently addressed? To answer this, we conduct a large-scale analysis [194] demonstrating that the kernel attack surface accessible from untrusted contexts extends beyond GPUs. We identify several accessible drivers, including those for the DSP, JPEG decoder, and AI coprocessor. Then, we examine public resources, such as git histories and bug reports, to identify vulnerability patches in the drivers' code. Our subsequent patch inclusion analysis reveals that 61.4 % of analyzed Android devices were vulnerable to n-day vulnerabilities of any severity at the time of analysis. We also uncover critical findings regarding patch delays and dependencies, which vary across device vendors, chipset vendors, and vulnerability types. These insights underscore the urgent need for stronger security measures in Android, especially as concurrent research [120, 121, 130] has demonstrated the active in-the-wild exploitation of accessible drivers. This paper was published at USENIX Security 2025 [194] in collaboration with Florian Draschbacher, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard.

Analysing the Defense Integration against N-Days. To make kernel-level exploitation more difficult, security researchers have proposed numerous defenses, available in the upstream Linux kernel. However, the extent to which these defenses are integrated and effective in device vendor-supplied Android kernels remains unclear at scale prior to the Generic Kernel Image (GKI) project [8]. To address this gap, we present Defects-in-Depth [193], a two-part analysis of the security of downstreamed Android kernels from leading device vendors. In the first part, we evaluate the used exploit techniques from n-days and the mainline kernel defense-in-

depth mechanisms capable of preventing them. This allows us to quantify the maximum level of protection that can be achieved by enabling these defenses. In the second part, we assess how well these defenses are actually integrated by vendors. Our findings reveal substantial disparities: depending on the vendor, the real-world protection offered by downstream kernels is between 2.95 to 4.62 times worse than the maximum achievable baseline. We envision that our findings will guide the integration of effective defenses in vendor kernels, enhancing the security of Android devices. This paper was published at USENIX Security 2024 [193] in collaboration with Florian Draschbacher, Lukas Lamster, and Stefan Mangard.

1.2. Other Contributions

In addition to the main contributions, I contributed to five publications that advance the field of system security in three areas: enhancing side-channel attacks, uncovering weaknesses in mobile security features for Android and iOS, and improving memory safety.

Side-Channel Attacks. To enhance side-channel attacks, we first target a popular attack surface: Android mobile devices, in both local and remote attack scenarios. We then extend our work to desktop processors, specifically AMD CPUs, in remote attack scenarios.

For Android, we demonstrate that these mobile devices expose numerous power-related signals, enabling power side-channel attacks [232]. In a systematic analysis, we investigate unprivileged sensors that leak such power-related information. In a local attack scenario, we present a proof-of-concept AES attack that leaks individual key bytes. In a remote scenario, we show the leakage of pixels protected by cross-origin isolation. This paper was published at NDSS 2025 [232] in collaboration with Mathias Oberhuber, Martin Unterguggenberger, Andreas Kogler, and Stefan Mangard.

We advance the SQUIP attack [76], which exploits contention in AMD processor scheduler queues. In a remote setting, we demonstrate inter-keystroke timing attacks and a JavaScript-based covert channel. This paper was published at FC 2024 [75] in collaboration with Stefan Gast, Jonas Juffinger, Christoph Royer, Andreas Kogler, and Daniel Gruss.

Weaknesses in Mobile Security Feature. Mobile devices are highly attractive targets for attackers, with several real-world incidents observed [120, 121, 122, 168, 169, 203, 205, 231, 273, 274, 275]. We contribute novel attack techniques and large-scale analyses, responsibly disclose our findings, and help improve the security of mobile platforms.

We revisit the USB security model of mobile devices by examining the JuiceJacking threat, in which malicious chargers could compromise connected smartphones. Although major vendors introduced confirmation prompts in 2013 to mitigate such attacks, we discover that these defenses are insufficient. We introduce ChoiceJacking [61], which bypasses these JuiceJacking mitigations, effectively re-enabling the original attack. Our evaluation shows that ChoiceJacking enables access to sensitive user files, such as photos and documents, on all tested devices across 8 vendors. This paper was published at USENIX Security 2025 [61] in collaboration with Florian Draschbacher, Mathias Oberhuber, and Stefan Mangard.

In another study, we conduct a large-scale analysis [60] of code transparency in Android Application Bundle (AAB)s. We identify multiple flaws and demonstrate attacks that achieve code execution or unauthorized data access in target apps. This paper was published at ACSAC 2024 [60] in collaboration with Florian Draschbacher.

Memory Safety. To improve memory safety, we introduce Cryptographic Least Privilege Enforcement (CLPE) [289], a lightweight ISA extension that uses message authentication codes to enforce fine-grained memory isolation in C/C++ programs. CLPE enhances memory safety with minimal performance overhead and maintains compatibility with legacy software. This paper was published at HOST 2025 [289] in collaboration with Martin Unterguggenberger, David Schrammel, Lukas Lamster, Vedad Hadziz, and Stefan Mangard.

1.3. Outline

Chapter 2 provides background information on the memory management of modern computer systems, operating system architecture, kernel exploitation fundamentals, and side-channel attacks. Chapter 3 summarizes the state of the art of side-channel attacks that make kernel exploitation

more stable and reliable, recent kernel mitigations and defenses, and analyses of the Android kernel security landscape. Chapter 4 concludes the thesis and discusses potential future research.

2

Background

This chapter provides the necessary background for this thesis. Section 2.1 describes the memory management of modern computer systems, including virtual memory and the separation between user and kernel space. Section 2.2 presents the architecture of modern operating systems, focusing on the Linux kernel. Section 2.3 introduces the fundamentals of kernel exploitation as well as covers modern kernel attacks and exploit techniques. Finally, Section 2.4 analyzes side-channel attacks on the Linux kernel, both software-induced and hardware-induced.

2.1. Memory Management in Modern Computer Systems

Modern computer systems rely on a complex and layered architecture to enable efficient multitasking, protection, and resource management. These systems typically comprise of three levels: the application, the operating system and hardware level. At the core of this architecture is memory management, which is built on the concept of virtual memory (see Section 2.1.1). Virtual memory abstracts the underlying physical memory and is managed by the operating system. This abstraction allows to separate different user processes from each other as well as the user from the kernel space (see Section 2.1.2).

2.1.1. Virtual Memory

Virtual memory is a fundamental abstraction in modern computer systems. It provides each process with the illusion of a large, contiguous, and private address space, independent of the actual limitations of physical memory [49, 146]. A process is instantiated when a user launches a program, such as

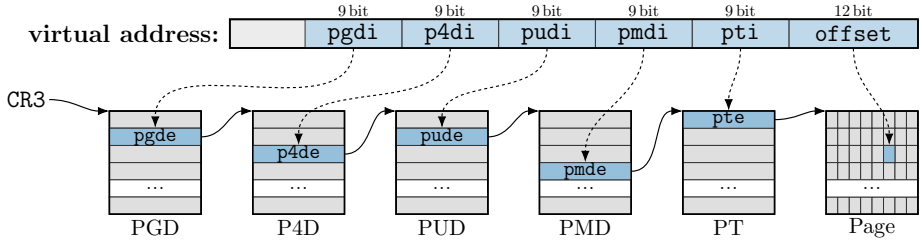


Figure 2.1: The address translation from the virtual to its physical address using a 5-level paging hierarchy [117], assuming a page size of 4 kB on a x86_64 system.

opening a web browser. The operating system loads the program's code into memory and assigns it a dedicated virtual address space. At a high level, virtual memory operates by translating virtual into physical addresses. The Memory Management Unit (MMU) performs this translation in cooperation with the operating system, which maintains per-process page tables. These page tables map virtual pages to physical pages and include essential metadata such as access permissions.

Figure 2.1 illustrates the process of translating a virtual to a physical address in a modern x86_64 system using five-level paging [117]. The architecture assumes a page size of 4 kB and supports virtual addresses up to 57 bits in width. In a five-level translation, the virtual address is divided into six logical segments: five 9-bit fields that act as indices into the page tables and a 12-bit physical offset that addresses the final byte. These five indices correspond to the five levels of page tables: the Page Global Directory (PGD), the Page 4th-level Directory (P4D), the Page Upper Directory (PUD), the Page Middle Directory (PMD), and the Page Table (PT). Each level contains 512 entries, each of which is 8 B in size. The 9-bit index selects a single entry from the corresponding table. The final 12-bit offset determines the byte-level position within the selected physical page.

On x86_64, the translation begins with the CR3 register [117], which holds the process's PGD base address. The most significant 9-bit segment of the virtual address, referred as `pgdi`, is used to select an entry in the PGD, referred to as the `pgde`. This entry points to the P4D, where the next 9-bit segment `p4di` is used to index into the directory. This procedure continues through each level: the PUD, PMD, and PT, using the corresponding index segments `pudi`, `pmdi`, and `pti`. Within the PT, the `pti` index selects

the final page-table entry, which contains the physical address of the 4 kB page. The least significant 12-bit `offset` of the virtual address are then added to this physical address to determine the byte in memory.

This hierarchical paging scheme enables efficient and flexible memory management [49]. It allows the operating system to allocate page tables on demand, thus avoiding the overhead of reserving large contiguous memory regions. Furthermore, it provides access control with permission bits on each 8 B page-table entry. These bits set access permissions such as read, write, or execute, enforcing memory protection policies at various granularities.

2.1.2. Address-Space Separation

Modern computer systems provide two address-space separations: First, they isolate distinct user-space processes [146]. Second, they isolate user-space processes and the operating system kernel. The kernel is the privileged core component of the operating system. It manages hardware resources and provides fundamental services such as memory management and hardware resource handling. Both separations are critical to prevent malicious interactions between processes and protect the kernel's integrity.

To achieve separation between user processes [146], each process is assigned its own virtual address space. This means that a process cannot directly access the memory of another process. The operating system, combined with the MMU, ensures that virtual addresses used by one process are mapped to physical memory independently of other processes unless explicitly declared otherwise, e.g., via shared memory. During a context switch between processes, the operating system changes the memory mappings by updating the page-table base register. This switching ensures that each process operates in an isolated address space.

The separation between user processes and the kernel is also enforced using virtual memory [52]. Page-table entries include a supervisor bit, which specifies whether a page can be accessed from the user mode or only from the high-privileged kernel mode. Pages marked as supervisor-only are inaccessible to processes running in user mode. This enforcement is carried out by the MMU [52], which checks access permissions for every memory access. If a process running in user-mode attempts to access a supervisor-only page, the processor triggers a page-fault exception. This exception

transfers control to the operating system's fault handler, allowing it to safely handle the violation.

2.2. Operating System Architecture

The architecture of an operating system defines how its components are structured and how they interact with both, user-space applications and the underlying hardware. Interaction with applications is facilitated via a system call interface [62] and an interrupt and exception mechanism [145] (see Section 2.2.1). The kernel manages communication with the hardware using data structures that represent the hardware resources (see Section 2.2.2).

2.2.1. System Calls, Exceptions, and Interrupts

System calls, exceptions, and interrupts are key mechanisms that enable applications to interact with the kernel and, therefore, hardware resources [62, 145].

System calls offer a controlled interface through which user programs request services from the kernel, such as file operations, memory management, or process control [62]. On system call invocation, the processor saves the current user context and switches from user mode to kernel mode via a defined transition mechanism [23]. The kernel then executes the requested service with high privileges. Once the system call completes, the processor restores the saved context, and resumes the application.

In this work, we refer to interrupts as asynchronous system events and exceptions as sychronous system events [23, 145]. These events are generated by hardware or software to signal the processor of conditions requiring immediate attention. Hardware interrupts are typically triggered by external hardware such as I/O devices or timers. They cause the CPU to pause execution, save the execution context, invoke an appropriate interrupt handler, and then resume the interrupted task. Exceptions, on the other hand, are synchronous events raised by the CPU when, e.g., an instruction cannot complete normally, such as a page fault or division by zero. Like interrupts, they prompt the processor to transfer control to a handler that resolves the fault before execution continues or terminates.

Notably, some system call invocation mechanisms are also implemented as software-generated exceptions. For instance, the legacy `int 0x80` instruction on x86 systems triggers a synchronous exception to transition into kernel mode, much like a typical fault.

2.2.2. Internal Structure

The internal structure of an operating system kernel is designed to manage hardware resources, support multitasking, and enforce protection boundaries, both between user and kernel space and across user processes [23]. This section first outlines the kernel's primary components, and then focuses on kernel memory allocators and kernel memory layout, which play a central role in kernel exploitation and, therefore, in this work.

At the core of the kernel are several fundamental components [23]:

- *Process and Thread Management:* The kernel maintains data structures such as process and thread control blocks to track the state of each executing entity. These structures store scheduling metadata, CPU register states, memory mappings, and permission information, enabling process creation, destruction, context switching, and scheduling.
- *Memory Management:* The kernel uses page tables and virtual memory mappings to manage address spaces for both user and kernel processes. These structures enforce memory isolation and enable features like memory protection and dynamic memory allocation.
- *File System and I/O Subsystems:* The kernel abstracts hardware devices and storage using file descriptors, along with internal representations of file system hierarchies. Device drivers interact with hardware to manage asynchronous operations and data transfers.
- *Device and Interrupt Handling:* The kernel maintains interrupt descriptor tables and registrations of per-device handlers to service interrupts. Associated data structures track the state of devices.
- *System Call Interface:* The kernel maps each system call identifier to a specific handler function, using a system call table. This provides an entry point into privileged operations.

These components interact via shared data structures, synchronization primitives, and event notifications. Together, they form the kernel's control over execution, memory, storage, and communication, building the foundation of user-space applications. As this thesis evaluates and experiments with Linux, we describe the Linux kernel from this point forward.

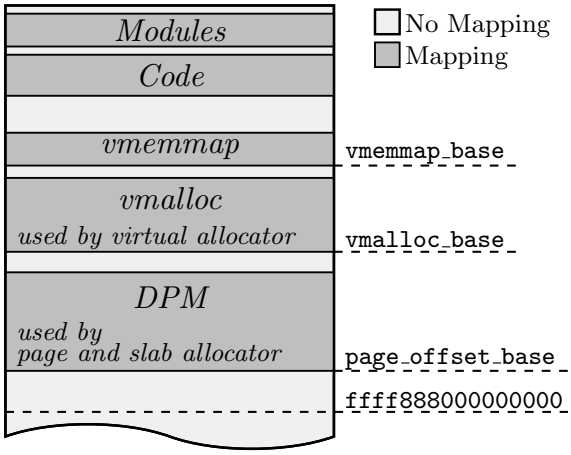


Figure 2.2: Virtual memory layout of the x86_64 Linux kernel [144].

Kernel Memory Allocators. The Linux kernel provides three primary kernel allocators [148]: the *page allocator*, the *slab allocator*, and the *virtual memory allocator*. These allocators operate on distinct regions of the kernel's address space [144], as illustrated in Figure 2.2.

First, the *page allocator* is the most fundamental and operates at the level of physical pages [148]. It allocates contiguous physical memory and is commonly used for lower-level kernel structures, such as physical-backed buffers for pipes. At its core, the page allocator manages memory within the Direct-Physical Map (DPM)[1], a linear virtual mapping of (typically) the entire physical memory. This memory space is divided into page-order chunks, where each chunk represents a power-of-two number of pages. The allocator supports both allocation and deallocation through free lists, combined with merging of adjacent free chunks to reduce fragmentation. Specifically, the kernel maintains global and per-CPU page-order free lists. Allocation requests are first served from the per-CPU lists. If these are exhausted, the allocator falls back to the global lists. Similarly, on deallocation, chunks are returned to the per-CPU lists unless those lists exceed a predefined capacity, in which case chunks are returned to the global free lists.

Second, there are three implementations of the *slab allocator*: SLAB, SLOB, and SLUB, with SLUB being the default in most modern Linux

[1]Although often referred to as the physical map, linear map, or direct mapping, we refer to it as the DPM throughout this work.

distributions [48, 148]. The slab allocator[2] builds on top of the page allocator by allocating page-order chunks for use as slabs, which are divided into smaller, fixed-size memory slots. The slabs are optimized for frequent allocations and deallocations of kernel objects, such as task structures and file objects. To minimize fragmentation and allocation overhead, the slab allocator maintains per-object-type caches. These come in two forms:

- *Dedicated caches* are used for object allocations of specific kernel object types with `kmem_cache_alloc`.
- *Generic caches* provide general-purpose allocation and deallocation interfaces such as with `kmalloc`. These caches handle allocation requests using preconfigured caches for common size classes from `kmalloc-8` to `kmalloc-8k`.

The SLUB allocator implementation enhances performance through several optimizations [148]. These optimizations include per-CPU caching of active slabs, per-allocation node management, and the use of partial slab lists for efficient management of free memory slots.

Third, the *virtual memory allocator* (or vmalloc allocator) handles memory requests for virtually contiguous, but not necessarily physically contiguous memory regions [148]. This is particularly useful for large buffers that cannot be satisfied by the page allocator due to fragmentation. The virtual memory allocator maps physical pages into a reserved region (see Figure 2.2) of the kernel address space using page tables, allowing flexible layout without the need for contiguous physical memory.

Together, these allocators provide the kernel with an efficient memory management subsystem. Their internal behaviors, particularly around memory reuse and metadata handling, are highly relevant for understanding memory-corruption vulnerabilities and exploitation techniques.

Kernel Memory Layout. Figure 2.2 illustrates the virtual memory layout of the Linux kernel on x86_64 systems [144], highlighting five key regions that are relevant for kernel exploitation. Starting from the highest kernel virtual addresses, the *Modules* and *Code* regions contain the executable instructions of dynamically inserted kernel modules and the kernel binary itself, respectively. Below lies the *vmemmap* region, which

[2]SLAB, SLOB and SLUB refer to different implementations, whereas slab (in lowercase) is commonly used to denote the generic memory allocator interface for managing caches of small memory objects.

provides a virtual mapping of metadata for each physical page frame. Specifically, it stores 64 B of metadata for each frame and is indexed by the physical frame number. Next, the *vmalloc* area holds memory regions allocated through the virtual memory allocator. Finally, the *DPM* covers a virtual mapping of (typically) the entire physical memory. It is directly used by the page allocator and includes memory chunks allocated by slab allocator. Crucially, the DPM serves as the main area for the kernel heap and most internal kernel data structures.

2.3. Kernel-Level Exploitation

Modern operating systems rely on the kernel to isolate privileged and unprivileged execution contexts. Attackers typically break this isolation by exploiting kernel-level vulnerabilities, allowing them to escalate privileges, bypass security mechanisms, and fully compromise the system. Kernel exploitation refers to this process of abusing bugs within kernel software to gain unauthorized control. Despite decades of research and widespread deployment of mitigations, real-world attacks continue to show that kernel exploitation remains practically feasible.

This section introduces key exploitation terminology (see Section 2.3.1), covering the threat model, and keywords such as exploit primitive and exploit technique. Building on this foundation, it then analyzes how attackers achieve privilege escalation through kernel exploitation (see Section 2.3.2).

2.3.1. Kernel Exploitation Terminology

Kernel attacks typically begin with the exploitation of vulnerabilities to gain an initial foothold in the kernel, with the ultimate goal of fully compromising the system. Given the complexity of modern Linux kernels and the variety of techniques used by attackers, clear and consistent terminology is essential. The field involves low-level system behavior, enforcement of security boundaries, and manipulation of kernel primitives, all of which demand precise language. To avoid ambiguity, we align our terminology with that used by Google Project Zero [15], a leading security research group known for its detailed analysis of real-world kernel vulnerabilities and exploitation techniques.

Threat Model. The threat model for kernel exploitation typically begins with code execution in an untrusted and unprivileged context [96, 120, 130, 184, 195, 332]. This initial execution is typically obtained by exploiting vulnerabilities in user-space applications that expose interfaces to the external world. Examples include web browsers like Chrome [126] and Firefox [272], messaging services like WhatsApp [122, 206] and iMessage [19], or USB interfaces [120, 121]. From there, threat actors typically aim to escalate privileges by exploiting vulnerabilities in the kernel [67, 97, 102, 126, 127, 128, 130, 183, 214, 227, 230, 242, 256, 310, 317, 342], whether in the kernel's core components or in subsystems such as device drivers. This has been shown by numerous in-the-wild attacks as documented in Google Project Zero's yearly reports [32, 268, 273, 274], as well as other research groups such as from Google Threat Analysis Group (TAG) [267], Amnesty International's Security Lab [120, 121, 123, 231], or Citizen Lab [168, 169, 206, 231]. Successful kernel exploitation typically allows threat actors to compromise the entire system.

Zero-Day and N-Day Vulnerabilities. Vulnerabilities that are actively exploited in the wild are generally categorized as either a zero-day or an n-day [15]. A zero-day vulnerability is one that is unknown to the vendor or public at the time of exploitation, meaning that no patch is available. These vulnerabilities are particularly valuable to threat actors due to their stealthiness. An n-day vulnerability[3] is a previously disclosed issue for which either no patch exists, or a patch exists but has not been applied to the target systems. Therefore, the system remains exploitable to this known vulnerability. Both zero-day and n-day vulnerabilities are often used in kernel exploitation [32, 203, 268, 273, 274, 276].

Vulnerability Classes. Vulnerabilities fall into various classes depending on the nature of the programming error [15]. Common classes include Use-After-Free (UAF), Out-Of-Bounds (OOB) read/write, Double Free (DF), Invalid Free (IF), uninitialized memory, and logic bugs such as improper permissions checks. A UAF occurs when memory is accessed after it has been freed. DF and IF are two types of memory deallocation error that can lead to UAF conditions or facilitate similar exploitation scenarios. In a DF, the same memory region is freed more than once, while an IF involves freeing an invalid pointer, such as one that points to an interior region of an object. An OOB access occurs when memory is read or written beyond the bounds of an allocated buffer. Each class offers

[3] An n-day vulnerability is also sometimes referred to as a one-day vulnerability.

Figure 2.3: The exploitation flow ***EF*** is a vulnerability-agnostic chain of exploitation techniques ***ET***. Each ***ET*** is a reusable and reasonably generic strategy to turn one exploitation primitive to a more powerful one [15]. Ultimately, the ***EF*** leverages the capabilities of one or more vulnerabilities—such as UAF write—to gain root privileges—such as via an arbitrary read/write primitive.

distinct exploitation surfaces and challenges, shaping the strategies used in constructing exploits.

Exploit Primitive, Technique, and Flow. In the context of kernel exploitation, an exploitation primitive is a generic capability that an attacker can gain through a vulnerability [15]. Examples are an n-byte linear heap overflow, arbitrary memory read/write, program counter control, or arbitrary code execution. These primitives serve as essential building blocks to achieve kernel compromise.

An exploitation technique is a general, reusable strategy used to elevate a primitive to a more powerful form [15]. One example is Return-Oriented Programming (ROP) [25], which elevates a program counter control primitive to arbitrary code execution. These techniques are not tied to a specific vulnerability and can be applied across different scenarios.

The exploitation flow represents a chain of techniques [15] (see Figure 2.3) that progressively increase the capabilities of the exploit primitives, ultimately compromising the kernel. The exploitation flow allows attackers to transform limited control into root privileges and full system compromise, as observed in several in-the-wild attacks [32, 268, 273, 274, 276].

2.3.2. Kernel Attacks and Exploitation Techniques

Despite increasingly robust kernel-level defenses, security experts continue to show that weaker [184, 195, 211, 332] or even previously considered unexploitable [129] vulnerabilities can still lead to a full kernel compromise. In the Linux kernel, the most common type of vulnerability involves memory corruption, typically through unauthorized modification of memory regions. Examples of memory-corruption vulnerabilities include UAF, DF, IF, OOB accesses, and uninitialized memory usage, as described in

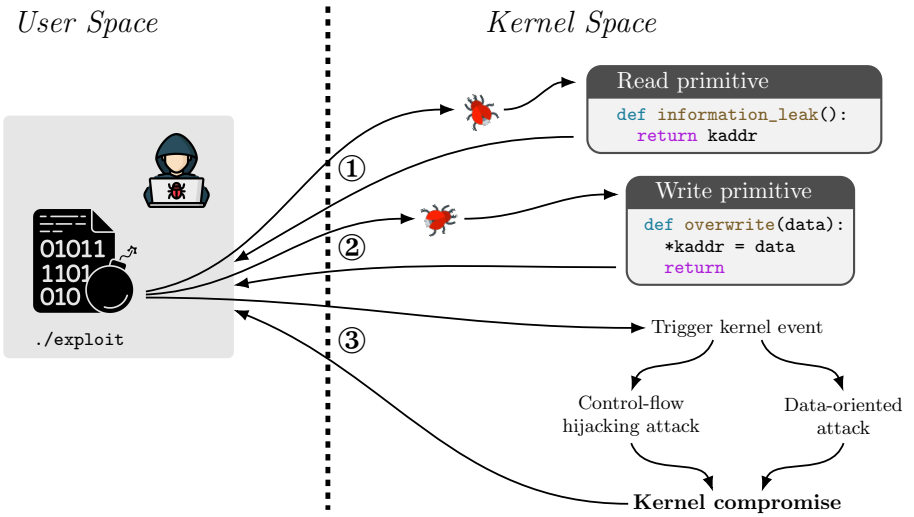


Figure 2.4: **Kernel compromise:** first stage exploits a vulnerability for a read primitive; second stage exploits the same or another one for a write primitive; third stage triggers a kernel event which uses the corrupted data to perform a control-flow hijacking or data-oriented attack.

Section 2.3.1. Prior work has shown that these vulnerabilities allow for robust exploitation primitive used in kernel compromise. For instance, Wu et al. [312] presented a framework to facilitate kernel UAF exploitation, while Chen et al. [38] designed KOOBE to assist exploitation of OOB vulnerabilities. Chen et al. [41] introduced SLAKE which facilitates slab manipulation for kernel exploitation, particularly targeting the slab allocator. Other works focused on exploiting uninitialized memory [42, 138, 192] or information leakages [181].

End-to-end kernel compromises typically involve three stages as illustrated in Figure 2.4. First, an attacker exploits a vulnerability to obtain a kernel read primitive [193, 277, 293]. Using this read primitive allows to locate target objects and break Kernel Address Space Layout Randomization (KASLR) [65], a defense that randomizes the memory layout and objects in the kernel. In the second stage, the attack continues with exploiting a vulnerability to obtain a kernel write primitive, where the vulnerability is the same or a different one as in the first stage. Using this write primitive allows to change the value of the leaked object's members to attacker-controlled ones. Finally, the third stage triggers a kernel event which internally uses the overwritten object to perform a kernel

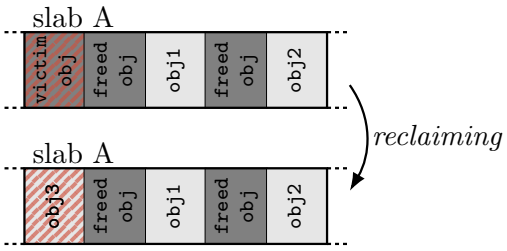


Figure 2.5: **In-cache reuse:** the `victim`'s memory slot is reclaimed for `obj3`.

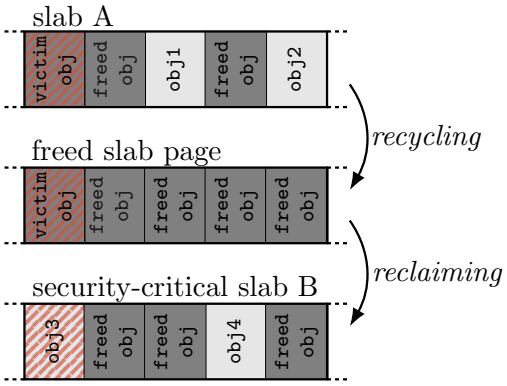


Figure 2.6: **Cross-cache reuse:** the `victim`'s memory slot is reclaimed for `obj3` from the security-critical slab B.

control-flow hijacking or data-oriented attack, ultimately compromising the kernel.

When considering exploitation of memory-corruption vulnerabilities for the read or write primitive, most kernel exploits follow a similar exploitation flow: An attacker triggers a vulnerability to prematurely free an object, often referred to as the *victim object*. The attacker then reuse the victim's memory slot and reallocate it as a different object. There are two primary reuse variants: First, in-cache reuse (see Figure 2.5) reallocates the victim's slot as another object from the same slab cache, e.g., `kmalloc-*`. However, this method is constrained by the Linux kernel's design, which separates kernel objects into distinct slab caches based on size and allocation context. This heap separation prevents an attacker from reusing the victim's memory slot for a security-critical object [46, 51, 191], which we detail later in Section 3.2.1. Second, cross-cache reuse (see Figure 2.6) bypasses this heap separation. In this method, the attacker frees all objects on the

slab page containing the victim's slot. This triggers slab page recycling, returning the page to the page allocator. Then, the attacker reclaims the freed page for a security-critical slab page, with its memory slots then used for security-critical objects.

In addition to object-slot reuse, attackers can exploit prematurely freed data pages to carry out page-level UAF attacks [108, 185, 249, 346]. One example of this type of data page is the pipe page-backed buffer. In these reuse attacks, the freed data page is maliciously reclaimed, often as a slab page for a security-critical slab, enabling access or corruption of security-critical data.

These reuse techniques typically rely on the availability of objects with specific properties. In particular, two types of objects are relevant: *spray objects* and *target objects*. *Spray objects* are used for heap spraying [331], an exploit technique that floods the slab cache with many controlled allocations. It increases the likelihood that a *target object* will occupy the victim's memory slot, where the target object contains security-sensitive fields. These fields are typically data or function pointers. Overwriting them enables arbitrary read and write primitives or hijacks control flow. Identifying suitable objects for spraying or for targeted overwrites is a key part of exploitation. To assist with this, prior work has introduced several approaches for finding such objects [12, 40, 195, 299, 315, 331].

While these techniques form the foundation for constructing robust exploit primitives, real-world kernel exploitation often faces additional challenges. Most discovered vulnerabilities only grant weak overwriting capabilities, often tied to tight race windows, and exploits are typically tailored to specific kernel versions. To overcome these challenges, research has advanced in four key areas. First, discovering new vulnerabilities remains a foundational challenge. Techniques such as fuzzing [17, 37, 104, 157, 222, 278, 282, 284, 298, 313, 322, 329, 345, 349], static analysis [179, 224, 334, 335, 336, 337], dynamic instrumentation [83, 101, 158, 223], or a combination of these, have become crucial tools for systematically identifying bugs in the kernel, an essential step toward kernel compromises. Noteworthy developments include Syzkaller [296], a widely-used coverage-guided kernel fuzzer, and runtime sanitizers such as the Kernel Address SANitizer (KASAN) [247], which detect memory safety violations during execution. Second, multiple exploit techniques have been proposed [96, 184, 195, 211, 230, 233, 242, 310, 332] to transform weak exploit primitives from discovered vulnerabilities into stronger ones. An example of a weak primitive is overwriting the limited to n-bytes within a slab cache at a

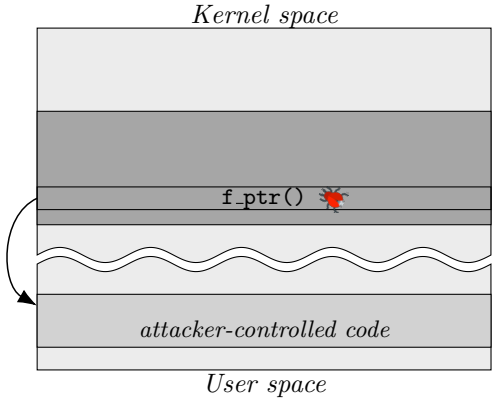


Figure 2.7: **ret2usr [141]:** the control flow is redirected to attacker-controlled user-space code to gain kernel code execution.

specific offset [242]. Strong primitives, on the other hand, typically provide arbitrary read and write capabilities or enable control-flow hijacking. Third, for vulnerabilities constrained by tight race windows [105, 242, 310], several techniques have been introduced to widen the race window [105, 173, 174, 252, 333]. These techniques include enforcing CPU cache misses and deliberately preempting the target thread using mechanisms such as `timerfd`-based interrupts. Fourth, to generalize the applicability of kernel exploits, recent work has focused on assessing exploitability across different kernel versions and configurations [132, 348]. These efforts help bridge the gap between bug discovery and practical exploit development.

With the fundamentals of vulnerability types and exploit strategies, we now focus on three areas relevant to our work regarding kernel exploitation:

Control-Flow Hijacking Attacks. In the kernel, control-flow hijacking attacks [141] aim to redirect execution to attacker-controlled code or code gadgets by corrupting control data such as function pointers, return addresses, or data-to-function pointers. A common goal of these attacks is to execute privileged kernel functions to escalate the process's privileges. A widely used method for privilege escalation involves invoking the kernel's credential management interface using `prepare_kernel_cred` and `commit_creds`. These functions create a new set of root credentials and assigns them to the current task. This effectively grants the process full root access within its current namespaces. In cases where the process is containerized, the attack continues with namespace escape techniques to access the host's global namespaces and break out of the container.

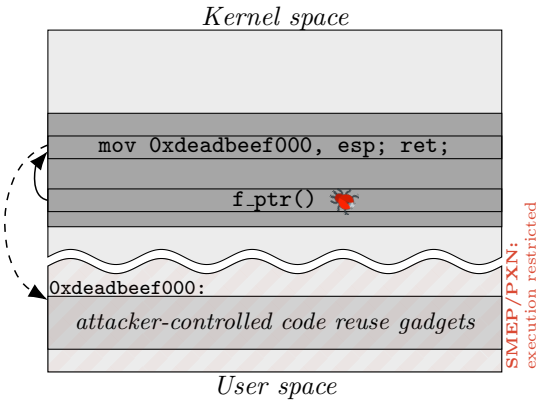


Figure 2.8: **pivot2usr [140]:** the control flow is redirected to a stack pivoting gadget that overwrites the kernel stack pointer, causing it to refer to attacker-controlled code reuse gadgets in user space to gain kernel code execution.

Control-flow hijacking attacks [21, 25, 29, 113, 141] have evolved significantly over time in response to increasingly sophisticated kernel defenses. Early techniques achieved arbitrary kernel code execution by redirecting execution to attacker-controlled user-space code (i.e., ret2usr [141]; see Figure 2.7). Subsequent techniques performed code reuse attacks by leveraging attacker-controlled gadgets located in user space (i.e., pivot2usr [140]; see Figure 2.8). These early methods were mitigated by hardware-enforced protections such as Intel SMEP/SMAP [52] and ARM PXN/PAN [221], which restrict user-space memory execution and access while in kernel mode. To counter these defenses, modern hijacking techniques leverage code reuse gadgets stored in kernel objects or in the Direct-Physical Mapping (DPM) (i.e., ret2dir [140]) as shown in Figure 2.9. These attacks typically gain control by performing stack pivoting, overwriting the stack pointer with an attacker-controlled memory location containing the code reuse gadgets. Attackers have also exploited other subsystems such as the Berkeley Packet Filter (BPF) [31, 135], an in-kernel execution engine that allows user-defined programs to run safely in the kernel. Some attacks corrupt BPF programs or use them to store gadgets [31].

A common method to achieve malicious redirection of kernel control flow to attacker-controlled gadgets involves overwriting function pointers as with `f_ptr` of Figures 2.7 to 2.9. These function pointers are then called during kernel events such as on pipe closure [187, 227] or via networking

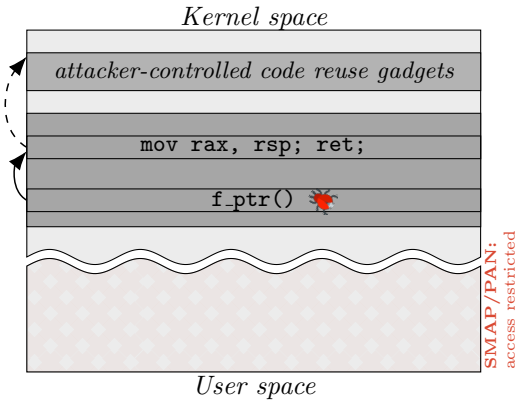


Figure 2.9: **Recent hijacking attacks:** the control flow is redirected to a stack pivoting gadget that overwrites the kernel stack pointer, causing it to refer to attacker-controlled code reuse gadgets in kernel space to gain kernel code execution.

subsystems [58, 98]. Beyond direct function pointer overwrites, attackers also target data pointers that reference function pointers, such as operation table pointers refer to an array of function pointers [19, 126, 198, 259]. Another popular technique targets the global variable `modprobe_path` [177, 290, 330] or `core_pattern` [147], enabling execution of arbitrary user-space binaries with root privileges, ultimately leading to full system compromise.

More advanced exploitation techniques improve stability and cross-version reliability of kernel control-flow hijacking. KEPLER [311] converts control-flow hijacking primitives into more powerful stack-overwriting capabilities. RetSpill [332] demonstrates how spilled CPU registers during system calls can be leveraged to build reliable gadget chains. Other techniques hijack system registers to disable protections like Intel SMEP/SMAP [156] or exploit interrupt-spilled registers [128] for kernel control-flow hijacking. Miller et al. [211] further generalize these methods, demonstrating how register corruption can enable reliable hijacking.

Data-Oriented Attacks. Beyond control-flow hijacking, attackers can escalate privileges by using an arbitrary memory read and write primitive to manipulate sensitive kernel data. A particularly attractive target is the `cred` structure, which holds a process's credentials, including user and group IDs, capability sets, and security contexts. These attacks typically begin by using the arbitrary read primitive to locate the current process's credential structure. The attack then modifies the `cred` structure in place

using the arbitrary write primitive. Alternatively, it uses the read primitive to locate the `cred` structure of a privileged process and overwrites the pointer in their own task to reference it.

Techniques for obtaining arbitrary read and write primitives have evolved significantly in response to defensive mechanisms. One notable technique involved abusing the `addr_limit` field [270], which defines the maximum accessible user-space address for copy functions. By maliciously manipulating `addr_limit`, attackers could trick copy functions (e.g., `copy_*_user`) into reading from or writing to arbitrary kernel memory. This technique was used in multiple kernel exploits, including the 2019 in-the-wild Android exploit BadBinder [193, 270]. However, it was eventually mitigated by removing `addr_limit` functionality altogether. Subsequent techniques have adapted similar ideas by targeting data pointers or fields that serve comparable roles with which legitimate system calls interact. These techniques typically use a limited write primitive to heap allocated kernel memory, overwriting data pointers or fields in objects such as `msg_msg` [196, 242, 315], `file` [262, 269], `seq_file` [256, 285], or `pipe_buffer` [96, 138, 185, 196, 251]. These objects can be interacted through the system calls, allowing user space to interact with the corrupted pointers. Therefore, by carefully crafting these interactions, attackers can effectively gain the desired arbitrary read and write primitive. In addition to pointer manipulation, other exploits target page-table entries. Examples include corrupting PT entries in Dirty PageTable [310], overlaying PMDs with PTs in Dirty PageDirectory [230], or exploiting write primitives to PT, PMD, or PUD entries in SLUBStick [195].

When considering Android kernel exploitation, Yong et al. [326] introduced the Kernel-Space Mirroring Attack (KSMA), a powerful Android exploit technique that transforms a limited write primitive into direct manipulation of kernel code. This is achieved by altering an AArch64-specific kernel PGD to expose kernel code to user space, enabling kernel code tampering. KSMA techniques have been widely adopted in Android kernel exploits [99, 193, 327]. While being patched [324] or mitigated [59, 112], alternatives may be still mountable [190, 193].

Notably, Lin et al. [184] demonstrated alternative paths for privilege escalation to an arbitrary read and write primitive in their work, DirtyCred. DirtyCred exploits credential reuse by replacing a process's maliciously freed `cred` structure with that of a privileged task. This bypasses traditional memory corruption by exploiting data manipulation, enabling privilege escalation through direct kernel data tampering.

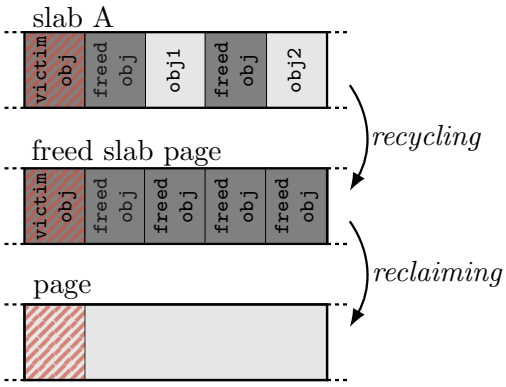


Figure 2.10: **Page-level cross-cache reuse:** the `victim`'s slab page is reclaimed for general-purpose page allocation.

Cross-Cache Reuse Attacks. Cross-cache reuse is a powerful exploit technique. It allows attackers to convert an attack capability from a vulnerable slab cache to a security-critical cache, bypassing heap separation (see later in Section 3.2.1). This reuse occurs when a slab page, once freed, is reclaimed by another slab cache (see Figure 2.6). Alternatively, the prior slab page is reclaimed by the page allocator for general-purpose allocation, referred to as page-level cross-cache reuse (see Figure 2.10). Xu et al. [319] introduced the concept of cross-cache reuse. However, their approach suffered from low success rates due to unpredictable allocations and deallocations by other tasks. Horn et al. [107] later proposed a more reliable strategy for slab page recycling and reclaiming. Subsequent kernel exploits have leveraged cross-cache reuse to reclaim slab pages into various security-critical caches, including `msg_msg` [13, 68, 125, 240, 315], `pipe_buffer` [301], credential reference object [137, 184], `eventpoll_epi` [285], or `urb` [315]. For page-level cross-cache reuses, examples include reclaiming the slab page for a backed pipe page [96, 97, 185] or page tables [107, 195, 310]. Most of these exploits recycle slab pages from dedicated caches, which are more predictable, due to less noise from unrelated allocations and deallocations [182, 195, 319]. To improve the reliability of cross-cache reuse particularly from generic caches, SLUBStick [195] presents a side-channel attack on the slab allocator. Other techniques extend cross-cache reuse to exploit slab pages across different CPU caches [43] and slab page sizes [309].

2.4. Side-Channel Attacks

A side channel is an unintended information channel that leaks metadata during the execution of a program. This metadata arises from observable side effects of computation and is typically not part of the program's intended output. Common forms of side-channel metadata include execution time [152], power consumption [154], electromagnetic radiation [4], and temperature [115]. Side-channel attacks exploit this leaked metadata to infer sensitive information. For example, by observing small variations of the time a cryptographic algorithm takes to execute, an attacker may be able to recover secret keys [325]. These attacks do not target the mathematical foundations of cryptographic algorithms. Instead, they exploit weaknesses in their implementation, often through low-level hardware or system behavior. Other examples of side-channel attacks include covertly transmitting data [207], spying on user input [91], or breaking Address Space Layout Randomization (ASLR) [114], a software defense that aims to make exploitation harder. Since this thesis exploits timing as an observable metadata, this section will focus on timing side channels.

Timing side channels can arise from various system behaviors, including caching mechanisms [152], weak cryptographic routines [234], insecure data structure implementations [106], or weak algorithm implementations [18, 139, 254, 260]. As a representative example, consider caching: caches are designed to avoid repeated access to slower memory resources, thereby improving system performance. Instead of always fetching data from main memory, caches store local copies of frequently accessed data, offering a faster path. However, this optimization introduces timing differences between cache hits and misses, where the data is stored in the faster or slower path, respectively [152]. These variations can leak access patterns that attackers can monitor to infer sensitive information. Such timing side channels may originate from both, hardware [325]—such as CPU caches that store data to avoid accessing slower RAM—and software [234]—where frequently used objects are stored to speed up repeated operations.

When considering hardware-induced side channels, multiple cache-like components have been shown to introduce security issues. These include data caches, which accelerate access to frequently used data. They have been exploited in attacks like Prime+Probe [234] and Flush+Reload [94, 325], which infer victim access patterns based on timing differences. Beyond data caches, the Translation Lookaside Buffer (TLB), which caches recent virtual-to-physical address translations, also leaks timing information

through hit/miss differences [89]. By observing these differences, attackers can infer memory mappings and bypass defenses such as KASLR [114], making kernel-level exploitation more reliable (see Section 3.2). Other microarchitectural features can also leak information, including branch prediction units [70, 71], execution port contention [5], and scheduler contention [75, 76]. Notably, timing-based side channels have played a critical role in enabling speculative execution attacks such as Spectre [153] and Meltdown [90], which exploit transient execution [26] to leak sensitive data via observable microarchitectural effects.

Similar to hardware-induced side channels, software-level components can also introduce observable timing differences that depend on secret or privileged information. These software-induced side channels may arise at both, the application and operating system level, leaking sensitive data through subtle variations in execution time. A wide range of software abstractions have been shown to be vulnerable to such attacks. Examples include software caches [22, 73, 88, 292], memory allocators [172, 195], synchronization primitives [131, 236, 263, 340], interrupt handling mechanisms [57, 281], and implementations of data structures [106, 197].

3

State of the Art

This chapter surveys recent advances in kernel security, with a focus on offensive and defensive techniques, and positions the contributions of this thesis within the broader state of the art. The discussion is organized around three key areas, aligned with the thesis's contributions. Section 3.1 explores side-channel-assisted kernel attacks to enhance the reliability of kernel-level exploitation. Section 3.2 presents recent work on defenses against kernel-level exploitation. Section 3.3 evaluates broader studies of kernel security resilience, with a focus on the Android landscape.

3.1. Side-Channel-Assisted Kernel Attacks

Traditional kernel exploits (see Figure 2.4) have predominantly relied on memory-corruption vulnerabilities to gain unauthorized read and write primitives. However, attempts to trigger such vulnerabilities inherently risk corrupting unintended memory, which can destabilize the kernel and potentially lead to system crashes. This instability stems from two key factors. First, exploitation typically involves massaging the kernel's internal memory state to improve the successful reclamation of specific memory slots. A common technique is heap spraying [331], which aims to coerce the kernel's memory allocator into placing attacker-controlled objects at targeted locations. However, since the allocator's internal state is opaque, attempts to reclaim specific memory slots can be prone to failure [172, 331]. If the memory slot is reclaimed incorrectly, the exploit may access unrelated or unpredictable objects, often resulting in a crash. Second, even after successful memory reclamation, converting vulnerabilities into reliable read or write primitives remains complex. It is particularly challenging to obtain a read primitive at the early stage of the exploit because the attacker has minimal control and limited visibility into the kernel's internal state. This makes the process fragile and prone to failure.

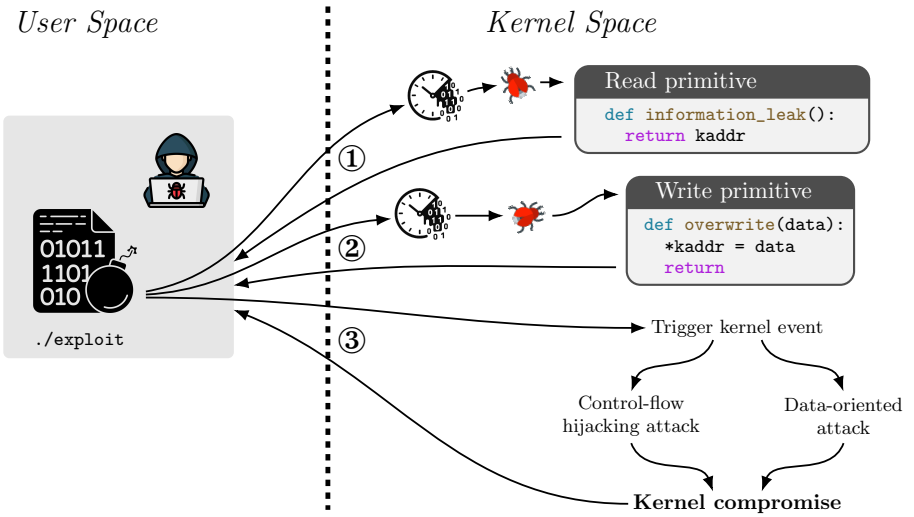


Figure 3.1: **Side-channel-assisted kernel exploitation:** The first and second stage use a side channel to increase the success rates of vulnerability exploitation for a read and write primitive, respectively. The third stage triggers a kernel event which uses the corrupted data to perform a control-flow hijacking or data-oriented attack.

This fragility poses a significant problem. In the context of kernel exploitation, a system crash is not merely a technical failure. It compromises stealth, increases the risk of detection, and can trigger forensic investigations [123, 130, 204]. As a result, enhancing the reliability and robustness of end-to-end kernel compromises is a critical objective. To achieve this goal, side-channel techniques have emerged as a promising solution. Side channels can reveal parts of the kernel's internal state by passively observing information leaks such as timing variations, cache behavior, or access patterns. This capability offers two key advantages: First, it can refine heap spraying or object targeting strategies, increasing the success rate of exploitation techniques [172, 195] (see Figure 3.1). Second, it can partially [89, 114, 189] or fully [196, 197] replace the need for a read primitive, offering a more stable and reliable alternative (see Figure 3.2). Both benefits represent a significant improvement. Unlike memory-corruption-based techniques, side channels can be triggered arbitrarily without risking a system crash, making them a safer and more controlled technique for kernel exploitation.

Side-channel attacks can stem from both software and hardware sources, with several demonstrated techniques targeting the Linux kernel. When

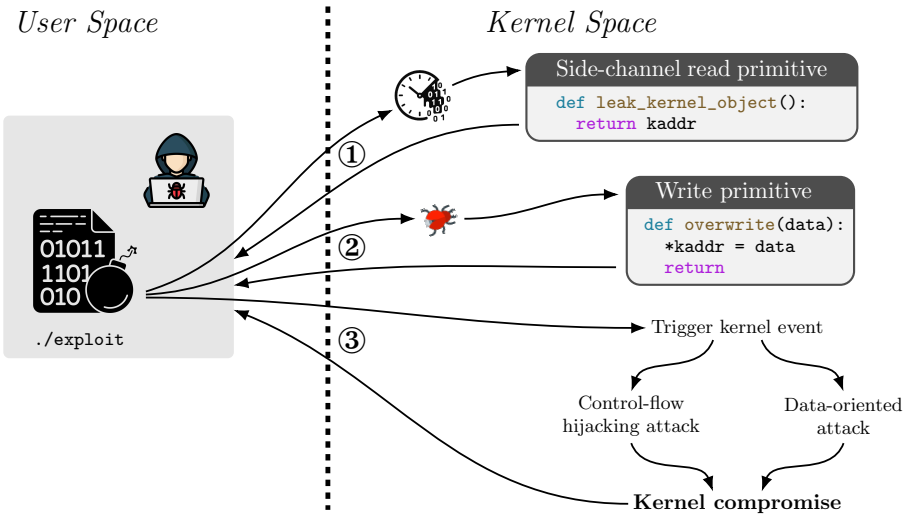


Figure 3.2: **Kernel exploitation with a side-channel read primitive:** The first stage uses a side channel as a read primitive. The second stage exploits a vulnerability for a write primitive. The third stage triggers a kernel event which uses the corrupted data to perform a control-flow hijacking or data-oriented attack.

considering software-induced side channels, prior work [33, 88, 92, 131, 197, 236, 261, 263, 281, 304, 340, 341] has mostly focused on leaking user data or behavior, such as via covert channels or fingerprinting. In contrast, a more recent line of work explores software-induced side channels that enhance kernel exploitation [172, 195, 197]. Lee et al. [172] introduced PSPRAY, a timing side channel on the slab allocator that different between object allocation timings: fast allocations indicate reuse within the same slab page, while slower allocations indicate a fallback to the page allocator. This information improves the success rate of exploit techniques that rely on in-cache reuse, e.g., those that exploit OOB or UAF vulnerabilities.

Contribution 3.1

With SLUBStick, Maar et al. [195] extended the slab timing side channel to work across multiple allocator caches, enabling reliable and stable cross-cache reuse attacks. This significantly improves the success rate of multiple exploit techniques (see Figure 3.1). Maar et al. then showed how this side channel reliably enables the conversion of constrained heap write bugs into powerful page-table manipulation primitives, ultimately allowing an arbitrary read/write primitive.

Contribution 3.2

KernelSnitch [197] continues this line of research and enables the replacement of traditional read primitives with a side-channel-based method (see Figure 3.2). It exploits timing variations in kernel data structure access and exploits how the kernel indexes hash tables, to infer internal memory layout. KernelSnitch demonstrates the first kernel heap pointer leak via a side channel. This allows attackers to leak kernel heap object locations without triggering potentially unstable vulnerabilities.

In the domain of hardware-induced side channels, extensive research has exploited the TLB, a CPU cache that stores virtual-to-physical address mappings. These attacks exploited timing differences between cached and uncached address translations to infer kernel memory locations and, thereby, break KASLR. Prior work [27, 89, 114, 124, 186, 189] showed that TLB-based side channels can leak the base addresses of kernel regions, such as the kernel code, modules, and the DPM. Wang et al. [297] then generalized the leakage attack with the page-walk side channel. These leak primitives have been successfully used in real-world kernel exploits [82, 128, 187, 188], offering a more reliable and stable alternative to vulnerability exploitation. Subsequent research [85, 86, 159, 286, 347] extended these techniques and reverse-engineered TLB behavior on more recent CPUs.

Contribution 3.3

Most recently, Maar et al. [196] demonstrated that combining kernel allocator massaging with a TLB side channel enables fine granular location disclosures. Their method leaks the base addresses of all major memory sections, as well as the locations of kernel heap objects, kernel stacks, and page tables. These fine-grained disclosures effectively replace the need for traditional read primitives, as shown in Figure 3.2. As a result, this approach enables new reliable exploit techniques and significantly improves reliability of existing ones [196].

3.2. Defenses against Kernel-Level Exploitation

In this section first, we review the defenses that have been incorporated into the mainline Linux kernel, are currently in use, and are under consideration (see Section 3.2.1). We then discuss academic approaches to

mitigating kernel-level exploitation of memory-corruption vulnerabilities (see Section 3.2.2). Finally, we survey bypasses of mainline and academic defenses (see Section 3.2.3).

3.2.1. Mainline Kernel Defenses and Hardenings

The Linux kernel incorporates upstream defenses to make kernel exploitation more difficult. While we discuss key mainline protections in this section, a comprehensive list can be found in public resources such as the Kernel Self-Protection Project (KSPP) [143] and the kernel-hardening-checker [243].

Limiting Control-Flow Hijacking Attacks. Early control-flow hijacking techniques involved injecting code into kernel memory. To prevent these injection attacks, Linux introduced a W^X policy that enforces the simultaneous non-executability and non-writability of memory [64]. Despite this, attackers could still perform control-flow hijacking attacks [21, 25, 29, 113, 141]. They redirected control flow to user space (e.g., ret2usr [141]; see Figure 2.7) or used user-space-stored code reuse gadgets (e.g., pivot2usr [140]; see Figure 2.8). To mitigate such attacks, processor vendors presented hardware-enforced defenses [52, 221] to restrict user-space code execution and data access in kernel mode with Intel SMEP/SMAP [52] and ARM PXN/PAN [221], respectively. As a result, sophisticated control-flow hijacking attacks now rely on storing the code reuse gadgets in kernel space and stack pivoting to them [311, 332] (see Figure 2.9). To limit these hijacking attacks, Linux incorporates kernel Control-Flow Integrity (CFI) [9, 65, 220]. Kernel CFI restricts kernel control-flow transfers to a predefined Control-Flow Graph (CFG), which is constructed statically at compile time or dynamically at runtime. It protects forward edges (e.g., indirect calls of function pointers) and/or backward edges (e.g., function returns), ensuring that control-flow transfers are limited to valid targets in the CFG. As of this writing, kernel software such as that used in Android [9] and supported by recent Intel platforms [220] enforces forward-edge kernel CFI with function-signature granularity. Here, control-flow transfers are only allowed when the signature of the calling function pointer matches the signature of the target function. Android enforces CFI through compiler instrumentation [9], whereas recent Intel systems implement it using a hardware-software co-design approach, namely FineIBT [220], which is discussed later in Section 3.2.2.

Limiting Escalation to Exploit Primitives. The Linux kernel implements significant hardening efforts to block earlier stages of exploitation. Specifically, it aims to prevent the escalation from triggering a vulnerability to obtaining usable exploitation primitives. These hardening measures include protections against stack tampering [295], which prevent stack overflows from corrupting return addresses. They also include the enforcement of safe memory operations via `CONFIG_HARDENED_USERCOPY`, which restricts user-to-kernel memory copies to valid memory regions. Additional defenses such as `CONFIG_LIST_HARDENED` and `CONFIG_DEBUG_LIST` deterministically block exploit techniques like unsafe unlinking [271]. This technique converts list pointer overwrites to arbitrary kernel writes and was used in multiple kernel exploits [256, 262, 285] including the in-the-wild BadBinder [270]. Other kernel defenses include diversification techniques such as KASLR [65], which randomizes the location of kernel memory sections and kernel objects with `CONFIG_RANDOMIZE_BASE` and `CONFIG_RANDOMIZE_MEMORY`. These location randomization schemes mitigate remote attacks on the kernel. Another diversification defense is `CONFIG_RANDSTRUCT_FULL` [143], which randomizes structure field order. Linux also relocates critical data to read-only memory sections such as kernel page tables [324] and arrays of function pointers [143]. Other efforts focus on hardening the kernel memory allocator. For example, freelist randomization [74] complicates OOB exploitation, while heap quarantine [244] aims to mitigate UAF attacks.

Heap Separation. To further reduce exploitation potential, Linux incorporates heap separation. At its core, it separates potentially vulnerable objects from security-critical ones by using distinct sets of slab caches, thereby limiting in-cache reuse attacks. This began with `KMALLOC_CGROUP` [191], which introduced separated cache sets based on security context. Follow-up advancements, such as `KMALLOC_SPLIT_VARSIZE` [109] and `SLAB_BUCKETS` [46], enabled finer-grained separation based whether objects allocated with user-controlled size, also called elastic objects [40]. `RANDOM_KMALLOC_CACHES` [51] introduced boot-time randomized slab caches based on the allocation site to prevent predictable cache sharing. The upcoming feature per-call-site slab cache sets (i.e., `CONFIG_SLAB_PER_SITE`) [50] will deterministically eliminate cache sharing between allocation sites. This approach is similar to the approach by grsecurity [182] and Apple [10]. While these efforts largely prevent in-cache reuse, cross-cache reuse techniques may still be viable, and potentially even more reliable due to reduced allocation noise. To address this, `SLAB_VIRTUAL` [109, 255] deterministically prevents cross-cache reuses, and is currently under review [255].

Minimizing the Kernel Attack Surface. Linux minimizes its kernel attack surface through a combination of isolation and access control mechanisms. The Linux Security Modules (LSM) framework enables the enforcement of security policies for mechanisms such as SELinux [208, 264], which implement mandatory access controls to constrain process capabilities. Additional reduction in exposure is achieved by approaches such as restricting unprivileged user namespaces [116, 155, 293], as these can otherwise grant untrusted code access to sensitive kernel interfaces. Seccomp is employed to filter system calls and, thereby, limit the kernel's externally visible interface. It is often used in conjunction with the BPF, which provides a flexible, programmable mechanism for defining system call filtering policies.

3.2.2. Academic Defenses and Mitigations

Academic defenses play a vital role in the ongoing arms race against threat actors. Although these techniques can provide robust protection, they often encounter significant delays before being integrated into mainline kernels, if adopted at all. This delay is usually due to challenges like integration complexity, performance concerns, and the need for agreement within the developer community. A notable example is kernel CFI, which limits control-flow hijacking attacks by restricting control transfers to a predefined CFG. Although CFI was first introduced in research in 2005 [1], kernel CFI has only been incorporated in a few downstream versions of the Linux kernel several years later. For instance, it has been available in the Android kernel since version 9, and became mandatory with Android 12 (with kernel version 5.10 or higher) as part of Generic Kernel Image (GKI) project 2.0 around 2021 [8, 9]. On the other hand, popular Linux distributions like Ubuntu, Debian, Arch, and Fedora include support for kernel CFI, but do not enable it by default. This shows how even well-researched and effective defenses can take many years to become widely used.

In the following, we categorize defenses based on mitigating control-flow hijacking, data-oriented attacks, and reducing the kernel attack surface. Finally, we examine non-monolithic kernel architectures.

Mitigating Control-Flow Hijacking Attacks. Because control-flow hijacking remains a prevalent class of attacks that grants arbitrary code execution, numerous academic defenses have been developed. At the user level,

CFI [1] has become the de facto defense. Code-Pointer Integrity (CPI) [69, 163] offers an alternative by deterministically protecting all code pointers, either through randomized location storage [163], segmentation [163], or hardware isolation primitives [316].

Protecting against control-flow hijacking in the kernel introduces additional challenges due to its low-level access, high complexity, and performance constraints [53, 198, 211]. To address this, a variety of kernel-level mitigations have been proposed. Early approaches such as kGuard [141] inserted checks mitigate specific attacks like ret2usr [141] (see Figure 2.7). KHide [78] defends against code reuse attacks by randomizing and practically hiding kernel code layouts. Meanwhile, XOM [241] prevents reading code sections, thereby obstructing gadget discovery, an essential for code reuse attacks.

KCoFI [53] adapts user-space CFI principles to the kernel by statically analyzing the CFG and instrumenting kernel code, although with a high performance overhead. KCoFI enforces CFI at the function-entry level, meaning that hijacks to valid function entry points remained possible. Fine-CFI [178] improved upon this by introducing path-sensitive control-flow tracking and providing function-signature granular CFI at a tolerable performance cost. kCFI [288] also provides function signature-based control-flow validation, which was later deployed in production systems such as Android [9]. Ge et al. [77] proposed a method for retrofitting kernel code based on conservative function pointer usage patterns and removal of CFI instructions, reducing the set of allowed targets compared to signature-based schemes. While these CFI techniques rely primarily on software-based approaches, modern CPUs have introduced hardware mechanisms that assist in securing control flow while having a low performance impact. On the ARM side, PATTER [323] protects function pointers and return addresses utilizing Pointer Authentication Code (PAC) [11]. PAC is a hardware feature that cryptographically signs pointers to ensure their integrity. Using PAC allows PATTER a function signature-based granularity with minimal performance overheads. Camouflage [56] extended PAC protections to operation table pointers which references an array of function pointers. PAL [280] offers a practical PAC deployment, using analysis for context-aware pointer verification to reducing the set of allowed control-flow targets. On x86 platforms, Intel presented Indirect Branch Tracking (IBT) [118] enforcing indirect calls to landing pads at function entries. FineIBT [220] leverages IBT with compiler-based validation for function signature-based enforcement with minimal overhead.

Contribution 3.4

HEK-CFI [198] is the first approach to protect all kernel backward edges. It also secures the thread state—CPU register contents saved during events such as system calls, exceptions, or interrupts—targeted in recent CFI bypasses [128, 156, 170, 210, 211]. Finally, it provides more fine-grained protection for forward edges than signature-based approaches, mitigating in-the-wild CFI bypasses [19, 126]. HEK-CFI achieves this by leveraging FineIBT for general forward-edge protection, and Intel CET shadow stacks for backward-edge, additional selected forward-edge, and thread state protection.

Mitigating Data-Oriented Attacks. Beyond control-flow hijacking, data-oriented attacks represent a significant class of exploits. These attacks compromise a system by corrupting non-control data [36, 110]. In user space, defenses such as Data-Flow Integrity (DFI) [30] and its hardware-assisted variant, Hardware DFI (HDFI) [266], have been proposed. They track and enforce legitimate data flows, thereby preventing unauthorized data manipulation. Apart from DFI, there are other isolation-based approaches, such as in-process isolation [103, 235, 258, 291]. These approaches enforce memory or privilege boundaries within a single process to prevent untrusted code from accessing sensitive data. Collectively, these user-space techniques aim to mitigate or reduce the impact of data-oriented attacks.

Nevertheless, these threats extend beyond user space: Xiao et al. [314] showed that data-oriented attacks also affect the kernel, underscoring the need for robust kernel defenses. Azab et al. [14] proposed a hypervisor-based scheme to protect the OS kernel via a secure monitor. This idea was later adopted as Real-time Kernel Protection (RKP) in Samsung kernels [59]. Similarly, Huawei also introduced a hypervisor-based solution called Huawei Kernel Integrity Protection (HKIP) [112]. These schemes safeguard sensitive data such as page tables and privilege-escalation-related objects. Song et al. [265] presented a method to identify security-critical kernel objects, which are potential targets of data-oriented attacks. They also introduced KENALI, a system that enforces kernel DFI to protect these objects. KENALI ensures that kernel data flows comply with predefined security policies by combining static analysis and runtime instrumentation, thereby mitigating memory-corruption-based privilege escalation attacks. Sergej et al. [248] proposed the xMP hypervisor-assisted scheme that enforces policy-driven access control over user and kernel memory. It enhances security by selectively restricting memory access based

on execution context. xMP achieves this by leveraging Intel's extended page table pointer switching to manage multiple memory views.

As CPUs began to offer dedicated hardware-based security features, many mitigation strategies shifted to architectural support, not relying solely on hypervisors. On ARM platforms, HAKC [209] addresses data manipulation threats by compartmentalizing kernel objects and allowing access only from predefined, authorized code regions. HAKC achieves compartmentalization by combining ARM PAC with ARM Memory Tagging Extension (MTE) [11]. MTE tags memory regions and pointers, ensuring that only pointers with matching tags can access their associated memory blocks. Also relying on these features, PeTAL [149] uses MTE to isolate sensitive kernel objects and PAC to protect their pointers. IUBIK [217] hardens the OS kernel by using ARM MTE to isolate attacker-controlled data in shadow memory, preventing it from corrupting sensitive kernel objects. On Intel platforms, Protection Keys for Userspace (PKU) and Protection Keys for Supervisor (PKS) [119] allow access control over memory page groups by tagging them with protection keys and enforcing permissions via a protection key register. These mechanisms allow faster and more fine-grained protection—such as selectively removing read or write access—without directly modifying page-table entries. Hence, several systems have leveraged PKU and PKS potentially combined with other mechanisms to enhance kernel protection. KDPM [164], KPRM [166], KDRM [165], and RKPM [167] combine protection keys with memory isolation and relocation techniques to defend against memory corruption. IskiOS [87] retrofits PKU to enforce execution-only memory and shadow stacks, using userspace keys to protect kernel memory rather than using the supervisor bit.

Contribution 3.5

DOPE [199] is the first to utilize Intel PKS for enforcing fine-grained access control over sensitive kernel data according to the principle of least privilege. As a result, DOPE protects multiple selected data objects from data-oriented attacks, while maintaining a reasonable performance overhead.

Although PKS only supports 16 keys, BULKHEAD [95] increases its usability by integrating it with address-space switching and in-kernel monitoring to implement effective compartmentalization.

Beyond page-level isolation, intra-kernel separation techniques [54, 225, 226, 228] aim to isolate entire kernel subsystems. For example, VirtuOS [228] isolates kernel services into domains using the Xen hypervisor.

Apart from these isolation-, restriction-, and compartmentalization-based schemes to limit data-oriented attacks, other approaches include allocator hardening, randomizing identifiers or layouts, hiding objects, and monitoring data. For kernel allocator hardening, SeaK [303] isolates and manages sensitive kernel objects using type-aware protection. SafeSlab [219] protects freed slab pages, while ISLAB [218] secures allocator metadata, leveraging Intel PKU and SMAP, respectively. In terms of randomization, SALADS [34] dynamically randomizes data structure layouts at runtime, making it difficult to correctly manipulate target fields. Wei et al. [305] proposed randomizing identifiers to break predictable relationships of object identification. For object hiding, PT-Rand [55] randomizes the location of page tables, preventing unauthorized modifications. Monitoring-based defenses include AKO [321] and PrivGuard [250], which track changes to sensitive data by maintaining duplicates and allowing only legitimate updates. PrivWatcher [35] also monitors, but stores the duplicates in read-only sections, preventing tampering.

Minimizing Kernel Attack Surface. Other defenses focus on reducing the kernel attack surface through specialization, debloating, or preventing access to vulnerable code. Specialization involves tailoring the kernel to the needs of a specific application. This minimizes exposure to unused functionality, as shown by prior work [2, 161, 162, 212]. In contrast, debloating [111, 344] aims to remove unused or legacy features from the kernel. Finally, some approaches prevent vulnerabilities from being triggered at runtime, e.g., PET [302], VULMET [320], and Zhang et al. [343].

Non-Monolithic Operating System Approaches. Alternative operating system architectures to traditional monolithic kernels include unikernels, microkernels, and library operating systems. A unikernel is a minimal, single-purpose OS image that links the application with the required OS components into a single binary. Examples include Jitsu [202], Unikraft [160], IncognitOS [63], and others [44, 279]. A microkernel runs only essential functions in kernel space, with other services in user space. These typically include components such as scheduling and Inter-Process Communication (IPC). An example of a microkernel is UnderBridge [93] that retrofits PKU-based isolation and introduces an efficient IPC mechanism. A library operating system provides traditional OS functionalities as modular libraries that applications can directly link against, as

demonstrated in systems such as Exokernel [66], FlexOS [175], and CubicleOS [257], among others [151, 245].

3.2.3. Mitigation Bypasses

Kernel CFI [9, 65, 220] is designed to prevent or limit control-flow hijacking by enforcing valid control-flow transfers. However, even state-of-the-art CFI implementations have notable limitations. Techniques such as control-flow bending [28] demonstrate that redirection can still occur within the legal bounds of the control-flow graph. Additionally, in-the-wild exploits have successfully targeted CFI-enabled systems, including Android [126] and iOS [19, 168] devices. This demonstrates that function signature-based CFI may not be sufficient for kernel software. WarpAttack [318] highlights how compiler optimizations can inadvertently weaken CFI protections. Page-Oriented Programming (POP) [100] demonstrates that a severe page remapping vulnerability (e.g., CVE-2013-2595) allows bypassing CFI schemes. Finally, Ret2Entry [211] reveals that FineIBT [220] can be bypassed via manipulation of spilled CPU registers [170, 210].

Beyond CFI bypasses, various other defense mechanisms in Android kernels have also been circumvented. These include Samsung KNOX [328], Samsung RKP [193, 229] (a component of KNOX), Huawei HKIP [112, 193], and SELinux [237]. Even hardware-level mitigations can be bypassed when considering transient execution and side-channel attacks. ARM features such as MTE [24, 84, 150] and PAC [253] have also been shown to be bypassable under certain conditions.

3.3. Analysis of Android Kernel Attack Surface

The Android kernel is a Google-maintained downstream variant of the Linux kernel and plays a critical role in the security of Android devices. Over the past years, it has become a frequent target for threat actors seeking to escalate privileges or gain persistent access. A growing number of kernel-level vulnerabilities have been exploited in the wild as part of complex exploit chains targeting mobile devices [32, 339].

Patching Gaps and Defense Challenges. In response to this evolving threat landscape, Google's Project Zero began systematically tracking

in-the-wild zero-day Android exploits in 2019, starting with the discovery of the BadBinder exploit [270]. BadBinder was used in the wild to escalate privileges in Android kernel compromises and has been linked to advanced spyware campaigns, such as those involving Pegasus [123]. Since then, Project Zero has published annual reports [32, 268, 272, 273, 274, 275, 276] aimed at increasing the cost and complexity of developing zero-day exploits. Their 2022 report [275] emphasized that modern exploit development requires significantly more time, resources, and expertise. This is likely due to improved platform defense and hardening schemes. However, Project Zero also noted a persistent challenge: delays between upstream patch releases and downstream updates often allow known n-day vulnerabilities to function as effective zero-days. In 2022 [275], for instance, vulnerabilities such as CVE-2022-38181 and CVE-2022-22706 were widely exploited months after patches had been issued.

Other investigations [20, 126, 171, 203] have reinforced that n-day vulnerabilities remain a favored target, due to the delay between upstream patch release and downstream device deployment. Threat actors often exploit this patch gap to weaponize known issues, rather than investing in the more demanding process of developing zero-day exploits.

Researchers have studied the patch gap for n-day vulnerabilities by analyzing how security patches are identified, propagated, and deployed across the Android ecosystem. Tian et al. [287] used commit metadata to identify Linux bug-fix patches, while Jiang et al. [133] and Zhang et al. [338] developed techniques to extract patch-specific signatures and verify their presence in kernel binaries. More recently, GraphSPD [300] improved patch detection using enriched code semantics and graph-based representations. Building on these efforts, Wu et al. [307] showed that most Android Security Bulletin issues originate from native code, while Farhang et al. [72] found that kernel CVEs experience the longest delays in appearing in vendor bulletins. Jones et al. [136] and Zhang et al. [343] quantified patch deployment delays, often spanning weeks or months for Android security updates. Acar et al. [3] and Lell et al. [176] further showed fragmentation in patch delivery and misreporting of device patch levels.

Complementing these detection studies, numerous studies have proposed strategies to mitigate risks stemming from delayed or incomplete patch integration in order to reduce n-day exploitability. Li et al. [180] analyzed Linux patch porting practices, identifying structural and organizational barriers that hinder timely patch propagation across kernel versions. Wang et al. [302] proposed patch-on-demand mechanisms for temporary

patch deployment, while Chen et al. [39] and Xu et al. [320] developed hot patching approaches to dynamically apply fixes. In a related line of work, Talebi et al. [283] addressed the issue of latent vulnerabilities by using system call instrumentation to prevent the execution of potentially harmful code paths. A persistent challenge in this area is that device vendors are often slow or reluctant to integrate available patches. To address this issue, Machiry et al. [200] proposed Spider, a framework aimed at accelerating and verifying correct patch propagation across Android ecosystems. Meanwhile, Android has introduced the APEX framework [7], which delivers critical system component updates independently of full firmware releases. This improves the timeliness and consistency of patches.

While patching is a reactive approach to fixing vulnerabilities, hardening is a proactive strategy aimed at limiting their exploitation. For example, Stone et al. [271] showed that enabling the `CONFIG_DEBUG_LIST` kernel configuration can break the unlink exploit primitive used by BadBinder [270] and related attacks [242, 256, 262]. However, similar to patch inclusion, adoption of kernel hardening features remains inconsistent.

Contribution 3.6

To evaluate this, Maar et al. [193] analyzed the integration of effective defenses that would prevent n-day exploitation in their Defects-in-Depth study. They found that prior to the GKI project [8], on average, devices from downstream vendors are 2.95 to 4.62 times less secure than they would be with proper adoption of available mitigations.

The introduction of Google's GKI project [8] represents a shift in Android kernel maintenance. Under GKI, Google centrally maintains the core Android kernel binary, decoupling it from device-specific customisations. Consequently, device kernels now incorporate upstream security fixes, patches, and mandatory kernel defenses and hardening schemes directly. This approach ensures that critical kernel updates are delivered to devices more quickly, thereby reducing device vendor dependency and narrowing the window of opportunity for threat actors. However, vulnerabilities in chipset-specific driver code fall outside the scope of GKI, as Google only maintains the core kernel and not the broad set of vendor-maintained drivers. As a result, many drivers remain subject to the same patching and adoption delays that characterized the pre-GKI ecosystem.

Kernel Driver Exploitation. Google Project Zero's annual reports on in-the-wild exploits targeting Android devices highlight kernel drivers

as a frequent attack target [268, 272, 274, 276]. This observation has been confirmed by other research groups [120, 121] as well as academic studies [16, 201, 339]. A key reason is that driver code tends to have lower quality compared to the core kernel [127, 130], making it more prone to unintentionally introducing vulnerabilities. Additionally, drivers are often chipset-specific, meaning that a single vulnerability can affect a wide range of devices using the same hardware platform. Hence, both zero-day and n-day exploits have increasingly focused on Android kernel drivers as an attack vector for compromising devices.

Multiple device drivers have been exploited by security researchers and threat actors, including drivers for the Graphics Processing Unit (GPU) [20, 67, 80, 81, 102, 213, 215, 216, 274, 275, 276, 310, 317], Digital Signal Processor (DSP) [67, 120, 130, 194], Neural Processing Unit (NPU) [214, 238, 239, 309, 342], JPEG coprocessor [127], USB [121], audio [126], and other subsystems [127, 134, 308]. Among these, GPU vulnerabilities stood out due to their broad impact and accessibility from untrusted security contexts. With only four major GPU vendors dominating the Android ecosystem, a single vulnerability can affect a wide range of devices. It is particularly concerning that such vulnerabilities can be triggered from untrusted contexts, such as regular apps that have been compromised. This eliminates the need to escalate to higher privilege levels, which were often previously required to reach kernel vulnerabilities [126, 134]. Consequently, GPU driver vulnerabilities were responsible for four out of five in-the-wild Android kernel compromises in 2023, with only one incident involving the core Linux kernel [276]. In response, Google has made GPU security a top priority [317], working with the Android Red Team and ARM.

Subsequent studies from Google Project Zero [127, 130], Amnesty International's Security Lab [120], and Maar et al. [194] demonstrated that, beyond GPU drivers, other drivers are also accessible from untrusted security contexts. Among these, the DSP drivers have emerged as a significant concern, having been actively exploited in the wild [120, 130].

Contribution 3.7

Maar et al. [194] further demonstrated a critical challenge in the patching lifecycle of driver vulnerabilities. Incorporating fixes often involves substantial delays [127, 194]. This delay gives threat actors opportunities to exploit known n-day vulnerabilities rather than investing in the discovery and exploitation of new zero-day ones.

In line with these findings, Google Project Zero has revised its disclosure policy [306]. While the previous policy emphasized a fixed deadline of typically 90 days for public disclosure, the updated framework prioritizes transparency around patch deployment delays across downstream ecosystems. Project Zero now explicitly includes the affected vendor or project, the impacted product, the report date, and the expected deadline. This policy change reflects a growing consensus in the community—empirically highlighted by Maar et al. [194]—that the key barrier to mitigating n-day exploitability lies not in the absence of technical fixes, but in the timeliness and completeness of their downstream integration.

In-The-Wild Full-Chain Mobile Exploits. Kernel-level vulnerabilities have been exploited in full-chain surveillance campaigns linked to spyware such as Pegasus [123], Predator [206], and NoviSpy [120]. Prior works [19, 20, 126, 168, 169, 203, 204, 231, 269, 270] have demonstrated how threat actors leverage kernel vulnerabilities to bypass sandboxing mechanisms and achieve persistent device compromise. Much of the existing research has focused on iOS, largely because Apple devices offer forensic artifacts and better tooling support. However, the core techniques, such as memory corruption across security boundaries, apply equally to Android, and a smaller but growing body of work has examined such cases in the Android ecosystem [20, 126, 130, 269, 270]. In contrast, prominent iOS case studies include operations such as Great iPwn [204], and exploits like FORCEDENTRY [169] and BLASTPASS [19, 168], among others [203].

One particularly noteworthy case is the in-depth analysis by Cearbhaill et al. [231]. They presented a forensic history of NSO Group's zero-click attacks in the wild. Zero-clicks are remote exploits that require no user interaction, making them especially stealthy and dangerous. Cearbhaill et al. reverse-engineered sophisticated exploit chains delivered via messaging services like iMessage and WhatsApp. These chains involved initial code execution, sandbox or process isolation escape, kernel-level privilege escalation, and bypassing code signature verification. Key insights from their forensic reconstruction include: First, zero-click exploits have been active since at least 2017, with cases uncovered from 2019 onward. Second, in-the-wild exploits can remain effective for anywhere from a few days to multiple months. Third, Google Project Zero's efforts have played a major role in disrupting NSO Group's operations. This study underscores the growing sophistication of mobile spyware and the vital role of public research in exposing and countering these threats.

4

Conclusion

This thesis addresses three under-explored and increasingly urgent aspects of kernel security to make modern computing platforms more resilient against kernel-level threats. Specifically, it enhances the *reliability of exploit techniques* through novel side channels, increases the *effectiveness of mitigations* against modern exploits, and reveals *shortcomings in the Android ecosystem*. While strengthening mitigations results in direct improvements in kernel security, increasing exploit reliability and exposing shortcomings have indirect but equally important effects. By demonstrating the feasibility of advanced attack techniques and identifying real-world weaknesses, this work pressures defenders to evolve, thereby driving defensive adaptation across both academic and industrial domains.

To enhance the *reliability of exploit techniques*, we presented three novel side channels: SLUBStick, KernelSnitch, and an advanced TLB side channel. SLUBStick is a generic timing side channel in the kernel's slab allocator, enabling reliable cross-cache reuse attacks. KernelSnitch is the first software-based side channel to leak fine-grained kernel heap object locations, exploiting timing differences in kernel data structures. Our advanced TLB side channel leaks multiple kernel memory regions, including heap objects, page tables, and stacks. These three side channels render previously unreliable or impractical exploit techniques feasible with high reliability and stability.

Collectively, these contributions demonstrate how specific design decisions in the Linux kernel can inadvertently introduce or amplify side-channel leakage. SLUBStick and KernelSnitch exploit the leakage that was introduced in the allocator behavior and the data structure implementation. The TLB side channel, on the other hand, demonstrates that some kernel defenses amplify side-channel leakage. Therefore, these leakage amplifying defenses improve security in one dimension but reduce it in another. By

uncovering new pathways for reliable kernel exploitation, these works motivate the development of defensive actions or countermeasures to address these, resulting in a more secure kernel.

To increase the *effectiveness of mitigations*, we introduced two defense mechanisms: HEK-CFI and DOPE. HEK-CFI defends against sophisticated control-flow hijacking attacks by combining fine-grained CFI with thread-state protection and Intel's shadow stacks. DOPE mitigates data-oriented attacks by leveraging domain-based memory protection with Intel PKS. Both defenses prevent previous bypass techniques and impose reasonable performance overheads, thereby advancing kernel security.

To reveal *shortcomings in the Android ecosystem*, we conducted two large-scale analyses on the Android kernels. The Defects-in-Depth analysis systematically examined the integration of upstream defenses in device vendor-supplied Android kernels against n-day exploitation flows. Our findings revealed significant disparities between the maximum and actual security achieved for these Android devices. The Doom of Device Drivers investigated the real-world attack surface accessible from untrusted execution contexts. We found that many Android devices remain vulnerable to publicly known n-day vulnerabilities. These studies provide actionable insights for improving the security of Android's kernel at scale.

Together, the contributions of this thesis advance the state of the art in kernel security, both offensively and defensively. They support raising the cost and complexity of kernel exploitation, making it increasingly difficult for even well-resourced actors to successfully compromise the kernel. This growing difficulty is reflected in a recent shift exemplified by the Paragon spyware campaign: rather than targeting the kernel to install spyware, threat actors focused on directly loading spyware into legitimate user-space apps and processes, which then unknowingly act as hosts. While this does not represent a definitive victory, it signals meaningful progress toward the central goal of kernel security research: to make compromise so costly and complex that it becomes infeasible. This thesis takes a significant step in that direction, though continued research will be required to ultimately achieve this objective.

Future Work. Looking forward, several promising directions emerge for future research. First, this thesis has taken a significant step toward understanding how side channels can leak critical kernel information to make kernel-level exploitation more reliable and stable. While we introduced three new side-channel attacks, this represents only a small part of the

potential kernel attack surface. Two of these channels, SLUBStick and KernelSnitch, are software-induced, caused by implementation details in the Linux kernel. The third attack, a TLB-based side channel, originates in hardware but is amplified by software-level decisions in kernel defenses and memory allocators. These findings suggest that other software subsystems may similarly expose exploitable side-channel leakage, yet remain largely unexplored. A comprehensive evaluation of the Linux kernel for such leakages remains an open and impactful research direction.

Second, while numerous academic defenses have been proposed to mitigate kernel exploitation, adoption in end-user devices remains limited. And when these defenses are deployed, they often appear only after considerable delays. This raises a crucial question: what are the practical, technical, or organizational barriers preventing timely and widespread deployment of these security mechanisms? Future work could focus on understanding these blockers—from performance overheads and engineering complexity to portability issues—and on developing effective, scalable defenses. Only by addressing the challenges of real-world adoption can we ensure that advances in kernel security translate into stronger protection for end users.

Third, threat actors continue to discover and target vulnerabilities, particularly in kernel subsystems. On Android devices, many of these vulnerabilities originate in vendor-maintained device drivers, which are typically developed by third parties and may not be maintained with security as a primary concern. As a result, many exploited kernel vulnerabilities remain relatively low-hanging fruit, including both zero-day and n-day issues identified in DSP, USB, or JPEG drivers. To reduce this persistent attack surface, future work could focus on developing more effective vulnerability detection techniques for kernel subsystems, particularly within the Android ecosystem, where inconsistent maintenance and delayed patching increase exposure to both known and unknown threats.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In: CCS. 2005 (pp. 43, 44).

[2] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In: USENIX Security. 2021 (p. 47).

[3] Abbas Acar, Güliz Seray Tuncay, Esteban Luques, Harun Oz, Ahmet Aris, and Selcuk Uluagac. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In: NDSS. 2024 (p. 49).

[4] National Security Agency. TEMPEST: A Signal Problem. 1972. URL: https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf (p. 35).

[5] 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 (p. 36).

[6] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In: USENIX Security. 2022 (p. 4).

[7] Android. APEX file format. 2022. URL: https://source.android.com/docs/core/ota/apex (p. 50).

[8] Android. Generic Kernel Image (GKI) project. 2024. URL: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image (pp. 12, 43, 50).

[9] Android. Kernel Control Flow Integrity. 2022. URL: https://source.android.com/docs/security/test/kcfi (pp. 10, 41, 43, 44, 48).

[10] Apple. Towards the next generation. 2022. URL: https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/ (p. 42).

[11] ARM. Arm Architecture Reference Manual for A-profile architecture. Feb. 2022 (pp. 44, 46).

[12] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation. In: USENIX Security. 2024 (p. 29).

[13] Awarau and pql. CVE-2022-29582 An io_uring vulnerability. 2022. URL: https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/ (pp. 8, 34).

[14] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In: CCS. 2014 (pp. 11, 45).

[15] Brandon Azad. A survey of recent iOS kernel exploits. 2020. URL: https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html (pp. 24–26).

[16] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In: USENIX ATC. 2019 (p. 51).

[17] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In: CCS. 2024 (p. 29).

[18] Tal Be'ery and Amichai Shulman. A Perfect CRIME? Only TIME Will Tell. In: Black Hat Europe. 2013 (p. 35).

[19] Ian Beer. Blasting Past Webp: An analysis of the NSO BLASTPASS iMessage exploit. 2025. URL: https://googleprojectzero.blogspot.com/2025/03/blasting-past-webp.html (pp. 5, 6, 10, 25, 32, 45, 48, 52).

[20] Ian Beer. Mind the Gap. 2022. URL: https://googleprojectzero.blogspot.com/2022/11/ (pp. 49, 51, 52).

[21] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In: AsiaCCS. 2011 (pp. 10, 31, 41).

[22] Novak Boskov, Naor Radami, Trishita Tiwari, and Ari Trachtenberg. Union Buster: A Cross-Container Covert-Channel Exploiting Union Mounting. In: International Symposium on Cyber Security, Cryptology, and Machine Learning. 2022 (p. 36).

[23] Daniel P Bovet and Marco Cesati. Understanding the Linux Kernel. O'Reilly Media, Inc., 2005 (pp. 20, 21).

[24] Mark Brand. MTE As Implemented, Part 2: Mitigation Case Studies. 2023. URL: https://googleprojectzero.blogspot.com/2023/08/mte-as-implemented-part-2-mitigation.html (p. 48).

[25] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: CCS. 2008 (pp. 10, 26, 31, 41).

[26] 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. 2019 (p. 36).

[27] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 8, 40).

[28] 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. 2015 (p. 48).

[29] Nicholas Carlini and David A. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In: USENIX Security Symposium. 2014 (pp. 10, 31, 41).

[30] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In: OSDI. 2006 (p. 45).

[31] Pumpkin Chang. How I use a novel approach to exploit a limited OOB on Ubuntu at Pwn2Own Vancouver 2024. 2024. URL: https://u1f383.github.io/slides/talks/2024_POC-How_I_use_a_novel_approach_to_exploit_a_limited_OOB_on_Ubuntu_at_Pwn2Own_Vancouver_2024.pdf (p. 31).

[32] Casey Charrier, James Sadowski, Clement Lecigne, and Vlad Stolyarov. Hello 0-Days, My Old Friend: A 2024 Zero-Day Exploitation Analysis. 2025. URL: https://cloud.google.com/blog/topics/threat-intelligence/2024-zero-day-trends (pp. 5, 7, 25, 26, 48, 49).

[33] Congcong Chen, Jinhua Cui, Gang Qu, and Jiliang Zhang. Write+Sync: Software Cache Write Covert Channels Exploiting Memory-disk Synchronization. In: TIFS (2024) (p. 39).

[34] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. A Practical Approach for Adaptive Data Structure Layout Randomization. In: ESORICS. 2015 (pp. 11, 47).

[35] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. PrivWatcher: Non-Bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In: AsiaCCS. 2017 (pp. 11, 47).

[36] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-Control-Data Attacks Are Realistic Threats. In: USENIX Security. 2005 (pp. 10, 45).

[37] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In: S&P. 2024 (p. 29).

[38] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In: USENIX Security. 2020 (p. 27).

[39] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android Kernel Live Patching. In: USENIX Security. 2017 (p. 50).

[40] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In: CCS. 2020 (pp. 29, 42).

[41] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In: CCS. 2019 (p. 27).

[42] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. In: USENIX WOOT. 2020 (p. 27).

[43] Mingi Cho and Wongi Lee. Utilizing Cross-CPU Allocation to Exploit Preempt-Disabled Linux Kernel. 2024. URL: https://www.hexacon.fr/slides/Cho_Lee-Utilizing_Cross-CPU_Allocation_to_Exploit_Preempt-Disabled_Linux_Kernel.pdf (p. 34).

[44] Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He, and Mohamed Kassi-Lahlou. Unikernel-based approach for software-defined security in cloud infrastructures. In: Network Operations and Management Symposium (NOMS). 2018 (p. 47).

[45] Kees Cook. Linux Kernel Hardening: Ten Years Deep. 2025. URL: https://outflux.net/slides/2025/lss/kspp-decade.pdf (p. 4).

[46] Kees Cook. mm/slab: Introduce kmem_buckets_create and family. 2024. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b32801d1255be1da62ea8134df3ed9f3331fba12 (pp. 8, 28, 42).

[47] Kees Cook. Security bug lifetime. 2016. URL: https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/ (p. 4).

[48] Jonathan Corbet. A slab allocator (removal) update. 2023. URL: https://lwn.net/Articles/932201/ (p. 23).

[49] Jonathan Corbet. Five-level page tables. 2017. URL: https://lwn.net/Articles/717293/ (pp. 17, 19).

[50] Jonathan Corbet. Per-call-site slab caches for heap-spraying protection. 2024. URL: https://lwn.net/Articles/986174/ (p. 42).

[51] Jonathan Corbet. Randomness for kmalloc(). 2023. URL: https://lwn.net/Articles/938637/ (pp. 28, 42).

[52] Jonathan Corbet. Supervisor mode access prevention. 2012. URL: https://lwn.net/Articles/517475/ (pp. 19, 31, 41).

[53] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In: S&P. 2014 (pp. 5, 10, 44).

[54] 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 (p. 47).

[55] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In: NDSS. 2017 (pp. 11, 47).

[56] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Camouflage: Hardware-assisted CFI for the ARM Linux kernel. In: DAC. 2020 (pp. 10, 44).

[57] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In: S&P. 2016 (p. 36).

[58] SSD Secure Disclosure. SSD Advisory - Linux Kernel nft_validate_register_store Integer Overflow Privilege Escalation. 2024. URL: https://ssd-disclosure.com/ssd-advisory-linux-kernel-nft_validate_register_store-integer-overflow-privilege-escalation/ (p. 32).

[59] Samsung Knox Documentation. Real-time Kernel Protection (RKP). 2023. URL: https://docs.samsungknox.com/admin/fundamentals/whitepaper/core-platform-security/real-time-kernel-protection/ (pp. 11, 33, 45).

[60] Florian Draschbacher and Lukas Maar. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024 (p. 14).

[61] Florian Draschbacher, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade ago. In: USENIX Security. 2025 (p. 14).

[62] David Drysdale. Anatomy of a system call, part 1. 2014. URL: https://lwn.net/Articles/604287/ (p. 20).

[63] Kha Dinh Duy, Jaeyoon Kim, Hajeong Lim, and Hojoon Lee. IncognitOS: A Practical Unikernel Design for Full-System Obfuscation in Confidential Virtual Machines. In: S&P. 2025 (p. 47).

[64] Jake Edge. Extending the use of RO and NX. 2011. URL: https://lwn.net/Articles/422487/ (p. 41).

[65] Jake Edge. Kernel address space layout randomization. 2013. URL: https://lwn.net/Articles/569635/ (pp. 8, 27, 41, 42, 48).

[66] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In: SOSP. 1995 (p. 48).

[67] Alisa Esage. Deep Dive: Qualcomm MSM Linux Kernel & ARM Mali GPU 0-day Exploit Attacks of October 2023. 2023. URL: https://zerodayengineering.com/insights/qualcomm-msm-arm-mali-0days.html (pp. 25, 51).

[68] ETenal. CVE-2022-27666: Exploit esp6 modules in Linux kernel. 2022. URL: https://etenal.me/archives/1825 (pp. 8, 34).

[69] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In: S&P. 2015 (p. 44).

[70] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: MICRO. 2016 (p. 36).

[71] 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 (p. 36).

[72] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An Empirical Study of Android Security Bulletins in Different Vendors. In: WWW. 2020 (p. 49).

[73] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In: CCS. 2000 (p. 36).

[74] Thomas Garnier. mm: SLAB freelist randomization. 2016. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c7ce4f60ac199fb3521c5fcd64da21cee801ec2b (p. 42).

[75] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks. In: FC. 2024 (pp. 13, 36).

[76] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: S&P. 2023 (pp. 13, 36).

[77] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In: Euro S&P. 2016 (pp. 10, 44).

[78] Jason Gionta, William Enck, and Per Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In: CNS. 2016 (p. 44).

[79] GlobalStats. Operating System Market Share Worldwide. 2025. URL: https://web.archive.org/web/20250428074102/https://gs.statcounter.com/os-market-share/ (p. 4).

[80] Guang Gong. TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices. 2020. URL: https://github.com/secmob/TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices/blob/master/us-20-Gong-TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices-wp.pdf (p. 51).

[81] Xiling Gong, Xuan Xing, and Eugene Rodionov. The Way to Android Root: Exploiting Your GPU On Smartphone. 2024. URL: https://i.blackhat.com/BH-US-24/Presentations/REVISED02-US24-Gong-The-Way-to-Android-Root-Wednesday.pdf (p. 51).

[82] Google. kernelCTF rules. 2023. URL: https://google.github.io/security-research/kernelctf/rules.html (pp. 8, 40).

[83] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In: CCS. 2022 (p. 29).

[84] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In: S&P. 2024 (p. 48).

[85] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security. 2018 (pp. 9, 40).

[86] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (p. 40).

[87] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In: RAID. 2021 (p. 46).

[88] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 9, 36, 39).

[89] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 6, 8, 9, 36, 38, 40).

[90] Daniel Gruss, Michael Schwarz, and Moritz Lipp. Meltdown: Basics, Details, Consequences. In: BlackHat USA. 2018 (p. 36).

[91] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (p. 35).

[92] Cheng Gu, Yicheng Zhang, and Nael Abu-Ghazaleh. I Know What You Sync: Covert and Side Channel Attacks on File Systems via syncfs. In: S&P. 2025 (pp. 9, 39).

[93] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-Kernel Isolation and Communication. In: USENIX ATC. 2020 (p. 47).

[94] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P. 2011 (p. 35).

[95] Yinggang Guo, Zicheng Wang, Weiheng Bai, Qingkai Zeng, and Kangjie Lu. BULKHEAD: Secure, Scalable, and Efficient Kernel Compartmentalization with PKS. In: NDSS. 2025 (p. 46).

[96] Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing. Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation. In: USENIX Security. 2024 (pp. 25, 29, 33, 34).

[97] h0mbre. Escaping the Google kCTF Container with a Data-Only Exploit. 2023. URL: https://h0mbre.github.io/kCTF_Data_Only_Exploit%5C# (pp. 8, 25, 34).

[98] h0mbre. Patch-Gapping the Google Container-Optimized OS for $0. 2025. URL: https://h0mbre.github.io/Patch_Gapping_Google_COS/ (p. 32).

[99] Hongli Han, Rong Jian, Xiaodong Wang, and Peng Zhou. Typhoon Mangkhut: One-click Remote Universal Root Formed with Two Vulnerabilities. 2021. URL: https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Typhoon-Mangkhut-One-Click-Remote-Universal-Root-Formed-With-Two-Vulnerabilities.pdf (p. 33).

[100] Seunghun Han, Seong-Joong Kim, Wook Shin, Byung Joon Kim, and Jae-Cheol Ryou. Page-Oriented Programming: Subverting Control-Flow Integrity of Commodity Operating System Kernels with Non-Writable Code Pages. In: USENIX Security. 2024 (p. 48).

[101] Tianshuo Han, Xiaorui Gong, and Jian Liu. CARDSHARK: Understanding and Stablizing Linux Kernel Concurrency Bugs Against the Odds. In: USENIX Security. 2024 (p. 29).

[102] Ben Hawkes. Attacking the Qualcomm Adreno GPU. 2020. URL: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html (pp. 25, 51).

[103] 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 (p. 45).

[104] Sean Heelan. How I used o3 to find CVE-2025-37899, a remote zeroday vulnerability in the Linux kernel's SMB implementation. 2025. URL: https://sean.heelan.io/2025/05/22/how-i-used-o3-to-find-cve-2025-37899-a-remote-zeroday-vulnerability-in-the-linux-kernels-smb-implementation/ (p. 29).

[105] Jann Horn. Exploiting race conditions on [ancient] Linux. 2019. URL: https://static.sched.com/hosted_files/lsseu2019/04/LSSEU2019%20-%20Exploiting%20race%20conditions%20on%20Linux.pdf (p. 30).

[106] Jann Horn. Firefox: ALSR leak and cross-frame oracle via pointer scrambling in Map/Set. 2016. URL: https://thejh.net/misc/firefox-cve-2016-9904-and-cve-2017-5378-bugreport (pp. 35, 36).

[107] Jann Horn. How a simple Linux kernel memory corruption bug can lead to complete system compromise. 2021. URL: https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html (pp. 8, 34).

[108] Jann Horn. Linux >=6.4: io_uring: page UAF via buffer ring mmap. 2023. URL: https://project-zero.issues.chromium.org/issues/42451653 (p. 29).

[109] Jann Horn. MITIGATION_README. 2022. URL: https://github.com/thejh/linux/blob/slub-virtual/MITIGATION_README (pp. 8, 42).

[110] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In: S&P. 2016 (pp. 10, 45).

[111] Zhenghao Hu, Sangho Lee, and Marcus Peinado. Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis. In: CCS. 2023 (p. 47).

[112] Huawei. EMUI 11.0 Security Technical White Paper. 2020. URL: https://consumer.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/emui_11.0_security_technical_white_paper_v1.0.pdf (pp. 33, 45, 48).

[113] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: USENIX Security Symposium. 2009 (pp. 10, 31, 41).

[114] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (pp. 6, 8, 9, 35, 36, 38, 40).

[115] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In: International Conference on Smart Card Research and Advanced Applications. 2013 (p. 35).

[116] Loutfi Ijlal. Restricted unprivileged user namespaces are coming to Ubuntu 23.10. 2023. URL: https://ubuntu.com/blog/ubuntu-23-10-restricted-unprivileged-user-namespaces (p. 43).

[117] Intel. 5-Level Paging and 5-Level EPT White Paper. 2017. URL: https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html (p. 18).

[118] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (p. 44).

[119] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers. May 2019 (p. 46).

[120] Amnesty International. "A Digital Prison": Surveillance and the suppression of civil society in Serbia. 2024. URL: https://securitylab.amnesty.org/latest/2024/12/a-digital-prison-surveillance-and-the-suppression-of-civil-society-in-serbia/ (pp. 4, 11, 12, 14, 25, 51, 52).

[121] Amnesty International. Cellebrite zero-day exploit used to target phone of Serbian student activist. 2025. URL: https://securitylab.amnesty.org/latest/2025/02/cellebrite-zero-day-exploit-used-to-target-phone-of-serbian-student-activist/ (pp. 11, 12, 14, 25, 51).

[122] Amnesty International. Europe: Paragon attacks highlight Europe's growing spyware crisis. 2025. URL: https://www.amnesty.org/en/latest/news/2025/03/europe-paragon-attacks-highlight-europes-growing-spyware-crisis/ (pp. 7, 11, 14, 25).

[123] Amnesty International. Forensic Methodology Report: How to catch NSO Group's Pegasus. 2021. URL: https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-how-to-catch-nso-groups-pegasus/ (pp. 4, 5, 11, 25, 38, 49, 52).

[124] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (p. 40).

[125] javierprtd. No CVE for this bug which has never been in the official kernel. 2023. URL: https://soez.github.io/posts/no-cve-for-this.-It-has-never-been-in-the-official-kernel/ (pp. 8, 34).

[126] Seth Jenkins. Analyzing a Modern In-the-wild Android Exploit. 2023. URL: https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html (pp. 5, 6, 10, 11, 25, 32, 45, 48, 49, 51, 52).

[127] Seth Jenkins. Driving forward in Android drivers. 2024. URL: https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html (pp. 25, 51).

[128] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack. 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html (pp. 5, 6, 8, 10, 25, 32, 40, 45).

[129] Seth Jenkins. Exploiting null-dereferences in the Linux kernel. 2023. URL: https://googleprojectzero.blogspot.com/2023/01/exploiting-null-dereferences-in-linux.html (p. 26).

[130] Seth Jenkins. The Qualcomm DSP Driver - Unexpectedly Excavating an Exploit. 2024. URL: https://googleprojectzero.blogspot.com/2024/12/qualcomm-dsp-driver-unexpectedly-excavating-exploit.html (pp. 5, 12, 25, 38, 51, 52).

[131] Qisheng Jiang and Chundong Wang. Sync+Sync: A Covert Channel Built on fsync with Storage. In: USENIX Security. 2024 (pp. 9, 36, 39).

[132] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In: S&P. 2023 (p. 30).

[133] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In: CCS. 2020 (p. 49).

[134] Xingyu Jin and Clement Lecigene. CVE-2024-44068: Samsung m2m1shot_scaler0 device driver page use-after-free in Android. 2024. URL: https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2024/CVE-2024-44068.html (p. 51).

[135] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel. 2021. URL: https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf (p. 31).

[136] Kailani R. Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G. Bardas. Deploying Android Security Updates: an Extensive Study Involving Manufacturers, Carriers, and End Users. In: CCS. 2020 (p. 49).

[137] Choo Yi Kai. A new method for container escape using file-based DirtyCred. 2023. URL: https://starlabs.sg/blog/2023/07-a-new-method-for-container-escape-using-file-based-dirtycred/ (p. 34).

[138] Max Kellermann. The Dirty Pipe Vulnerability. 2022. URL: https://dirtypipe.cm4all.com/ (pp. 27, 33).

[139] John Kelsey. Compression and Information Leakage of Plaintext. In: Fast Software Encryption. 2002 (p. 35).

[140] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security. 2014 (pp. 31, 41).

[141] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In: USENIX Security. 2012 (pp. 10, 30, 31, 41, 44).

[142] The Linux Kernel. 1. Introduction. 2025. URL: https://docs.kernel.org/process/1.Intro.html (p. 3).

[143] The Linux Kernel. Kernel Self-Protection. 2025. URL: https://docs.kernel.org/security/self-protection.html (pp. 41, 42).

[144] The Linux kernel. Complete virtual memory map with 4-level page tables. 2013. URL: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt (pp. 22, 23).

[145] The Linux kernel. Interrupt Handling. 2005. URL: https://static.lwn.net/images/pdf/LDD3/ch10.pdf (p. 20).

[146] The Linux kernel. Process Address Space. 2007. URL: https://kernel.org/doc/gorman/html/understand/understand007.html (pp. 17, 19).

[147] Google KernelCTF. Exploit. 2023. URL: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-3776_lts/docs/exploit.md (p. 32).

[148] Imran Khan. Linux SLUB Allocator Internals and Debugging. 2022. URL: https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-1 (pp. 22, 23).

[149] Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux. In: CCS. 2024 (p. 46).

[150] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. TikTag: Breaking ARM's Memory Tagging Extension with Speculative Execution. In: S&P. 2025 (p. 48).

[151] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv-Optimizing the Operating System for Virtual Machines. In: USENIX ATC. 2014 (p. 48).

[152] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 35).

[153] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 36).

[154] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (p. 35).

[155] Tamás Koczka. Learnings from kCTF VRP's 42 Linux kernel exploits submissions. 2023. URL: https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html (p. 43).

[156] Andrey Konovalov. Exploiting the Linux kernel via packet sockets. 2017. URL: https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html (pp. 5, 6, 10, 32, 45).

[157] Andrey Konovalov. External fuzzing of USB drivers with syzkaller. In: SAFACon. 2025 (p. 29).

[158] Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. Uncontained: Uncovering Container Confusion in the Linux Kernel. In: USENIX Security. 2023 (p. 29).

[159] 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 (pp. 8, 9, 40).

[160] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In: EuroSys. 2021 (p. 47).

[161] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In: NDSS. 2013 (p. 47).

[162] Anil Kurmus and Robby Zippel. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In: CCS. 2014 (p. 47).

[163] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In: OSDI. 2014 (p. 44).

[164] Hiroki Kuzuno and Toshihiro Yamauchi. KDPM: Kernel Data Protection Mechanism Using a Memory Protection Key. In: International Workshop on Security (2022) (pp. 11, 46).

[165] Hiroki Kuzuno and Toshihiro Yamauchi. Mitigation of privilege escalation attack using kernel data relocation mechanism. In: International Journal of Information Security (2024) (p. 46).

[166] Hiroki Kuzuno and Toshihiro Yamauchi. Prevention of Kernel Memory Corruption Using Kernel Page Restriction Mechanism. In: Journal of Information Processing (2022) (pp. 5, 11, 46).

[167] Hiroki Kuzuno and Toshihiro Yamauchi. RKPM: Restricted Kernel Page Mechanism to Mitigate Privilege Escalation Attacks. In: International Conference on Network and System Security. 2024 (p. 46).

[168] Citizen Lab. Blastpass NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild. 2023. URL: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/ (pp. 14, 25, 48, 52).

[169] Citizen Lab. ForcedEntry NSO Group iMessage Zero-Click Exploit Captured in the Wild. 2021. URL: https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/ (pp. 14, 25, 52).

[170] Michael Larabel. Linux's FineIBT Protections "Critically Flawed" Until Intel CPUs Appear With FRED. 2025. URL: https://www.phoronix.com/news/Linux-FineIBT-Critically-Flawed (pp. 10, 45, 48).

[171] Clement Lecigne. Spyware vendors use 0-days and n-days against popular platforms. 2023. URL: https://blog.google/threat-analysis-group/spyware-vendors-use-0-days-and-n-days-against-popular-platforms/ (p. 49).

[172] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In: USENIX Security. 2023 (pp. 6, 9, 36–39).

[173] Yoochan Lee, Byoungyoung Lee, and Chanwoo Min. Exploting Kernel Races Through Taming Thread Interleaving. 2020. URL: https://lifeasageek.github.io/papers/yoochan-exprace-bh.pdf (p. 30).

[174] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting Kernel Races through Raising Interrupts. In: USENIX Security. 2021 (p. 30).

[175] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. In: ASPLOS. 2022 (p. 48).

[176] Jakob Lell and Karsten Nohl. Mind the Gap - Uncovering the Android patch gap through binary-only patch analysis. 2018. URL: https://conference.hitb.org/hitbsecconf2018ams/materials/D2T1%20-%20Karsten%20Nohl%20&%20Jakob%20Lell%20-%20Uncovering%20the%20Android%20Patch%20Gap%20Through%20Binary-Only%20Patch%20Level%20Analysis.pdf (p. 49).

[177] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A Hybrid Alias Analysis and Its Application to Global Variable Protection in the Linux Kernel. In: USENIX Security. 2023 (p. 32).

[178] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. In: IEEE Transactions on Information Forensics and Security (2018) (pp. 10, 44).

[179] Tuo Li, Jia-Ju Bai, Gui-Dong Han, and Shi-Min Hu. LR-Miner: Static Race Detection in OS Kernels by Mining Locking Rules. In: USENIX Security. 2024 (p. 29).

[180] Xingyu Li, Zheng Zhang, Zhiyun Qian, Trent Jaeger, and Chengyu Song. An Investigation of Patch Porting Practices of the Linux Kernel Ecosystem. In: ACM Conference on Mining Software Repositories. 2024 (p. 49).

[181] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In: NDSS. 2024 (p. 27).

[182] Zhenpeng Lin. How AUTOSLAB Changes Kernel Self-Protection the Memory Unsafety Game. 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game (pp. 8, 34, 42).

[183] Zhenpeng Lin, Yueqi Chen, Xinyu Xing, and Kang Li. Your Trash Kernel Bug, My Precious 0-day. 2021. URL: https://www.blackhat.com/eu-21/briefings/schedule%5C#your-trash-kernel-bug-my-precious--day-24849 (p. 25).

[184] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In: CCS. 2022 (pp. 8, 25, 26, 29, 33, 34).

[185] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android. 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf (pp. 8, 29, 33, 34).

[186] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In: USENIX Security. 2022 (pp. 8, 9, 40).

[187] William Liu. CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers. 2022. URL: https://www.willsroot.io/2022/01/cve-2022-0185.html (pp. 7, 31, 40).

[188] William Liu. EntryBleed: Breaking KASLR under KPTI with Prefetch (CVE-2022-4543). 2022. URL: https://www.willsroot.io/2022/12/entrybleed.html (pp. 8, 9, 40).

[189] William Liu, Joseph Ravichandran, and Mengjia Yan. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In: HASP. 2023 (pp. 6, 8, 38, 40).

[190] Yong Liu, Jun Yao, and Xiaodong Wang. USMA: Share Kernel Code with Me. In: Black Hat Asia. 2022 (p. 33).

[191] Waiman Long. [PATCH v5 2/3] mm: memcg/slab: Create a new set of kmalloc-cg-<n> caches. 2021. URL: https://lore.kernel.org/lkml/20210512145107.6208-1-longman@redhat.com/ (pp. 8, 28, 42).

[192] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In: NDSS. 2017 (p. 27).

[193] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024 (pp. 5, 6, 12, 13, 27, 33, 48, 50).

[194] Lukas Maar, Florian Draschbacher, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard. The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities. In: USENIX Security. 2025 (pp. 5, 6, 12, 51, 52).

[195] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 5, 6, 8, 9, 25, 26, 29, 33, 34, 36, 38, 39).

[196] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025 (pp. 5, 6, 8–10, 33, 38, 40).

[197] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025 (pp. 5, 6, 8, 9, 36, 38–40).

[198] Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In: AsiaCCS. 2024 (pp. 5, 6, 10, 11, 32, 44, 45).

[199] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (pp. 5, 6, 11, 46).

[200] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In: S&P. 2020 (p. 50).

[201] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In: USENIX Security. 2017 (p. 51).

[202] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: just-in-time summoning of unikernels. In: NSDI. 2015 (p. 47).

[203] Bill Marczak, Adam Hulcoop, Etienne Maynier, Bahr Abdul Razzak, Masashi Crete-Nishihata, John Scott-Railton, and Ron Deibert. Missing Link Tibetan Groups Targeted with 1-Click Mobile Exploits. 2019. URL: `https://citizenlab.ca/2019/09/poison-carp-tibetan-groups-targeted-with-1-click-mobile-exploits/` (pp. 14, 25, 49, 52).

[204] Bill Marczak, John Scott-Railton, Noura Aljizawi, Siena Anstis, and Ron Deibert. The Great iPwn: Journalists Hacked with Suspected NSO Group iMessage 'Zero-Click' Exploit. 2020. URL: https://citizenlab.ca/2020/12/the-great-ipwn-journalists-hacked-with-suspected-nso-group-imessage-zero-click-exploit (pp. 5, 38, 52).

[205] Bill Marczak, John Scott-Railton, Kate Robertson, Astrid Perry, Rebekah Brown, Bahr Abdul Razzak, Siena Anstis, and Ron Deibert. Virtue or Vice? A First Look at Paragon's Proliferating Spyware Operations. 2025. URL: https://citizenlab.ca/2025/03/a-first-look-at-paragons-proliferating-spyware-operations/ (pp. 7, 14).

[206] Bill Marczak, John Scott-Railton, Daniel Roethlisberger, Bahr Abdul Razzak, Siena Anstis, and Ron Deibert. PREDATOR IN THE WIRES Ahmed Eltantawy Targeted with Predator Spyware After Announcing Presidential Ambitions. 2023. URL: https://citizenlab.ca/2023/09/predator-in-the-wires-ahmed-eltantawy-targeted-with-predator-spyware-after-announcing-presidential-ambitions/ (pp. 4, 11, 25, 52).

[207] 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 (p. 35).

[208] Rene Mayrhofer, Jeff Vander Stoep, Chad Brubaker, Dianne Hackborn, Bram Bonné, Güliz Seray Tuncay, Roger Piqueras Jover, and Michael Specter. The Android Platform Security Model (2023). In: arXiv:1904.05572 (2024) (p. 43).

[209] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In: NDSS. 2022 (p. 46).

[210] Jennifer Miller. [RFC] Circumventing FineIBT Via Entrypoints. 2025. URL: https://lore.kernel.org/linux-hardening/Z60NwR4w%2F28Z7XUa@ubun/ (pp. 10, 45, 48).

[211] Jennifer Miller, Manas Ghandat, Kyle Zeng, Hongkai Chen, Abdelouahab Benchikh, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. System Register Hijacking: Compromising Kernel Integrity By Turning System Registers Against the System. In: USENIX Security. 2025 (pp. 5, 6, 10, 26, 29, 32, 44, 45, 48).

[212] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In: OSDI. 2006 (p. 47).

[213] Man Yue Mo. Corrupting memory without memory corruption. 2022. URL: https://github.blog/security/vulnerability-research/corrupting-memory-without-memory-corruption/ (p. 51).

[214] Man Yue Mo. Fall of the machines: Exploiting the Qualcomm NPU (neural processing unit) kernel driver. 2021. URL: https://github.blog/security/vulnerability-research/fall-of-the-machines-exploiting-the-qualcomm-npu-neural-processing-unit-kernel-driver/ (pp. 25, 51).

[215] Man Yue Mo. Gaining kernel code execution on an MTE-enabled Pixel 8. 2024. URL: https://github.blog/security/vulnerability-research/gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/ (p. 51).

[216] Man Yue Mo. One day short of a full chain: Part 1 - Android Kernel arbitrary code execution. 2021. URL: https://securitylab.github.com/research/one_day_short_of_a_fullchain_android/ (p. 51).

[217] Marius Momeu, Alexander J Gaidis, Jasper vd Heidt, and Vasileios P Kemerlis. IUBIK: Isolating User Bytes in Commodity Operating System Kernels via Memory Tagging Extensions. In: S&P. 2025 (p. 46).

[218] Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and Vasileios P Kemerlis. ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels. In: AsiaCCS. 2024 (p. 47).

[219] Marius Momeu, Simon Schnückel, Kai Angnis, Michalis Polychronakis, and Vasileios P Kemerlis. Safeslab: Mitigating Use-After-Free Vulnerabilities via Memory Protection Keys. In: CCS. 2024 (p. 47).

[220] Joao Moreira. Kernel FineIBT Support. 2022. URL: https://lwn.net/Articles/891976/ (pp. 5, 10, 41, 44, 48).

[221] James Morse. 2015. URL: https://lwn.net/Articles/651614/ (pp. 31, 41).

[222] Slava Moskvin. Finding Bugs in Kernel. Part 1: Crashing a Vulnerable Driver with Syzkaller. 2024. URL: https://slavamoskvin.com/finding-bugs-in-kernel.-part-1-crashing-a-vulnerable-driver-with-syzkaller/ (p. 29).

[223] Slava Moskvin. Hunting Bugs in Linux Kernel With KASAN: How to Use it & What's the Benefit? 2024. URL: https://slavamoskvin.com/hunting-bugs-in-linux-kernel-with-kasan-how-to-use-it-whats-the-benefit/ (p. 29).

[224] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In: NDSS. 2022 (p. 29).

[225] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In: USENIX Security. 2019 (p. 47).

[226] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In: Virtual Execution Environments (VEE). 2020 (p. 47).

[227] Andy Nguyen. CVE-2021-22555: Turning \x00\x00 into 10000$. 2021. URL: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html (pp. 7, 25, 31).

[228] Ruslan Nikolaev and Godmar Back. VirtuOS: an operating system with kernel virtualization. In: SOSP. 2013 (p. 47).

[229] Vitaly Nikolenko. SELinux RKP misconfiguration on Samsung S20 devices. 2020. URL: https://duasynt.com/blog/samsung-s20-rkp-selinux-disable (p. 48).

[230] Lau Notselwyn. Flipping Pages: An analysis of a new Linux vulnerability in nf_tables and hardened exploitation techniques. 2024. URL: https://pwning.tech/nftables/ (pp. 5, 6, 25, 29, 33).

[231] Donncha O'Cearbhaill and Bill Marczak. Exploit archaeology a forensic history of in the wild NSO Group exploits. In: Virus Bulletin Conference. 2022 (pp. 5, 11, 14, 25, 52).

[232] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In: NDSS. 2025 (p. 13).

[233] Phil Oester. Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel1. 2016. URL: https://dirtycow.ninja/ (p. 29).

[234] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 35).

[235] 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 (p. 45).

[236] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Using Trāṭr to tame Adversarial Synchronization. In: USENIX Security. 2022 (pp. 9, 36, 39).

[237] Sean Pesce. Bypassing SELinux with init_module. 2023. URL: https://seanpesce.blogspot.com/2023/05/bypassing-selinux-with-initmodule.html (p. 48).

[238] Maxime Peterlin. Reversing and Exploiting Samsung's Neural Processing Unit. 2021. URL: https://blog.longterm.io/samsung_npu.html (p. 51).

[239] Maxime Peterlin. Reversing and Exploiting Samsung's NPU (Part 2). 2021. URL: https://blog.impalabs.com/2110_exploiting-samsung-npu.html (p. 51).

[240] Pedro Pinto. Unleashing a 0day-Pivoting Capabilities and Conquering the Linux Kernel. 2024. URL: https://www.figma.com/deck/GyXCgKKy6rMuY7NVZtInjY/Unleadhing-a-0day---0sec?node-id=19-610 (p. 34).

[241] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. kR^ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In: EuroSys. 2017 (p. 44).

[242] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html (pp. 7, 25, 29, 30, 33, 50).

[243] Alexander Popov. kernel-hardening-checker. 2018. URL: https://github.com/a13xp0p0v/kernel-hardening-checker (p. 41).

[244] Alexander Popov. Linux kernel heap quarantine versus use-after-free exploits. 2020. URL: https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html (p. 42).

[245] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In: ASPLOS. 2011 (p. 48).

[246] ProHoster. The Linux Kernel surpasses 40 Million lines of code: A historic nilestone in Open-Source software. 2025. URL: http://www.stackscale.com/blog/linux-kernel-surpasses-40-million-lines-code (p. 3).

[247] LLVM Project. Kernel AddressSanitizer (KASAN). 2020. URL: https://clang.llvm.org/docs/KernelAddressSanitizer.html (p. 29).

[248] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In: S&P. 2020 (pp. 5, 11, 45).

[249] Zhiyun Qian, Jiayi Hu, Jinmeng Zhou, Qi Tang, and Wenbo Shen. PageJack: A Powerful Exploit Technique With Page-Level UAF. 2024. URL: https://i.blackhat.com/BH-US-24/Presentations/US24-Qian-PageJack-A-Powerful-Exploit-Technique-With-Page-Level-UAF-Thursday.pdf (p. 29).

[250] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks. In: (2018) (pp. 11, 47).

[251] Jayden R. pipe_buffer arbitrary read write. 2022. URL: https://www.interruptlabs.co.uk/miscs/pipe-buffer (p. 33).

[252] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. GhostRace: Exploiting and Mitigating Speculative Race Conditions. In: USENIX Security. 2024 (p. 30).

[253] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In: ISCAA. 2022 (p. 48).

[254] Juliano Rizzo and Thai Duong. The CRIME attack. In: Ekoparty Security Conference. 2012 (p. 35).

[255] Matteo Rizzo and Jann Horn. Prevent cross-cache attacks in the SLUB allocator. 2023. URL: https://lore.kernel.org/linux-mm/202309151425.2BE59091@keescook/T/ (p. 42).

[256] Eloi Sanfelix. A bug collision tale. 2020. URL: https://labs.bluefrostsecurity.de/files/OffensiveCon2020_bug_collision_tale.pdf (pp. 25, 33, 42, 50).

[257] Vasily A. Sartakov, Lluis Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In: ASPLOS. 2021 (p. 48).

[258] 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. 2020 (p. 45).

[259] 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 (p. 32).

[260] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical Timing Side Channel Attacks on Memory Compression. In: S&P. 2023 (p. 35).

[261] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote Page Deduplication Attacks. In: NDSS. 2022 (p. 39).

[262] Blue Frost Security. Exploiting CVE-2020-0041 - Part 2: Escalating to root. 2020. URL: https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/ (pp. 33, 42, 50).

[263] Chaoqun Shen, Jiliang Zhang, and Gang Qu. MES-attacks: Software-controlled covert channels based on mutual exclusion and synchronization. In: DAC. 2023 (pp. 9, 36, 39).

[264] Stephen Smalley. Implementing SELinux as a Linux Security Module. 2001. URL: https://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf (p. 43).

[265] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In: NDSS. 2016 (pp. 5, 11, 45).

[266] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In: S&P. 2016 (p. 45).

[267] Maddie Stone. 0-days exploited by commercial surveillance vendor in Egypt. 2023. URL: https://blog.google/threat-analysis-group/0-days-exploited-by-commercial-surveillance-vendor-in-egypt/ (pp. 5, 25).

[268] Maddie Stone. 2022 0-day In-the-Wild Exploitation...so far. 2023. URL: https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html (pp. 7, 25, 26, 49, 51).

[269] Maddie Stone. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain. 2022. URL: https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html (pp. 33, 52).

[270] Maddie Stone. Bad Binder: Android In-The-Wild Exploit. 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html (pp. 4, 33, 42, 49, 50, 52).

[271] Maddie Stone. CONFIG_DEBUG_LIST=y. 2020. URL: https://twitter.com/maddiestone/status/1245834936629616640?lang=de (pp. 42, 50).

[272] Maddie Stone. Déjà vu-lnerability – A Year in Review of 0-days Exploited In-The-Wild in 2020. 2021. URL: https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html (pp. 7, 25, 49, 51).

[273] Maddie Stone. Detection Deficit: A Year in Review of 0-days Used In-The-Wild in 2019. 2020. URL: https://googleprojectzero.blogspot.com/2020/07/detection-deficit-year-in-review-of-0.html (pp. 4, 7, 14, 25, 26, 49).

[274] Maddie Stone. The More You Know, The More You Know You Don't Know. 2022. URL: https://googleprojectzero.blogspot.com/search?q=year (pp. 7, 14, 25, 26, 49, 51).

[275] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022. 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html (pp. 14, 49, 51).

[276] Maddie Stone, Jared Semrau, and James Sadowski. We're All in this Together: A Year in Review of Zero-Days Exploited In-the-Wild in 2023. 2024. URL: https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf (pp. 7, 11, 12, 25, 26, 49, 51).

[277] Yordan Stoychev. Abusing RCU callbacks with a Use-After-Free read to defeat KASLR. 2023. URL: https://anatomic.rip/abusing_rcu_callbacks_to_defeat_kaslr/ (p. 27).

[278] Yue Sun, Yan Kang, Chenggang Wu, Kangjie Lu, Jiming Wang, Xingwei Li, Yuhao Hu, Jikai Ren, Yuanming Lai, Mengyao Xie, et al. SyzParam: Introducing Runtime Parameters into Kernel Driver Fuzzing. In: arXiv:2501.10002 (2025) (p. 29).

[279] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-Unikernel Isolation with Intel Memory Protection Keys. In: Virtual Execution Environments (VEE). 2020 (p. 47).

[280] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In: USENIX Security. 2022 (pp. 10, 44).

[281] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In: EuroSys. 2011 (pp. 36, 39).

[282] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In: USENIX Security. 2018 (p. 29).

[283] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo Workarounds for Kernel Bugs. In: USENIX Security. 2021 (p. 50).

[284] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In: CCS. 2023 (p. 29).

[285] Zi Fan Tan, Gulshan Singh, and Eugene Rodionov. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938. 2024. URL: https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938 (pp. 33, 34, 42).

[286] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In: USENIX Security. 2022 (pp. 9, 40).

[287] Yuan Tian, Julia Lawall, and David Lo. Identifying Linux bug fixing patches. In: International Conference on Software Engineering (ICSE). 2012 (p. 49).

[288] Sami Tolvanen. Control Flow Integrity in the Android kernel. 2018. URL: https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html (pp. 10, 44).

[289] Martin Unterguggenberger, David Schrammel, Lukas Maar, Lukas Lamster, Vedad Hadzic, and Stefan Mangard. Cryptographic Least Privilege Enforcement for Scalable Memory Isolation. In: HOST. 2025 (p. 14).

[290] V4bel. Reviving the modprobe_path Technique: Overcoming search_binary_handler() Patch. 2025. URL: https://theori.io/blog/reviving-the-modprobe-path-technique-overcoming-search-binary-handler-patch (p. 32).

[291] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. In: USENIX Security. 2019 (p. 45).

[292] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In: CCS. 2015 (p. 36).

[293] Eduardo Vela. Making Linux Kernel Exploit Cooking Harder. 2022. URL: https://security.googleblog.com/2022/08/making-linux-kernel-exploit-cooking.html (pp. 7, 27, 43).

[294] Eduardo Vela and Space Meyer. More Bang for Your Bug. In: Linux Pumbers Conference. 2025 (p. 7).

[295] Arjan van de Ven. Add -fstack-protector support the the kernel. 2006. URL: https://lwn.net/Articles/193307/ (p. 42).

[296] Dmitry Vyukov. Syzkaller: kernel fuzzer. 2016. URL: https://github.com/google/syzkaller (p. 29).

[297] Alan Wang, Boru Chen, Yingchen Wang, Christopher Fletcher, Daniel Genkin, David Kohlbrenner, and Riccardo Paccagnella. Peek-a-Walk: Leaking Secrets via Page Walk Side Channels. In: S&P. 2025 (pp. 9, 40).

[298] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In: USENIX Security. 2021 (p. 29).

[299] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects. In: USENIX Security. 2023 (p. 29).

[300] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In: S&P. 2023 (p. 49).

[301] Yong Wang. Ret2page: The Art of Exploiting Use-Afer-Free Vulnerabilities in the Dedicated Cache. 2022. URL: https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf (p. 34).

[302] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel. In: USENIX Security. 2023 (pp. 47, 49).

[303] Zicheng Wang, Yicheng Guang, Yueqi Chen, Zhenpeng Lin, Michael Le, Dang K Le, Dan Williams, Xinyu Xing, Zhongshu Gu, and Hani Jamjoom. SeaK: Rethinking the Design of a Secure Allocator for OS Kernel. In: USENIX Security. 2024 (p. 47).

[304] Zihao Wang, Jiale Guan, XiaoFeng Wang, Wenhao Wang, Luyi Xing, and Fares Alharbi. The Danger of Minimum Exposures: Understanding Cross-App Information Leaks on iOS through Multi-Side-Channel Learning. In: CCS. 2023 (pp. 9, 39).

[305] Lifeng Wei, Yudan Zuo, Yan Ding, Pan Dong, Chenlin Huang, and Yuanming Gao. Security Identifier Randomization: A Method to Prevent Kernel Privilege-Escalation Attacks. In: Advanced Information Networking and Applications Workshops (AINAW). 2016 (pp. 11, 47).

[306] Tim Willis. Policy and Disclosure: 2025 Edition. 2025. URL: https://googleprojectzero.blogspot.com/2025/07/reporting-transparency.html (p. 52).

[307] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In: AsiaCCS. 2019 (p. 49).

[308] Le Wu, Xuen Li, and Tim Xia. ExplosION: The Hidden Mines in the Android ION Driver. 2022. URL: https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Wu-ExplosION-The-Hidden-Mines.pdf (p. 51).

[309] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way! 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf (pp. 34, 51).

[310] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html (pp. 5, 6, 8, 25, 29, 30, 33, 34, 51).

[311] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: USENIX Security. 2019 (pp. 32, 41).

[312] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In: USENIX Security. 2018 (p. 27).

[313] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. In: USENIX Security. 2023 (p. 29).

[314] Jidong Xiao, Hai Huang, and Haining Wang. Kernel data attack is a realistic security threat. In: International Conference on Security and Privacy in Communication Systems. 2015 (pp. 10, 45).

[315] Dongchen Xie, Dongnan He, Wei You, Jianjun Huang, Bin Liang, Shuitao Gan, and Wenchang Shi. BridgeRouter: Automated Capability Upgrading of Out-Of-Bounds Write Vulnerabilities to Arbitrary Memory Write Primitives in the Linux Kernel. In: S&P. 2025 (pp. 29, 33, 34).

[316] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In: CCS. 2022 (p. 44).

[317] Xuan Xing, Eugene Rodionov, Jon Bottarini, Adam Bacchus, Amit Chaudhary, Lyndon Fawcett, and Joseph Artgole. Google & Arm - Raising The Bar on GPU Security. 2024. URL: https://security.googleblog.com/2024/09/google-arm-raising-bar-on-gpu-security.html (pp. 12, 25, 51).

[318] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini†, Bing Mao, and Mathias Payer. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In: S&P. 2023 (p. 48).

[319] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (pp. 8, 34).

[320] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic Hot Patch Generation for Android Kernels. In: USENIX Security. 2020 (pp. 4, 47, 50).

[321] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes. In: International Journal of Information Security (2021) (pp. 11, 47).

[322] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In: ASPLOS. 2025 (p. 29).

[323] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels. In: arXiv:1912.10666 (2019) (pp. 10, 44).

[324] Jun Yao. arm64/mm: move {idmap_pg_dir,tramp_pg_dir,swapper_pg_dir} to .rodata section. 2018. URL: https://patchwork.kernel.org/project/linux-hardening/patch/20180620085755.20045-2-yaojun8558363@gmail.com/ (pp. 33, 42).

[325] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (p. 35).

[326] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features. 2018. URL: https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf (p. 33).

[327] Wang Yong. Make KSMA Great Again: The Art of Rooting Android devices by GPU MMU features. 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-WANG-The-Art-of-Rooting-Android-devices-by-GPU-MMU-features.pdf (p. 33).

[328] Dong-Hoon You. KNOX Kernel Mitigation Bypasses. 2019. URL: https://powerofcommunity.net/poc2019/x82.pdf (p. 48).

[329] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In: USENIX Security. 2023 (p. 29).

[330] Kyle Zeng. SSD Advisory - Linux Kernel taprio OOB. 2024. URL: https://ssd-disclosure.com/ssd-advisory-linux-kernel-taprio-oob/ (p. 32).

[331] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In: USENIX Security. 2022 (pp. 29, 37).

[332] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In: CCS. 2023 (pp. 25, 26, 29, 32, 41).

[333] Google Project Zero. Racing against the clock – hitting a tiny kernel race window. 2022. URL: https://googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html (p. 30).

[334] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In: FSE. 2020 (p. 29).

[335] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V Krishnamurthy, Trent Jaeger, et al. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In: NDSS. 2022 (p. 29).

[336] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels. In: CCS. 2021 (p. 29).

[337] Hang Zhang, Jangha Kim, Chuhong Yuan, Zhiyun Qian, and Taesoo Kim. Statically Discover Cross-Entry Use-After-Free Vulnerabilities in the Linux Kernel. In: NDSS. 2025 (p. 29).

[338] Hang Zhang and Zhiyun Qian. Precise and Accurate Patch Presence Test for Binaries. In: USENIX Security. 2018 (p. 49).

[339] Hang Zhang, Dongdong She, and Zhiyun Qian. Android Root and its Providers: A Double-Edged Sword. In: CCS. 2015 (pp. 48, 51).

[340] Jiliang Zhang, Chaoqun Shen, and Gang Qu. Mex+Sync: Software Covert Channels Exploiting Mutual Exclusion and Synchronization. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023) (pp. 9, 36, 39).

[341] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In: NDSS. 2018 (pp. 9, 39).

[342] Ye Zhang, Le Wu, Shupeng Gao, and Zheng Huang. Attacking NPUs of Multiple Platforms. 2023. URL: https://i.blackhat.com/EU-23/Presentations/EU-23-Zhang-Attacking-NPUs-of-Multiple-Platforms.pdf (pp. 25, 51).

[343] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An Investigation of the Android Kernel Patch Ecosystem. In: USENIX Security. 2021 (pp. 4, 47, 49).

[344] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: a reliable and practical approach to attack surface reduction of commodity OS kernels. In: RAID. 2018 (p. 47).

[345] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In: USENIX Security. 2022 (p. 29).

[346] Jinmeng Zhou, Jiayi Hu, Ziyue Pan, Jiaxun Zhu, Wenbo Shen, Guoren Li, and Zhiyun Qian. Beyond Control: Exploring Novel File System Objects for Data-Only Attacks on Linux Systems. In: arXiv:2401.17618 (2024) (p. 29).

[347] Weixi Zhu. Exploring Superpage Promotion Policies for Efficient Address Translation. MA thesis. 2019 (pp. 9, 40).

[348] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem. In: NDSS. 2024 (p. 30).

[349] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel. In: USENIX Security. 2022 (p. 29).

Part II.

Publications

List of Publications

During my PhD, I contributed to 12 publications in conference proceedings, 7 of which I was the first author. Of my first author papers, 5 were published at top-tier conferences.

Publications in this Thesis

1. **Lukas Maar**, Florian Draschbacher, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard. The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities. In: USENIX Security. 2025.

2. **Lukas Maar**, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025.

3. **Lukas Maar**, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025.

4. **Lukas Maar**, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024.

5. **Lukas Maar**, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024.

6. **Lukas Maar**, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In: AsiaCCS. 2024.

7. **Lukas Maar**, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023.

Other Contributions

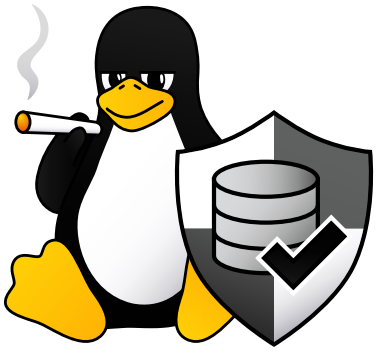
1. Florian Draschbacher, **Lukas Maar**, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade ago. In: USENIX Security. 2025.

2. Mathias Oberhuber, Martin Unterguggenberger, **Lukas Maar**, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In: NDSS. 2025.

3. Martin Unterguggenberger, David Schrammel, **Lukas Maar**, Lukas Lamster, Vedad Hadzic, and Stefan Mangard. Cryptographic Least Privilege Enforcement for Scalable Memory Isolation. In: HOST. 2025.

4. Florian Draschbacher and **Lukas Maar**. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024.

5. Stefan Gast, Jonas Juffinger, **Lukas Maar**, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks. In: FC. 2024.



5 DOPE: DOmain Protection Enforcement with PKS

Publication Data

Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023

Contributions

The author of this thesis is the main author of this work. The author's contributions are the proposal of *DOPE*, *proof-of-concept*, *case study* and *evaluation*, as well as most of the *systematic analysis* and written text.

DOPE: DOmain Protection Enforcement with PKS

Lukas Maar[1] Martin Schwarzl[2] Fabian Rauscher[1]
Daniel Gruss[1] Stefan Mangard[1]

[1] Graz University of Technology [2] Independent Researcher

Abstract

The number of Linux kernel vulnerabilities discovered has increased drastically over the past years. In the kernel, even simple memory safety vulnerabilities can have devastating consequences, e.g., compromising the entire system. Efforts to mitigate these vulnerabilities have so far focused mainly on control-flow hijacking attacks in the kernel. Yet, data-oriented attacks remain largely unmitigated in practice as existing mitigations are limited in providing robust security guarantees at reasonable performance overhead for multiple sensitive data objects.

In this paper, we present DOmain Protection Enforcement (DOPE), a novel kernel mitigation to protect against data-oriented attacks leveraging Intel's new hardware feature PKS. DOPE enforces domain protection, restricting memory access to sensitive data during kernel space execution based on the principle of least privilege. Hence, in case of an exploitable kernel bug, an attacker is prevented from using sensitive data for privilege escalation. We demonstrate DOPE's effectiveness and usefulness by implementing a proof-of-concept, protecting eight selected sensitive data objects. The proof-of-concept is realized as compiler-assisted and hardware-enforced kernel mitigation. It consists of less than 5 000 lines of code on the Linux kernel 5.19 and LLVM clang 15.0.1. Our evaluation on real hardware shows an average runtime overhead of 2.3 % for real-world user applications. Lastly, we systematically analyze 11 state-of-the-art kernel mitigations against data-oriented attacks and illustrate that DOPE is a significant improvement in terms of security with respect to performance.

1. Introduction

While memory safety is a well-researched topic, the challenge of finding complete, high-performance mitigations remains unsolved. Memory safety is especially relevant to the kernel since it is a valuable target for memory-corruption attacks. Memory-safety vulnerabilities in the kernel enable privilege escalation, rootkits, and confidential data leakage. Historically, Linux kernel exploits injected instruction sequences into kernel memory and hijacked the control flow to these sequences [28]. In 2009, Linux introduced an in-kernel `W^X` policy enforcing that kernel memory is never writable and executable, preventing code injection attacks [29].

However, attackers could still hijack the kernel control flow to escalate privileges [7, 9, 11, 38, 41]. In an attempt to mitigate control-flow hijacking attacks in the kernel, processor vendors introduced hardware-enforced restrictions for both user-space code execution (Intel SMEP [24] and ARM PXN [60]) and user-space data access (Intel SMAP [24] and ARM PAN [60]) in kernel mode. Subsequently, Linux added support for these features, making exploitation substantially more difficult. To further complicate control-flow hijacking attacks, Control-Flow Integrity (CFI) [1, 2, 20, 28, 58, 59] has been established as the state-of-the-art mitigation. CFI restricts the control flow to a set of transfers to ensure correct program execution, reducing the exploitation surface.

Besides hijacking the control flow, data-oriented attacks are a common attack class [15, 37]. Xiao et al. [82] showed that data-oriented attacks are not only a security concern for user applications but also the kernel. In the kernel, data-oriented attacks corrupt kernel data to indirectly change the control flow and escalate the attacker's privileges. Several mitigations [13, 14, 27, 45, 46, 55, 63, 64, 71, 81, 83] have been proposed to protect against data-oriented attacks. However, their practical deployment is hindered as they are limited in providing robust security guarantees at reasonable performance overhead for multiple sensitive data objects. For example, xMP [63] provides strong security benefits but has a performance overhead of up to 20 % for macro-benchmarks, potentially prohibited for commodity use cases. On the other hand, KDPM [45] has a low performance overhead but offers inadequate security guarantees as it does not mitigate forgery attacks.

In this paper, we present DOmain Protection Enforcement (DOPE), a novel kernel mitigation protecting against data-oriented attacks. By following

the principle of least privileges [66], DOPE restricts the memory accesses of threads during kernel space execution. To achieve this restriction, we move sensitive data objects into distinct security domains based on whether access to them could be used in privilege escalation exploits. Access to sensitive data objects is only granted if the thread has the associated domain's access permission, which DOPE enforces with Intel PKS [21]. DOPE only grants temporary access permission to domains in predefined, trusted code locations. In addition to protecting sensitive data objects, DOPE ensures the integrity of data pointers pointing to these sensitive data objects through ownership at runtime.

To demonstrate DOPE's effectiveness and usefulness, we implement a proof-of-concept and perform a case study. In our case study, we select eight sensitive data objects (i.e., credentials, virtual memory, virtual memory areas, inodes, page tables, filesystem mount, user-accessible pages, and stored registers) susceptible to being used for privilege escalation exploits and protect them with DOPE. The proof-of-concept implementation consists of less than 5 000 lines of code on a Linux kernel 5.19 and LLVM clang 15.0.1 [53]. Our implemented LLVM pass analyzes code and inserts domain switches and validation checks into the kernel's binary to ensure DOPE's functionality. We run our DOPE proof-of-concept on Ubuntu 22.04.1 LTS with a recent Intel Alder Lake processor. We evaluate the performance overhead of our proof-of-concept with micro-benchmarks from LMbench [57], showing an overhead of 32 %. In macro-benchmarks from Phoronix Test Suite [62] and SPEC CPU 2017 [25], we observed 2.3 % and 0.4 % overheads.

Lastly, we systematically analyze 11 state-of-the-art kernel mitigations against data-oriented attacks and point out blank spots in the mitigation landscape. We classify all analyzed mitigations against data-oriented attacks based on four techniques: Object monitoring [14, 64, 83], randomization [13, 27], compartmentalization [55, 81], and isolation [45, 46, 63, 71]. We further classify these mitigations according to their overheads and security guarantees. In this systematic analysis, we show that DOPE is a significant improvement in terms of security with respect to performance.

Contributions.

The main contributions of this work are:

1. **DOPE:** We present DOPE, a novel principled kernel mitigation for data-oriented attacks using Intel's new hardware feature PKS to enforce domain protection.
2. **Proof-of-concept:** We develop a proof-of-concept of DOPE[1] consisting of a Linux kernel extension and an LLVM pass illustrating the feasibility of our approach.
3. **Case study:** We perform a case study to demonstrate the effectiveness of DOPE in providing robust protection for eight selected sensitive data objects.
4. **Evaluation:** We evaluate DOPE's security and performance overhead, showing strong security guarantees with an overhead of 2.3 % for macro-benchmarks.
5. **Systematic analysis:** We examine 11 existing kernel mitigations for data-oriented attacks, identifying gaps in the mitigation landscape. We then show that DOPE provides superior security with respect to performance.

Outline.

In Section 2, we provide background and state-of-the-art countermeasures. In Section 3, we present our threat model. Section 4 presents our mitigation DOPE. While in Section 5, we describe our DOPE proof-of-concept, Section 6 performs a case study. In Section 7, we discuss DOPE's security and evaluate the proof-of-concept's performance overhead. Section 8 presents a systematic analysis. Lastly, we conclude our work in Section 9.

2. Background and State-of-the-Art

In this section, we provide background on Memory Protection Keys (MPK), as MPK plays an essential role in the design and implementation process. We then discuss existing user and kernel countermeasures against data-oriented attacks.

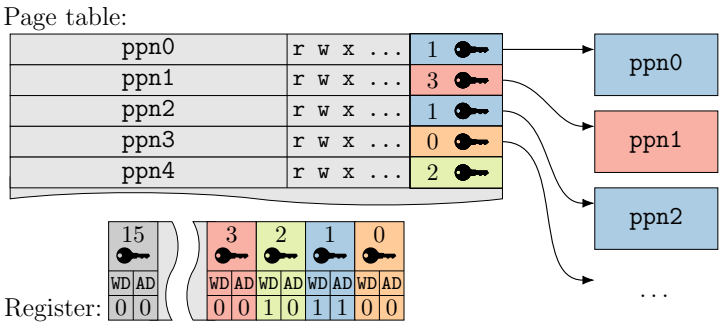


Figure 5.1: Working prinicple of MPK, where `AD` and `WD` stands for access and write disable, respectively. Pages tagged with key 0 are write- and access-permitted, while pages tagged with key 1 are write- and access-prohibited, and pages tagged with key 2 are only write-prohibited.

2.1. Memory Protection Keys

MPK are a hardware feature to enforce page-level permissions without modifying page-table entries [78]. The page permissions are enforced by tagging pages with a key and changing the permissions of these keys, stored in a dedicated hardware register. Intel features two variants of MPK, one for user- and one for kernel-space pages [39], namely Protection Keys for Userspace (PKU) [78] and Protection Keys for Supervisor (PKS) [21]. Both variants support 16 distinct keys, where a tagged page stores its applied key in its associated page-table entry. Each key comprises two permission bits: Access and write disable. Figure 5.1 shows MPK's working principle.

The dedicated register used by PKS is MSR `0x6E1` [39], called Protection Key Register for Supervisor (PKRS). Changing a key's permission is done by writing to PKRS. Since for permission changes no page-table walk or TLB flush is required [22], PKS is faster than changing permission bits directly in the page-table entry.

2.2. User Space Mitigations

Previous works [15, 37, 82] showed that data-oriented attacks can actively change the program's control flow by modifying sensitive data objects.

[1]Available `https://extgit.iaik.tugraz.at/sesys/dope`

Castro et al. [12] proposed Data-Flow Integrity (DFI), which tracks both read and write instructions. DFI enforces that data was not tampered with at runtime. Instead of validating a sensitive data object on every access, Akritidis et al. [3] proposed Write Integrity Test (WIT) that only performs checks on write instructions. WIT employs static analysis to assign a color to each write instruction and their associated sensitive data. Only write instructions with the correct color are permitted, practically preventing illegal write operations. Another approach to protecting sensitive data is data randomization [5, 6, 10], which encrypts sensitive data in memory with context-specific encryption keys.

To limit the exploitation surface of data-oriented attacks, prior works [35, 61, 69, 79] proposed various isolation mitigations. These mitigations use the Intel user-space implementation of MPK, PKU, to enforce isolation between different entities. Moreover, Intel's PKU is also commonly used in proposals to enforce isolation in unikernels [49, 72] and libOSes [48, 67].

2.3. Linux Kernel Mitigations

The explained user space techniques were adopted into the Linux kernel. Previous works provide DFI for sensitive data objects through object monitoring [14, 64, 83] or compartmentalization [55, 81]. Moreover, randomization-based approaches to the data location and layout were applied to the kernel [13, 27]. Other proposals [45, 46, 63, 71] isolate sensitive data objects from untrusted code.

As we later show in Section 8, existing mitigations have limitations in providing robust security guarantees, reasonable performance overhead, or protection for multiple sensitive data objects. To address these limitations, we propose our mitigation approach DOmain Enforcement Protection (DOPE). Our case study demonstrates that DOPE's proof-of-concept protects eight selected sensitive data objects with a reasonable runtime overhead. Compared to existing countermeasures, DOPE offers protection for multiple objects with strong security guarantees and lower performance overhead. This significant enhancement in security relative to performance underscores the superiority of our approach.

3. Threat Model

We assume an attacker can execute code in user space and has an arbitrary read-and-write primitive in the kernel accessible through the syscall interface without violating control-flow integrity. This aligns with the threat model of existing kernel defenses [13, 14, 27, 45, 46, 55, 63, 64, 71, 81, 83]. We do not specify the underlying flaw that leads to this primitive. We also assume modern kernel defense mechanisms such as Secure Boot, the `W^X` policy, KASLR [30], SMEP, and SMAP [24] are enabled, along with a CFI scheme [26, 31, 32, 58].

Attack vector. Since we assume that the enabled CFI scheme prevents control-flow hijacking attacks [7, 9, 11, 38], attackers focus on manipulating non-control data to elevate privileges. Such non-control data may include objects that contain information about privilege levels, like credential or inode objects, or information obtained from page permissions, such as page tables.

Out of scope. Although we acknowledge the presence of various types of attacks, such as side-channel [33, 51], microarchitectural [34, 44], and software fault injection [18, 70], as well as malicious hypervisors and operating systems, these are out of scope.

4. DOmain Protection Enforcement

DOmain Protection Enforcement (DOPE) is a principled mitigation for data-oriented attacks by protecting sensitive data objects from being exploited for privilege escalations. DOPE achieves this protection by adhering to the principle of least privilege [66]. During kernel space execution, DOPE restricts access to sensitive data for each thread based on their access permissions (cf. Section 4.1). To achieve this restriction, it places each sensitive data object into distinct security domains. Access to these objects is only granted if the thread has the associated domain permission (cf. Section 4.2), which DOPE enforces using Intel PKS (cf. Section 4.3). DOPE only grants temporary domain access permission in predefined, trusted code locations. We define code as trusted (cf. Section 4.5) if all accessed pointers are integrity ensured and, hence, trusted (cf. Section 4.4). Thus, DOPE effectively thwarts attackers from utilizing sensitive data for privilege escalation exploits in case of an exploitable bug.

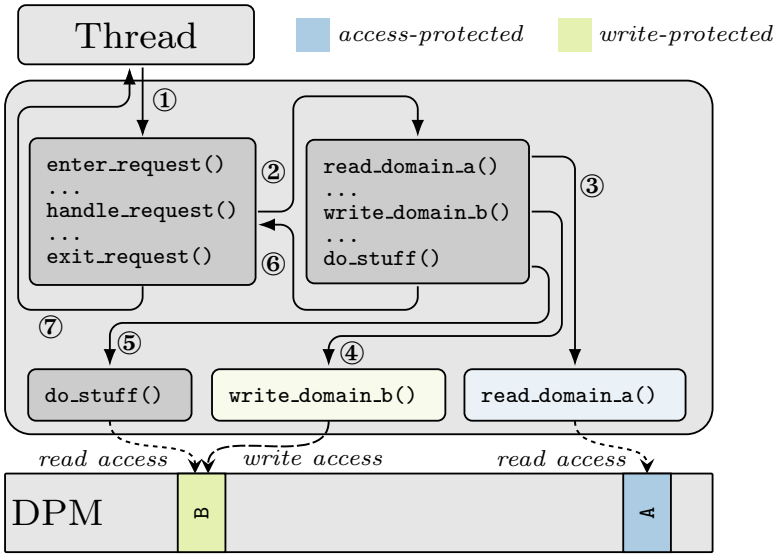


Figure 5.2: Access restriction of DOPE, where a thread invokes kernel space execution (①, ⑥), handles the request (②) and accesses sensitive data objects (③, ④, ⑤).

4.1. Restricted Access Permissions

DOPE provides a fine-grained permission setting for the security domains by prohibiting access or writing to the domain's data. On each kernel execution request entry, the thread's access permissions are set to restricted. The restricted access permissions list what domain is prohibited from being accessed or written to. They are defined before compile time by the system developers and enforced at runtime by DOPE. For example, the restricted access permissions in Figure 5.2 are: Prohibit access to domain `A` and prohibit write to domain `B`.

4.2. Sensitive Data Access

When a thread handles an execution request in kernel space, it is restricted from accessing sensitive data objects in accordance with its access permissions. A thread is only permitted to access such objects if it has the corresponding domain's permission, which DOPE temporarily grants by performing domain switches.

For instance, Figure 5.2 shows DOPE's access restriction. After a thread invokes kernel space execution ①, `enter_request` is called, which sets the thread's access permissions to restricted. While having restricted access, the thread handles the invoked request ②. In `read_domain_a` ③, the thread switches to domain `A` for every read access and, hence, acquires temporary read permission. The `write_domain_b` function ④ performs domain switches to gain domain `B` temporarily write permission. Since we define domain `B` as readable, the function `do_stuff` ⑤ is legally permitted to do so. Finally, `exit_request` ⑥/⑦ finishes the kernel execution request.

DOPE interprets any access from a thread that does not have domain permission as an exploit and terminates its execution, e.g., when the thread accesses domain `A` or writes to `B` within `do_stuff`.

DOPE supports two kernel execution requests: The first originates on thread creation and ends on termination. In between, the thread executes code in the kernel and user space, where switching between kernel and user space does not alter its permissions. The second originates on an asynchronous interrupt when the disrupted thread is in kernel space. Considering this execution request is crucial, otherwise, an attacker could perform the **elevated permission interrupt** attack, as we later describe in Section 7.1.

4.3. Enforcing Domain Protection with PKS

DOPE utilizes one PKS key per security domain and tags the pages of each security domain with their respective key. This tagging enables Intel PKS to enforce domain access restrictions. Any attempt to violate the access permissions of the tagged pages results in a fault, which DOPE interprets as an exploit attempt. Consequently, DOPE thwarts the exploit attempt, effectively mitigating the attack.

DOPE configures each `AD` and `WD` bit in the per-thread PKRS according to the access permission restrictions of the security domain, where `AD` and `WD` stand for access and write disable. In the example of Figure 5.2, DOPE sets `AD` and `WD` in the PKRS for domain key `A` to prohibit access to domain `A`. For domain key `B`, DOPE sets `WD` and resets `AD` to prevent writing to domain `B`. In order to enforce these restricted access permissions, the thread acquires this PKRS on each kernel execution request entry.

To perform a domain switch, DOPE alters the permission bit of the target domain key for the current thread. As a result, the current thread gains read or write access to the desired domain. The function `read_domain_a` in Figure 5.2 switches domains each time it reads data from domain `A`. It does so by modifying the `AD` bit in the PKRS of domain key `A`, where resetting the bit grants access permission and setting the bit removes the permission. Similarly, in `write_domain_b`, DOPE grants write permissions temporarily by resetting and setting the `WD` bit in the PKRS for domain key `B`.

Maintaining high performance is essential since the kernel is the lowest software abstraction layer. Previous research [61, 69] has demonstrated a direct correlation between the number of domain switches and the performance overhead. Therefore, DOPE aims to minimize domain switches by providing three variants of domain protection enforcement using Intel PKS. These variants differ in their level of spatial granularity, offering flexibility for system developers for their specific use cases. By minimizing domain switches, DOPE helps optimize system performance while providing strong security guarantees against data-oriented attacks.

4.3.1. Entire data object protection

The first protection variant of DOPE involves protecting an entire data object. In this approach, the page containing the data object is protected with PKS by tagging it with the associated domain key. Any data access that violates the domain's permissions is prohibited. This applies to both sensitive and non-sensitive members of the object. Consequently, to access a protected data object, the current thread must have the appropriate domain permissions and switch domains as needed, regardless of whether the accessed member is sensitive or non-sensitive. This approach is best suited when the data object consists mainly or entirely of sensitive data members. Since switching domains incurs a performance overhead, it may not be optimal if the object contains a mix of sensitive and non-sensitive members.

4.3.2. Shadow memory protection

DOPE introduces shadow memory protection for data objects containing a mix of sensitive and non-sensitive data members. In such cases, the

sensitive data members are duplicated on allocation, and the duplicated data are protected by tagging its page with the associated domain key. Consequently, the data object stores a pointer to the duplicated data. DOPE synchronizes sensitive and duplicated data every time sensitive data is written. With shadow memory protection, DOPE checks for each sensitive data read to see if the sensitive data matches the duplicated data. If sensitive and duplicated data differ, DOPE detects this as an exploitation attempt and terminates the thread's execution. To prevent an attacker from overwriting the pointer to the duplicated object, DOPE ensures the pointer's integrity, as we later show in Section 4.4.

This approach is particularly suitable for data objects with a mix of sensitive and non-sensitive data members. However, the runtime overhead associated with ensuring pointer integrity during read access to sensitive data members may make it less suitable for scenarios involving frequent access to such data members.

4.3.3. Sensitive data protection

Our proposed third variant provides a more efficient way to protect data objects containing both many non-sensitive data members and frequent access to sensitive data members. This variant enforces a specific data object layout, where all sensitive data members are placed on a PKS-protected page. On the other hand, non-sensitive data members are placed on an adjacent, non-protected page. As a result, accessing sensitive data members is protected by DOPE, while non-sensitive data members can be accessed without restrictions. This approach ensures that sensitive data is protected while minimizing the performance overhead. The only downside is that adapting the Linux kernel to accommodate this specific data layout requires effort.

The object layout of this variant is depicted in Figure 5.3, where the data object spans three contiguous pages. The sensitive data is safeguarded by Intel PKS and is only present on the middle page (grey). To prevent sensitive and non-sensitive data from ever coexisting on the same page, a dummy page is inserted between the end of non-sensitive data and the beginning of sensitive data, as well as between the end of sensitive data and the beginning of non-sensitive data. As a result, this approach entails a memory overhead of two pages per protected data object.

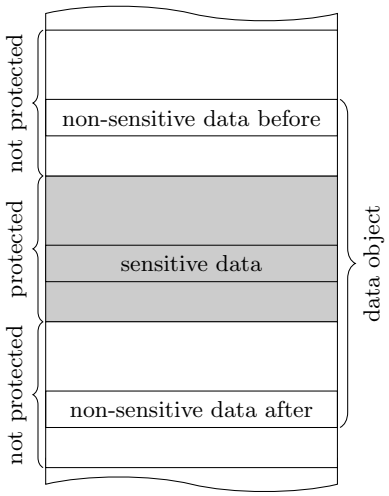


Figure 5.3: Data layout of sensitive data protection.

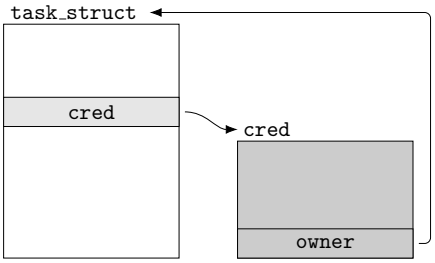


Figure 5.4: Ownership-based protection is employed to protect the sensitive pointer to `cred` within `task_struct`.

It is feasible to reduce the memory overhead by grouping sensitive data members from distinct data objects on the PKS-protected page while storing non-sensitive members on the adjacent page. Although implementing this memory layout necessitates even more engineering efforts to modify the Linux kernel, it presents a possible direction for future work.

4.4. Pointer Integrity through Ownership

DOPE ensures the integrity of data pointers to sensitive data objects by enforcing ownership, where access to the sensitive data object is restricted to its owner. The sensitive data object comprises the address of its owner object, and DOPE modifies the kernel to perform an owner validation before accessing a sensitive pointer. This validation checks whether the correct owner object is accessing the sensitive data object, thereby preserving the pointer's integrity.

DOPE utilizes two checks for owner validation: The first verifies if the sensitive object is tagged with the correct domain key. In contrast, the second compares if the owner address stored in the sensitive data object matches the owner object address. DOPE interprets a failure of either check as an exploitation attempt. Figure 5.4 exemplifies the credential struct

```
1 /* get ext4 inode */
2 struct inode *ext4_iget(){
3   struct ext4_inode *ei;
4   struct inode *inode;
5   ...
6   inode = dentry->inode;
7   inode->i_uid = i_uid;
8   inode->i_gid = i_gid;
9   ei->i_data[blk] = data;
10  ...
11  return inode;
12 }
```

Listing 5.1: `ext4_iget` reads `inode` from its owner `dentry`, and modifies its sensitive members `i_*id`.

```
1 /* get ext4 inode */
2 struct inode *ext4_iget(){
3   struct ext4_inode *ei;
4   struct inode *inode;
5   ...
6   inode = dentry->inode;
7 + owner_check(dentry, inode);
8 + enter_inode_wr();
9   inode->i_uid = i_uid;
10  inode->i_gid = i_gid;
11 + exit_inode_wr();
12  ei->i_data[blk] = data;
13  ...
14  return inode;
15 }
```

Listing 5.2: Modified and trusted `ext4_iget`.

`cred` with its owner `task_struct`. The validation procedure confirms that `cred` is tagged with the appropriate domain key and verifies whether `owner` and `task_struct` are identical. If the credential is suitably tagged, it is impossible to manipulate the `owner` member, thereby ensuring ownership.

In our ownership approach, we devise a reliable solution to handle aliasing, where a sensitive object is shared among multiple owners. To achieve this, we store the address of both the sensitive object and its owner object in a hashtable. On validation, DOPE checks whether the sensitive object with its owner is stored in the hashtable. This ensures that each owner is verified and eliminates any chances of ownership forgery of sensitive objects with multiple owners. Additionally, we tag the hashtable with a write-protected domain key, preventing any potential tampering attempts.

4.5. Trusted Code

DOPE imposes three constraints on trusted code. Firstly, only pointers whose integrity is ensured (cf. Section 4.4) are dereferenced in the trusted code area. Secondly, the memory pointed to by these pointers must be tagged with the domain to which the trusted code is temporarily granted access permission. Thirdly, objects are only permitted to be dereferenced with a fixed offset.

For instance, Listing 5.1 shows a code snippet where the `inode` is read from its owner `dentry`, and its sensitive data members `i_*id` are written. DOPE performs an owner validation to ensure the first constraint, as seen in Line 7 of the modified code snippet in Listing 5.2. With the trusted pointer, the trusted code part is between Lines 9 and 10, fulfilling all three constraints previously described. DOPE enters the write domain for inodes in Line 8 and exits in Line 11 to legally write to these sensitive data members.

Consider granularity. The granularity with which DOPE handles domain switches can be adjusted, influencing the performance overhead directly. However, this adjustment represents a trade-off: Finer granularity increases security at the expense of performance, while coarser settings can improve performance with potential security degradation. Determining the appropriate granularity, therefore, requires a thorough assessment of the balance between security and performance.

5. Implementation

In this section, we highlight our DOPE proof-of-concept implementation in the Linux kernel and LLVM pass [53]. We employed Linux version 5.19, the latest stable version when we started this work. At the time of writing, the Linux kernel did not have support for Intel PKS. Therefore, we implement secure PKS for the Linux kernel regarding data-oriented attacks. We then implement an LLVM pass to perform code analysis and automate function insertion. Finally, we implement our DOPE proof-of-concept.

Direct physical mapping and SLUB. Since DOPE requires permission setting at the page level granularity, we first break down the Direct Physical Mapping (DPM) from huge pages into 4 kB pages [23]. We then extend the buddy allocator to allocate pages tagged with a desired PKS key that defines the domain of the associated page. Moreover, we extend the functionality of the SLUB allocator to obtain an allocator that returns only data objects tagged with the desired domain. Our implementation extends the functionality of `kmem_cache` to provide the `kmem_dope_cache` object. Hence, each domain allocates tagged objects via `kmem_dope_cache`.

Sensitive state data. Since each thread can be in a different domain at a time, the PKRS has to be stored and restored on every context switch. The PKRS value of a currently not scheduled thread is stored in memory.

Storing the PKRS in an unprotected area poses the risk of an attack. With an arbitrary write primitive, an attacker could overwrite the stored PKRS and gain control over the hardware PKRS. To prevent this illegal control gain, we implement a secure way to store the PKRS. DOPE protects the stored PKRS with a write-prohibited security domain, where only limited and trusted locations are permitted to write to. We explain the sensitive state protection against attack scenarios in more detail in Section 7.1.

Thread creation. On thread creation, we allocate a sensitive state object and store a pointer to it in `thread_struct`. We then set its stored PKRS to restricted. Hence, the thread starts after it is first scheduled with restricted access permissions. DOPE protects the sensitive state objects with our ownership protection to prevent potential corruption attacks of pointers to sensitive state objects.

Domain switch. In DOPE, whenever a domain switch happens, it changes the permission bit of the target domain key for the current thread. This allows the current thread to gain read or write access to the desired domain temporarily. The permission bit is changed by writing to the MSR `0x6E1` with the `wrmsr` instruction.

Asynchronous interrupt. On asynchronous interrupt entry, DOPE stores the current PKRS in a stack-like structure within the write-protected sensitive state object, where the PKRS is read with instruction `rdmsr` from MSR `0x6E1`. Subsequently, the access permissions of the thread are set to restricted. On interrupt exit, DOPE restores the stored PKRS to obtain the interrupted permissions.

Instrumentation. We implement an LLVM pass that performs two crucial tasks: Code analysis and function insertion.

To protect sensitive data objects with either the entire data (cf. Section 4.3.1) or sensitive data (cf. Section 4.3.3) variant, our LLVM pass analyzes the code and identifies all read and write locations of the sensitive data. We then manually verify the analysis output to ensure domain switches are inserted at the appropriate locations. This combined approach of automatic analysis and manual verification provides the benefits of both methods. While automatic analysis helps identify difficult-to-find domain switch locations, manual verification ensures efficient domain switch placements and upholds constraints of trusted code. Additionally, the LLVM pass automatically inserts owner validations on each sensitive data object's read access from its owner object.

To estimate the manual effort required by our proof-of-concept, Listing 5.1 shows the function `__ext4_iget`, where `inode` is write-protected and `dentry` is its owner. The code analyzer outputs that between Lines 7 and 8 all sensitive data members (i.e., `i_*id`) are written. Hence, we manually insert an `enter_inode_wr` before Line 7 and `exit_inode_wr` after Line 8, where `*_inode_wr` enters and exits the inode write domain. Additionally, our LLVM pass automatically inserts an owner validation of the `inode` from its owner object `dentry`. Listing 5.2 shows the total instrumented code.

For sensitive data objects protected with shadow memory (cf. Section 4.3.2), our LLVM pass inserts synchronizations automatically for every write and validations for every read. The synchronization functions synchronize the sensitive and duplicated data. For validation, DOPE first performs an owner validation. DOPE then checks if the sensitive data has been modified illegally. If at least one of the two is true, DOPE detects the corruption attempt and terminates the thread's execution.

Trusted code. In Appendix 10, we provide measures to address any implementation issues while ensuring our code adheres to the trusted code constraints.

6. Case Study

We demonstrate the effectiveness and usefulness of DOPE by protecting eight sensitive data objects (cf. Section 6.1) from malicious accesses that violate restricted access permissions (cf. Section 6.2). For each sensitive data object, DOPE enforces domain protection with one of its three protection variants (cf. Section 6.3). Additionally, DOPE ensures the integrity of pointers to sensitive data objects by enforcing ownership (cf. Section 6.4) at runtime. Lastly, we discuss the manual efforts of our case study (cf. Section 6.5).

6.1. Sensitive Data Objects

All restriction-based mitigations against data-oriented attacks face a fundamental question of which data objects to protect. The more sensitive data objects a mitigation adequately protects, the higher the security and performance overhead. As the number of protected objects increases, the

security benefit of additional objects decreases as exploiting the system becomes substantially more difficult. How to set a trade-off between security and performance depends on the use case. For our case study, we demonstrate that DOPE can be deployed to protect eight sensitive data objects with a reasonable performance overhead. More precisely, we protect user-accessible pages, stored registers, credentials, inodes, page tables, virtual memory areas, virtual memory, and filesystem mount, with the former two discussed in detail and the remaining six in Appendix 11. Crucially, our approach protects more objects with strong security guarantees while imposing a lower runtime overhead than existing countermeasures [13, 14, 27, 45, 46, 55, 63, 64, 83].

User-accessible pages. To our knowledge, we are the first to consider user-accessible pages via the DPM in their threat model for data-oriented attacks. User-accessible pages are either mapped in any user space or read from the disk and remain in kernel space. These pages may either be from the current or another process's user space, including high-privilege processes. If left unprotected, attackers can perform DPM-FPATCH (cf. Appendix 12).

Stored registers. An attacker can convert an arbitrary read-and-write to a register manipulation primitive. To achieve this, the attacker enforces a victim thread to preempt, causing the registers to be stored in memory [73]. Consequently, the attacker corrupts this memory location. When the thread resumes, the registers are restored from the corrupted memory, granting the attacker control over them. Suppose the preemption happens when the victim thread has access permission to a domain, the attacker can manipulate the victim's registers to perform an access, bypassing the applied mitigation if not protected. However, unlike existing schemes [13, 14, 27, 45, 46, 55, 63, 64, 83], DOPE provides protection for stored registers during preemption, effectively preventing such attacks.

6.2. Restricted access permissions

DOPE provides a fine-grained permission setting that applies to our sensitive data objects, which comprise nine objects, including sensitive state. Our case study works with three security domains:

Default: Permits read and write to data[2].

[2]Crucially to note, the tagged PKS key does not override permission bits, such as writable bit.

Table 5.1: Applied protection variant for our sensitive data objects.

Variant	Sensitive data objects								
	User-accessible pages	Credentials	Inodes	Page tables	Virtual memory areas	Virtual memory	Filesystem mount	Stored registers	Sensitive state
4.3.1 Entire	●	●	○	●	○	○	○	●	●
4.3.2 Shadow	○	○	○	○	●	●	●	○	○
4.3.3 Sensitive	○	○	●	○	○	○	○	○	○

● Applied ○ Not applied

Write-protected: Permits read and prohibits write to data.
Access-protected: Prohibits read and write to data.

We place the stored register to the *access-protected* domain because an attacker could otherwise access confidential data, potentially bypassing DOPE. Moreover, we set the user-accessible page to be *access-protected* because an attacker could otherwise leak confidential data from a high-privilege user process. The other sensitive data objects are set to *write-protected* because: First, the data can be legally read via syscalls, i.e., credentials, inodes, and filesystem mount. Second, read access to these objects cannot be exploited for privilege escalation, i.e., page tables, `vm_page_prot` (virtual memory areas), `pgd` (virtual memory), and sensitive state.

While it may seem that assigning each sensitive data object to an individual security domain would increase security, our goal is not to isolate domains from each other but to isolate them from attackers with strong capabilities. Hence, we group sensitive data objects with the same access permissions to one security domain. This approach ensures that highly capable attackers cannot gain access to sensitive data that violates our restricted access permissions.

6.3. Enforcement variant

In this section, we demonstrate the feasibility and usefulness of the protection variants provided by DOPE, each of which is well-suited for specific sensitive data objects. Table 5.1 presents the applied variant for each

Table 5.2: Owner of each sensitive data objects.

Owner	Sensitive data objects								
	User-accessible pages	Credentials	Inodes	Page tables	Virtual memory areas	Virtual memory	Filesystem mount	Stored registers	Sensitive state
`task_struct`	-	●	○	-	○	○	○	●	●
`dentry`	-	○	●	-	○	○	○	○	○
`vfsmount`	-	○	○	-	○	○	●	○	○
`vma_struct`	-	○	○	-	●	○	○	○	○
`mm_struct`	-	○	○	-	○	●	○	○	○

● Owner ○ Not owner

object. In our case study, we protect credentials, user-accessible pages, page tables, stored registers, and sensitive states with entire data object protection (cf. Section 4.3.1) as they comprise mostly or entirely of sensitive data members. For virtual memory areas, virtual memory, and filesystem mount, we utilize shadow memory protection (cf. Section 4.3.2) as these objects contain a combination of sensitive and non-sensitive data. In contrast, since inodes contain many non-sensitive data members, such as locks and modification time, and their sensitive data is accessed frequently, sensitive data protection (cf. Section 4.3.3) is more suitable.

6.4. Ownership

DOPE ensures ownership of sensitive data objects to prevent forgery attacks. When accessing a data pointer to a sensitive data object, DOPE performs an owner validation to determine if the correct owner is accessing the data object. DOPE stores the address of its owner object in the sensitive data object, as shown in Table 5.2.

We identify seven sensitive data objects susceptible to forgery attacks. For the shadow data (virtual memory areas, virtual memory, and filesystem mount), sensitive state objects, and stored registers, DOPE stores the owner's address to bind the object to the owner uniquely. Neither page tables nor user-accessible pages can be forged as the higher-level page table is protected with the *write-protected* domain. In case an attacker tries to manipulate page-table entries, DOPE detects and prevents the tampering

attempt. The highest page-table level, `pgd` (i.e., virtual memory), can also not be forged, as it is also protected with the *write-protected* domain.

Both credentials and inodes may have multiple owners. In the case of credentials, they are shared among threads within a process. To ensure ownership, DOPE stores the `task_struct` address of the initial thread within the credential. For additional `task_struct`s, DOPE stores their address combined with the credentials in a dedicated hashtable. In the case of inodes, the dentry is designated as the owner since it links the user accessibility file to its inode [74]. Although inodes are not typically shared between different dentries, hardlinks result in multiple dentries sharing the same inode. Hence, the inode stores the dentry's address as its owner, and in case of a hardlink, both the dentry and inode are stored in a hashtable. Both hashtables, for credentials and inodes, are *write-protected*.

6.5. Instrumentation of our Case Study

Manual effort. To address the manual efforts, we followed three steps. Firstly, we modified the sensitive data objects by adjusting their layout to match the protection variant and adding a member variable to store the owner object's address. We also separated the `rcu_head` member from the credential by dynamically allocating the `rcu_head` and storing a pointer within the credential. Secondly, we replaced allocation and freeing of sensitive data objects with `kmem_dope_cache`. Thirdly, as explained in Section 5, we inserted domain switches based on the LLVM pass's code analyzer output. To ensure optimal performance without undermining security, precisely during multiple sensitive data accesses, we grouped these accesses. We then inserted a single domain switch both at the start and end of these grouped accesses.

False negatives. With proper domain switches in place, access to sensitive data is granted in trusted code locations. In cases where we would have missed inserting a domain switch (false-negative), DOPE would mistakenly identify the access as an exploitation attempt, as the current thread does not have access permissions. However, we did not encounter any such occurrences during our evaluation (cf. Section 7.2) and testing with LTP [36].

7. Evaluation

We assess DOPE's security before evaluating our proof-of-concept on real hardware with various benchmarks [25, 57, 62].

7.1. Security Discussion of DOPE

This section demonstrates the robustness of DOPE even in the presence of a powerful attacker, as described in Section 3.

Sensitive data objects. If access to sensitive data objects violates the restricted access permissions, Intel PKS triggers a fault, which DOPE interprets as an exploitation attempt and terminates the thread's execution. Hence, it is not possible to access sensitive data without proper access permissions, i.e., in trusted code.

Ownership. An attacker may aim to manipulate pointers to sensitive data objects protected with ownership. If the attacker tampers with the sensitive data pointer and points it to memory tagged with the wrong or no domain, DOPE detects it on owner validation. If the memory is tagged with the correct domain, the attacker cannot manipulate the owner member because they do not have write permissions to the domain-protected data. Additionally, if the attacker overwrites the pointer with an existing high- or low-privilege object correctly tagged, DOPE detects the manipulation on owner comparison during validation. Therefore, it is not possible to forge a sensitive data object that passes owner validation.

Asynchronous domain-protected data access. Compared to previous works that protect memory by mapping it as read-only [14, 71], PKS sets permissions on a logical core granularity. Even if a thread currently has access permission to a domain, another thread from another logical core does not. Therefore, asynchronous access to sensitive data is not possible with Intel PKS by design.

Elevated permission interrupt. Due to the preemptive nature of the Linux kernel, an asynchronous interrupt may occur while a thread has access permission to a domain. During the interrupt, the disrupted thread accesses data via untrusted pointers. An attacker could carefully craft the untrusted pointers to force the thread to perform domain data access. However, DOPE protects against this attack scenario by storing the access permissions (PKRS) and setting it to restricted on interrupt entry.

On interrupt exit, DOPE restores the access permissions and continues execution.

Sensitive state protection. DOPE protects all sensitive state data, comprising the stored PKRS values, by placing it in the *write-protected* domain. Pointers to the objects are integrity-ensured using our ownership approach. Only four routines are permitted to write to this data: Thread creation, context switch, and asynchronous interrupt entry and exit. During these operations, the thread validates ownership of the sensitive state object and temporarily grants write access for storing the PKRS to the object. This robust protection ensures that attackers cannot tamper with the stored PKRS.

`kmem_dope_cache` **manipulation.** An attacker may manipulate the state of the `kmem_dope_cache` object in order to return an attacker-controlled address. Therefore, the attacker can force the `kmem_dope_cache` object to return an object that is not tagged. DOPE protects against this attack by checking whether the returned address is tagged with the correct domain key after the allocation. DOPE interprets a domain key mismatch as an exploit attempt.

Arbitrary use-after-free. An arbitrary write primitive can be converted to an arbitrary use-after-free primitive by tampering with unprotected memory to obtain `kfree(sens_obj_in_use)`. On the next allocation, `sens_obj_in_use` may be returned, allowing it to be overwritten with either low- or high-privileged data. One attack scenario is overwriting credentials owned by a low-privilege thread with high-privilege credentials. Another scenario is to overwrite an inode owned by a high-privilege file with low-privilege metadata. If left unprotected, both scenarios would lead to privilege escalation. Notably, this attack closely resembles DirtyCred [50].

However, during allocation, our `kmem_dope_cache` overwrites the owner member of the sensitive data object with the new owner. When the actual owner first accesses the object, DOPE performs an owner validation, which fails since the owner was overwritten during allocation. DOPE interprets this attack scenario as an exploitation attempt and terminates the thread's execution.

Multi-ownership. DOPE employs a two-step validation before adding the sensitive data object and new owner to the hashtable. Firstly, it validates the old owner and sensitive data object; secondly, it validates that the new owner is not already present in the hashtable. If either of these validations fails, DOPE terminates execution. As a result, an attacker can

neither forge ownership nor perform **arbitrary use-after-free** with the new owner.

Physical memory. Attackers may tamper with the DPM to potentially bypass DOPE's protection of sensitive data objects. However, since all sensitive data objects and their permissions are directly accessed and set on the DPM, it is not possible to use the DPM for bypassing. Additionally, it is not possible to corrupt the permissions of sensitive data objects as the page tables containing the tagged domain key are protected by the *write-protected* domain.

Pointer-to-pointer attack. DOPE provides a robust mechanism for ensuring the integrity of pointers to sensitive data objects through ownership. Specifically, DOPE restricts pointer access to sensitive data objects (e.g., `cred`) only to their respective owner objects (e.g., `task_struct`). Although an attacker can manipulate a pointer to the owner object in an attempt to bypass DOPE, it is important to note that the sensitive pointer, such as the `cred` pointer, must still pass owner validation from its forged owner pointer, such as `task_struct`. Additionally, the attacker must find a valid execution path that does not cause a kernel panic due to the corrupted owner pointer. In summary, while a pointer-to-pointer attack is technically feasible, executing it may not be practical.

Confused deputy attack. A confused deputy attack [47] aims to trick a high-privilege function into performing access, violating the restricted access permissions. DOPE's trusted code design drastically reduces the exploitation surface of confused deputy attacks. Other isolation-based schemes [14, 45, 46, 64, 83] are vulnerable if an attacker corrupts a non-protected pointer that is dereferenced within trusted code. This allows the attacker to convert a low-privilege arbitrary read-and-write primitive to a high-privilege one. However, this conversion is not possible with DOPE's trusted code constraints, significantly improving protection against confused deputy attacks compared to existing countermeasures.

Even though there are some scenarios where DOPE is vulnerable to a confused deputy attack, the system's trusted code constraints make it challenging for attackers to exploit any vulnerabilities. Two possible attack scenarios are identified, where an integrity-ensured pointer temporarily stored on the stack could be corrupted or where the kernel stores the argument that will be written to the sensitive data object on the stack, which a TOCTTOU attack could exploit.

Although DOPE may have limitations regarding stack tampering, we view it as an opportunity for future research to enhance the protection of isolation-based schemes against confused deputy attacks.

Scalability. In our case study, we demonstrate the effectiveness of DOPE, as it protects eight sensitive data objects from exploitation. We firmly believe that these eight objects form an appropriate set for protection. Moreover, the flexibility of DOPE allows for expansion to safeguard additional objects. While this presents a standalone research question [65, 71] that requires further investigation, we see it as a promising avenue for future work.

Manual effort. Any kind of manual effort by developers may unintentionally introduce implementation bugs or instabilities, making it susceptible to Denial-of-Service (DoS). However, this is a fundamental issue in software development, particularly kernel development, which is one of the primary motivations behind DOPE. DOPE cannot eliminate the possibility of a developer introducing a security bug, but DOPE alleviates the security risk posed by the bug. To address the concern of introducing bugs while implementing the DOPE policies, we follow the state-of-the-art kernel software testing with the Linux Test Project (LTP) (cf. Section 6.5), which helped us attain a high code coverage and ensure the robustness of our system against potential DoS.

Comparison to PKU-based approaches. In comparison to approaches [35, 61, 68, 69, 79] based on Intel's PKU, DOPE addresses and resolves several kernel challenges inherent in PKS: Firstly, the kernel handles system events (i.e., exceptions, interrupts, and context switches), which attackers with memory can exploit write primitives to tamper with the PKS state. Our solution, detailed in Sections 5 and 6.1, introduces protections, preventing potential tampering attempts during these events. Secondly, the kernel manages low-level page permission handling (e.g., manipulating access permissions of pages, including MPK keys), posing a potential DOPE bypass. To counter this, Section 6 describes how we fortified page tables using DOPE, effectively eliminating the risk of page table tampering and subsequent DOPE bypasses. Thirdly, the kernel's memory allocator, the buddy allocator, recycles physical memory pages, requiring special handling. In Appendix 10, we describe how we adapted the allocator. Lastly, the Linux kernel combines sensitive and non-sensitive data within the same data structures. In Section 4.3, we propose three enforcement techniques, i.e., entire data object protection, shadow memory protection,

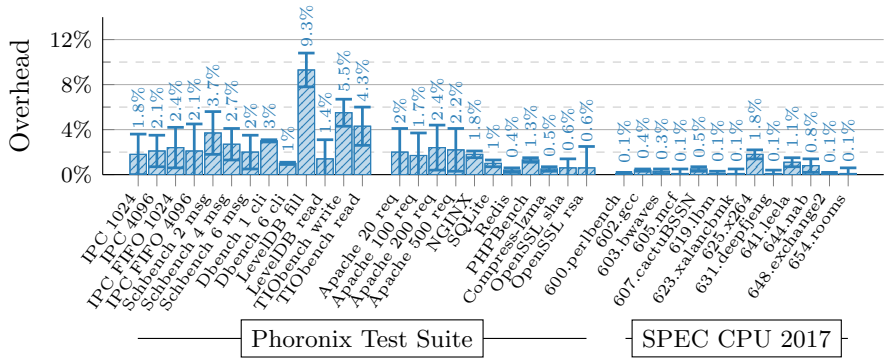


Figure 5.5: Overhead of macro-benchmarks.

and sensitive data protection, to securely and efficiently protect sensitive data.

Call gates. A primary security concern with PKU-based systems arises from the `wrpkru` instruction responsible for altering access rights. A malicious thread could execute this instruction, thereby changing its access rights [17, 80], e.g., using the kernel as a confused deputy. In response, researchers have introduced various countermeasures [61, 68, 69, 79]. These include advanced techniques for code/binary analysis and the integration of a call gate. Similarly, the PKS system uses the `wrmsr` instruction for modifying access rights. Kernel threads, by default, have unrestricted access to execute `wrmsr`. To fortify against this potential threat, DOPE has been designed to leverage call gates.

7.2. Performance Evaluation

We evaluate our DOPE proof-of-concept implementation's binary size, compile time, and performance overhead, where Appendix 13 shows the detailed results. We perform micro-benchmarks with LMbench [57], and macro-benchmarks with Phoronix Test Suite [62] and SPEC CPU 2017 [25]. Our benchmark CPU is Intel i7-1260P. We run Ubuntu 22.04.1 (kernel 5.19) as the Linux distribution.

Micro-benchmarks. We use LMbench to evaluate the latency and bandwidth overhead of our proof-of-concept. We consider the baseline kernel version 5.19, DOPE-light, and DOPE, where DOPE-light protects the same data objects as our case study DOPE, except for user-accessible

Table 5.3: Systematic overview of state-of-the-art mitigations against data-oriented attacks in the Linux kernel.

Mitigations	Technique	Protection Targets									Overhead
		Credentials	Virtual memory	Virtual memory areas	Inodes	Page tables	Filesystem mount	Other non-control data	User-accessible pages	Stored registers	
PrivGuard [64]	Monitoring	○	○	-	-	-	-	-	-	-	⧖
AKO [83]	Monitoring	○	-	-	-	-	-	-	-	-	⧖
PrivWatcher [14]	Monitoring	●	●	-	-	-	-	-	-	-	⧗[1]
SALADS [13]	Randomization	●	-	-	●	-	-	●[2]	-	-	⧗
PT-Rand [27]	Randomization	-	-	-	-	●	-	-	-	-	⧖
Mondrix [81]	Compartmentalization	-	-	-	-	-	-	●	-	●	⧗[1]
HAKC [55]	Compartmentalization	○	○	●	○	●	○	●	○	◐	⧗
KDPM [45]	Isolation	○	-	-	-	-	-	-	-	-	⧖[1]
KPRM [46]	Isolation	○	-	-	-	-	-	○	-	-	⧗
KENALI [71]	Isolation	◐	●	◐	◐	●	◐	●	-	●	⧗
xMP [63]	Isolation	◐	●	-	-	●	-	●[3]	-	-	⧗
DOPE	Isolation	●	●	●	●	●	●	-	●	●	⧖

● Strong protection ◐ Partial protection ○ Insufficient protection - Not protected

⧖ Low overhead ⧖ Reasonable overhead ⧗ High overhead

[1] Not tested on hardware [2] Non-sensitive data [3] User space data

pages. We include DOPE-light to highlight the overhead caused by protecting user-accessible pages. To achieve stable results, we run each benchmark 80 times and compute the mean and standard deviation, with the results shown in Table 5.4. We compute the total overhead by averaging over all overheads, resulting in an overhead of 32 % for DOPE and 17 % for DOPE-light.

Phoronix Test Suite macro-benchmarks. Our benchmarks from Phoronix Test Suite split up into stress tests and real-world applications, as shown in Figure 5.5. Among the stress tests are one inter-process communication, one kernel scheduler, two filesystem, and one threaded I/O benchmarks, while among the real-world applications are two web-server, two database, and four user application benchmarks. The average performance overhead of the Phoronix Test Suite macro-benchmarks is 2.3 %.

SPEC CPU 2017. We perform speed benchmarks of SPEC CPU 2017, as shown in Figure 5.5. The measured overheads of the macro-benchmarks are all below 1.8 %, consistent with the user application benchmarks from the Phoronix Test Suite. The overall overhead is calculated to be 0.4 % when averaging all the results.

8. Systematic Analysis

In this section, we systematically analyze existing mitigations against data-oriented attacks with a threat model aligned with ours. Table 5.3 illustrates the analysis results.

We categorize these mitigations into four techniques: Object monitoring, randomization, compartmentalization, and isolation. Moreover, we classify them based on the performance overhead they introduce. However, directly comparing DOPE's overhead with existing countermeasures is not possible as we have no access to the source code, kernel versions, kernel configurations, and hardware setup from these countermeasures. All of these factors influence the benchmark outcomes. As a result, any performance data reported in works should be treated as an estimate when comparing the performance across mitigations. We strive to perform a fair comparison with the following classification scheme.

We first consider macro-benchmark results as the primary criterion. If no macro-benchmarks are available, we rely on micro-benchmark results. For low overhead ⧖, macro-benchmarks are below 1 % or the micro-benchmarks are below 5 %. For reasonable overhead ⧗, we set the boundaries between 1 % to 3 % for macro-benchmarks and 5 % to 25 % for micro-benchmarks. For high overhead ⧗ mitigations have an overhead above 3 % for macro-benchmarks or 25 % for micro-benchmarks.

Monitoring. PrivGuard [64] protects credentials and the `pgd` by monitoring their changes and only permits its modification for high-privilege syscalls, e.g., `sys_set*id`. This monitoring involves duplicating these objects at the beginning of the syscall and checking them at both the beginning and end. However, an attacker may perform two attack scenarios. Firstly, the attacker modifies the sensitive data between the duplication and changes it back before the check. Secondly, during high-privilege syscalls, the kernel dereferences numerous untrusted pointers. The attacker may overwrite these pointers, enforcing these syscalls to perform a high-privilege write operation. Consequently, a low-privilege arbitrary write is converted into a high-privilege one. AKO [83] is similar PrivGuard, but it only protects credentials. Moreover, the duplicated data is not on the stack but on a reserved unprotected area.

PrivWatcher [14] protects credentials and the `pgd`. Compared to PrivGuard, PrivWatcher stores sensitive data in read-only domains and monitors its access. Furthermore, it assumes these domains are only writable

by PrivWatcher. With this assumption, an attacker cannot tamper with sensitive data, preventing their exploitation. As Quan et al. [14] discussed, this assumption was not supported by hardware. Hence, the actual performance overhead of PrivWatcher may be higher than the evaluated overhead.

Randomization. SALADS [13] protects sensitive (i.e., `cred` and `inode`) and non-sensitive (e.g., `list_head`) members of data objects by randomizing their layout at runtime. An attacker may manipulate the wrong data members since the data layout may change between the leak and attack phases. Hence, SALADS mitigates its exploitation. However, the protection level of SALADS strongly depends on how often the data objects are re-randomized. Moreover, the re-randomization rate determines the performance overhead.

PT-Rand [27] randomizes the location of all page tables by mapping them with an offset to a random base instead of the DPM. This random base is stored in a dedicated inaccessible register. Furthermore, PT-Rand ensures no leakage of the random location by substituting page table references with an offset to this random base. Therefore, the location of page tables cannot be leaked, preventing the manipulation of page tables. Davi et al. [27] achieve this strong security claim with a low runtime overhead.

Compartmentalization. Mondrix [81] provides memory protection by proposing significant hardware changes to add multiple protection features, a concept of ownership, and protection domains. A separate permission table stored in the main memory provides more fine-grained control over the memory access rights. Stacks are only writable within a thread's current stack frame. Witchel et al. [81] introduce a dedicated stack permission table for access outside the current stack frame. Access to functions that run in a higher privilege domain uses a new form of lightweight call gates that push the return address to a shadow call stack. Furthermore, Mondrix adds new caches for the added protection checks and call gates to increase performance. However, these significant hardware changes prevent the use of Mondrix today.

HAKC [55] performs compartmentalization of Loadable Kernel Modules (LKMs). It moves all data objects accessible by the LKM into partitions, where each data object belongs to exactly one partition. Each partition and hence data object can only be accessed by its owner, defined on compile time. Since HAKC does not account for simultaneous

ownership, data objects with simultaneous readers cannot be compartmentalized. In the Linux kernel, simultaneous readers are very common, e.g., RCU-locked credentials and inodes [56]. Hence, we mark all data objects with simultaneous readers as insufficiently protected. Credentials and virtual memory are shared between threads within one process. Inodes, filesystem mounts, and user-accessible pages can also be accessed simultaneously via the DPM [8, 76], e.g., during a pathname lookup. Furthermore, HAKC protects the stack only on compartment granularity and does not account for concurrent threads in these compartments. Therefore, in case of an exploitable bug, a thread can overwrite the stack (including stored registers on preemption) from another thread if they are in the same compartment. Although the HAKC proof-of-concept implementation only has two compartments, we classify the runtime overhead as high.

Isolation. KDPM [45] protects sensitive kernel data by only permitting certain syscalls (`sys_execve` and `sys_set*id`) to grant write permissions. However, since write permissions are granted to the entire syscalls, an attacker can tamper unprotected pointers which are dereferenced within these syscalls to obtain a high-privilege arbitrary write primitive. Moreover, KDPM is susceptible to forgery attacks. Lastly, they evaluated KDPM on an MPK emulator instead of real hardware.

KPRM [46] protects sensitive kernel data during syscalls by unmapping them from the threads' address space. To manage sensitive data access, KPRM hooks the page-fault handler. KPRM maps a restricted page if the access is allowed within the executed code or kills the process if the access is invalid. However, KPRM does not account for multiple threads with a shared kernel address space. A thread executing kernel code can access sensitive data currently mapped for a different thread within the same thread group. Furthermore, the high reliance on frequent page faults and unmapping restricted pages leads to high performance overhead.

Song et al. [71] proposed an automated tool for identifying sensitive kernel data objects. Moreover, they proposed mitigation KENALI protects these objects and the stack with shadow memory. KENALI also prevents various mitigation-bypass attacks. Unfortunately, they lack a hardware primitive to protect sensitive data efficiently, leading to high performance overhead. KENALI does not mitigate against the discussed **arbitrary use-after-free** attack from Section 7.1. Therefore, an attacker can convert an arbitrary write primitive to an arbitrary use-after-free primitive and perform DirtyCred [50], resulting in a privilege escalation KENALI cannot protect against. Since the principle of DirtyCred can be applied to `cred`,

`vm_area`, `vfsmount`, and all other objects allocated with a `kmem_cache`, these data objects are only partially protected.

xMP [63] employs Extended Page Table (EPT) switching to enforce domain protection similarly to DOPE. It protects page tables, credentials, the `pgd`, and sensitive data mapped in user space. Unfortunately, xMP does not mitigate against the discussed **arbitrary use-after-free** attack because it only ensures pointer integrity instead of ownership like DOPE. Therefore, the credentials are only partially protected. Besides credentials, xMP protects the `pgd` and page tables sufficiently. Even though xMP only protects three kernel data objects, their performance overhead is high.

We deploy DOPE to protect credentials, virtual memory, virtual memory areas, inodes, page tables, filesystem mount, stored registers, and user-accessible pages with strong security guarantees while maintaining a reasonable performance overhead.

9. Conclusion

In this paper, we presented our principled mitigation DOPE to protect against data-oriented attacks. DOPE enforces domain protection by restricting memory accesses during kernel execution based on the principle of least privilege. We implemented a DOPE proof-of-concept and conducted a case study that protects eight sensitive data objects from being used for privilege escalation exploits. For our proof-of-concept, we observed a reasonable performance overhead of 2.3 % for real-world user applications, significantly improving in terms of security with respect to performance over existing mitigations.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. Furthermore, we thank Gaëtan Cassiers, Martin Unterguggenberger, and Andreas Kogler for valuable discussions and feedback on this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Additionally, this work was supported by Red Hat. Any opinions, findings, 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 (p. 102).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. In: TISSEC (2009) (p. 102).

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In: S&P. 2008 (p. 106).

[4] Khalid Aziz. Add support for eXclusive Page Frame Ownership. Feb. 2019. URL: `https://lwn.net/Articles/779818/` (p. 142).

[5] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. Hardware assisted randomization of data. In: RAID. 2018 (p. 106).

[6] Sandeep Bhatkar and R Sekar. Data space randomization. In: DIMVA. 2008 (p. 106).

[7] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In: AsiaCCS. 2011 (pp. 102, 107).

[8] Neil Brown. RCU-walk: faster pathname lookup in Linux. July 2015. URL: `https://lwn.net/Articles/649729/` (p. 129).

[9] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: ACM Conference on Computer and Communications Security (CCS). 2008 (pp. 102, 107).

[10] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. In: (2008) (p. 106).

[11] Nicholas Carlini and David A. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In: USENIX Security Symposium. 2014 (pp. 102, 107).

[12] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In: OSDI. 2006 (p. 106).

[13] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. A Practical Approach for Adaptive Data Structure Layout Randomization. In: European Symposium on Research in Computer Security. 2015 (pp. 102, 103, 106, 107, 117, 126, 128, 140).

[14] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. PrivWatcher: Non-Bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In: AsiaCCS. 2017 (pp. 102, 103, 106, 107, 117, 121, 123, 126–128, 140).

[15] 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 (pp. 102, 105).

[16] Chromium. PartitionAlloc Design. 2018. URL: https://chromium.googlesource.com/chromium/src/+/lkcr/base/allocator/partition_allocator/PartitionAlloc.md (p. 138).

[17] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In: USENIX Security Symposium. 2020 (p. 125).

[18] Jonathan Corbet. Defending against Rowhammer in the kernel. Oct. 2016. URL: https://lwn.net/Articles/704920/ (p. 107).

[19] Jonathan Corbet. Exclusive page-frame ownership. Sept. 2016. URL: https://lwn.net/Articles/700647/ (p. 142).

[20] Jonathan Corbet. Kernel support for control-flow enforcement. 2018 (p. 102).

[21] Jonathan Corbet. Memory protection keys for the kernel. 2020. URL: https://lwn.net/Articles/826554/ (pp. 103, 105).

[22] Jonathan Corbet. Seeking an API for protection keys supervisor. May 2022. URL: https://lwn.net/Articles/894531/ (p. 105).

[23] Jonathan Corbet. Solutions for direct-map fragmentation. May 2022. URL: https://lwn.net/Articles/894557/ (p. 114).

[24] Jonathan Corbet. Supervisor mode access prevention. Sept. 2012. URL: https://lwn.net/Articles/517475/ (pp. 102, 107).

[25] Standard Performance Evaluation Corporation. SPEC CPU 2017. 2017. URL: https://www.spec.org/cpu2017/ (pp. 103, 121, 125).

[26] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In: S&P. 2014 (p. 107).

[27] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In: NDSS. 2017 (pp. 102, 103, 106, 107, 117, 126, 128, 140).

[28] Jake Edge. Control-flow integrity for the kernel. 2020. URL: `https://lwn.net/Articles/810077/` (p. 102).

[29] Jake Edge. Extending the use of RO and NX. 2011. URL: `https://lwn.net/Articles/422487/` (p. 102).

[30] Jake Edge. Kernel address space layout randomization. 2013. URL: `https://lwn.net/Articles/569635/` (p. 107).

[31] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: arXiv:2303.16353 (2023) (p. 107).

[32] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In: Euro S&P. 2016 (p. 107).

[33] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 107).

[34] Daniel Gruss, Michael Schwarz, and Moritz Lipp. Meltdown: Basics, Details, Consequences. In: BlackHat USA. 2018 (p. 107).

[35] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and Efficient Memory Protection Keys. In: USENIX Security Symposium. 2022 (pp. 106, 124).

[36] Cyril Hrubis. Linux Test Project. 2022. URL: `https://github.com/linux-test-project/ltp` (p. 120).

[37] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In: S&P. 2016 (pp. 102, 105).

[38] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: USENIX Security Symposium. 2009 (pp. 102, 107).

[39] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers. May 2019 (p. 105).

[40] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security Symposium. 2014 (p. 142).

[41] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In: USENIX Security Symposium. 2012 (p. 102).

[42] kernel.org. Virtual memory map with 4 level page tables (x86_64). 2009. URL: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt (p. 140).

[43] Michael Kerrisk. capabilities(7) — Linux manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html. Aug. 2021 (p. 140).

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 107).

[45] Hiroki Kuzuno and Toshihiro Yamauchi. KDPM: Kernel Data Protection Mechanism Using a Memory Protection Key. In: International Workshop on Security (2022), pp. 66–85 (pp. 102, 103, 106, 107, 117, 123, 126, 129).

[46] Hiroki Kuzuno and Toshihiro Yamauchi. Prevention of Kernel Memory Corruption Using Kernel Page Restriction Mechanism. In: Journal of Information Processing 30 (2022), pp. 563–576 (pp. 102, 103, 106, 107, 117, 123, 126, 129).

[47] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In: NDSS. 2022 (p. 123).

[48] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. In: Architectural Support for Programming Languages and Operating Systems. 2022 (p. 106).

[49] Guanyu Li, Dong Du, and Yubin Xia. Iso-UniK: lightweight multi-process unikernel through memory protection keys. In: Cybersecurity 3 (2020), p. 11 (p. 106).

[50] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In: ACM. 2022 (pp. 122, 129).

[51] 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 (p. 107).

[52] Yong Liu, Jun Yao, and Xiaodong Wang. USMA: Share Kernel Code with Me. In: BlackHat Asia (2022) (p. 140).

[53] LLVM. The LLVM Compiler Infrastructure. 2019. URL: `https://llvm.org` (pp. 103, 114).

[54] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (p. 99).

[55] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In: NDSS. 2022 (pp. 102, 103, 106, 107, 117, 126, 128).

[56] Paul McKenney. What is RCU, Fundamentally? Dec. 2007. URL: `https://lwn.net/Articles/262464/` (pp. 129, 139).

[57] Larry McVoy and Carl Staelin. Lmbench: Portable Tools for Performance Analysis. In: USENIX ATC. 1996 (pp. 103, 121, 125).

[58] Joao Moreira. Kernel FineIBT Support. Apr. 2022. URL: `https://lwn.net/Articles/891976/` (pp. 102, 107).

[59] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios Kemerlis. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel. In: Black Hat Asia. 2017 (p. 102).

[60] James Morse. arm64: kernel: Add support for Privileged Access Never. 2015. URL: `https://lwn.net/Articles/651614/` (p. 102).

[61] 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 (pp. 106, 110, 124, 125).

[62] Phoronix. OpenBenchmarking. 2022. URL: `https://openbenchmarking.org` (pp. 103, 121, 125).

[63] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In: S&P. 2020 (pp. 102, 103, 106, 107, 117, 126, 130, 139, 140).

[64] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks. In: IEEE Access 6 (2018), pp. 46584–46594 (pp. 102, 103, 106, 107, 117, 123, 126, 127, 140).

[65] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. uSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In: RAID. 2021 (p. 124).

[66] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In: Proceedings of the IEEE 63.9 (1975), pp. 1278–1308 (pp. 103, 107).

[67] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In: Architectural Support for Programming Languages and Operating Systems. 2021 (p. 106).

[68] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In: USENIX Security Symposium. 2022 (pp. 124, 125).

[69] 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 (pp. 106, 110, 124, 125).

[70] Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. Retrieved on June 26, 2015. Mar. 2015. URL: `http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html` (p. 107).

[71] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In: NDSS. 2016 (pp. 102, 103, 106, 107, 121, 124, 126, 129, 140, 141).

[72] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-Unikernel Isolation with Intel Memory Protection Keys. In: ACM. 2020 (p. 106).

[73] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In: USENIX Security Symposium. 2022 (p. 117).

[74] The Linux Kernel. File system drivers (Part 2). 2021. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/filesystems_part2.html (pp. 120, 140).

[75] The Linux Kernel. Index Nodes. 2022. URL: https://www.kernel.org/doc/html/latest/filesystems/ext4/inodes.html (p. 140).

[76] The Linux Kernel. Locking. 2022. URL: https://www.kernel.org/doc/html/latest/filesystems/locking.html (p. 129).

[77] The Linux Kernel. Memory Allocation Guide. 2022. URL: https://docs.kernel.org/core-api/memory-allocation.html?highlight=kmem_cache_alloc (p. 138).

[78] The Linux Kernel. Memory Protection Keys. 2022. URL: https://www.kernel.org/doc/html/latest/core-api/protection-keys.html (p. 105).

[79] 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 (pp. 106, 124, 125).

[80] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In: EuroSys. 2022 (p. 125).

[81] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In: ACM SIGOPS Operating Systems Review. 2005 (pp. 102, 103, 106, 107, 126, 128).

[82] Jidong Xiao, Hai Huang, and Haining Wang. Kernel data attack is a realistic security threat. In: International Conference on Security and Privacy in Communication Systems. 2015 (pp. 102, 105).

[83] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes. In: International Journal of Information Security 20 (2021) (pp. 102, 103, 106, 107, 117, 123, 126, 127).

Appendix

10. Implementation Details of Trusted Code

In this section, we provide measures to address any implementation issues while ensuring that our code adheres to the trusted code constraints we defined in Section 4.5.

Memory management. The buddy allocator in the Linux kernel allocates contiguous physical memory in page order chunks, i.e., $2^n \cdot PAGE\_SIZE$. On top of the buddy allocator sits the slab allocator, and stores caches of available objects with a desired predefined size [77]. The kernel supports three slab allocators: SLAB, SLOB, and SLUB, all of which store metadata on the allocated page. Allocations via `kmem_cache` and our modified `kmem_dope_cache` deploy one of these slab allocators, i.e., SLUB. If a domain protects a page allocated by the buddy allocator, then allocating or freeing an object via `kmem_dope_cache` would require write permission to the domain, as metadata may be written to the protected page. However, since during allocation and freeing, untrusted pointers are accessed, granting write permission would violate trusted code constraints.

To address this issue, we propose to extend the slab allocator by adopting a PartitionAlloc-based design similar to Chrome [16], which separates data and metadata into two distinct locations. This approach would eliminate the need for write permission to the domain when allocating or freeing an object via our `kmem_dope_cache`. While implementing this extension requires significant effort, we acknowledge that it is outside the scope of this work.

Outsourcing the metadata of the slab allocator to an unprotected object does not pose a security risk because DOPE does not rely on the allocator's trustworthiness. In Section 7.1, we illustrate various attacks on the allocator and show how DOPE mitigates them.

```
1  /**
2   * struct callback_head - callback structure for use with
3   * RCU and task_work
4   * @next: next update requests in a list
5   * @func: actual update function to call after the grace
6   *        period.
7   */
8  struct callback_head {
9    struct callback_head *next;
10   void (*func)(struct callback_head *head);
11 } __attribute__((aligned(sizeof(void *))));
12
13 /* Types */
14 #define rcu_head callback_head
15 typedef void (*rcu_callback_t)(struct rcu_head *head);
16
17 /**
18  * call_rcu() - Queue an RCU callback for invocation
19  * after a grace period.
20  */
21 void call_rcu(struct rcu_head *head, rcu_callback_t func){
22   ...
23   head->func = func;
24   head->next = NULL;
25   ...
26 }
```

Listing 5.3: Callback function provided by Linux's RCU locking mechanism.

Read-Copy Update. The Linux kernel supports the efficient synchronization mechanism Read-Copy Update (RCU) that enables concurrent updates by readers [56]. Besides blocking the current thread for synchronization, RCU also supports non-blocking updates by invoking a callback function either during a software interrupt (i.e., `softirq`) or by a dedicated RCU thread [63]. During the callback, data (stored as generic data type, i.e., `rcu_head`) is accessed that may or may not be protected by a domain. This is illustrated in Lines 23 and 24 from Listing 5.3, where the function `call_rcu` access the `head` pointer. If `head` is stored in a sensitive and protected data objects, e.g., `struct cred`, than this function would require write permission to the corresponding domain, e.g., credential domain. Otherwise, the hardware raises an fault, and DOPE would interpret this as an exploitation attempt. However, granting temporary access permission would violate trusted code constraints. To address this issue, we separated the `rcu_head` member from sensitive data objects and instead stored a pointer to a dynamically allocated `rcu_head`.

11. Detailed sensitive data objects

Credentials. The credential struct contains information on its thread's privilege level and is stored as a pointer in the `task_struct`. It is a popular attack target [13, 14, 63, 64, 71], cf. CVE-2021-26708, CVE-2017-16995, or CVE-2017-2636. Overwriting credentials immediately leads to privilege escalation as the kernel interprets the thread as high-privilege. In addition to traditional UNIX per-process credentials (i.e., `*id`), Linux supports per-thread capabilities [43]. These capabilities partition the privileges associated with the superuser into a finer granularity. Hence, we also protect capabilities.

Inodes. The inode in a UNIX filesystem, e.g., `ext4`, stores all metadata associated with its file [75]. Among this metadata is non-sensitive data, e.g., last modified or last access time, and sensitive data, e.g., access rights and owner and group identifier. By modifying the sensitive data of an inode, an attacker can change the permission or owner of the associated file. Moreover, we define the flag variable as sensitive because it contains information on whether the file is private or immutable. Another way to use inodes for privilege escalation is to forge their identifiers, uniquely identifying the inodes within the mounted file system [74]. In summary, we protect `i_ino`, `i_mode`, `i_uid`, `i_gid`, and `i_flags`.

Page tables. Page tables are valuable targets for attackers because they contain lower-level page permissions [27, 63, 71]. By overwriting those permission bits, an attacker controls the access permissions of the lower page levels, e.g., an attacker can modify permission bits in the page-table entry to change a kernel code page to writable. As a result, the attacker has a writable and executable kernel memory area.

Virtual memory areas. In Linux, there is another possibility to manipulate the permission bits of page tables by tampering with the `vm_page_prot` stored in `vm_area_struct`. Liu et al. [52] demonstrated a data-oriented attack called User Space Mapping Attack, in which an attacker overwrites `vm_page_prot` to modify the permissions of page-table entries. This attack results in a kernel page being mapped into the user space, which is then overwritten.

Virtual memory. In the Linux kernel, the thread's `pgd` is the top level of a page table [42] and is stored in the `mm_struct`. By overwriting the stored `pgd`, an attacker has control over the hardware `pgd` and may forge a virtual to physical page mapping [14, 63, 64, 71]. The attacker then

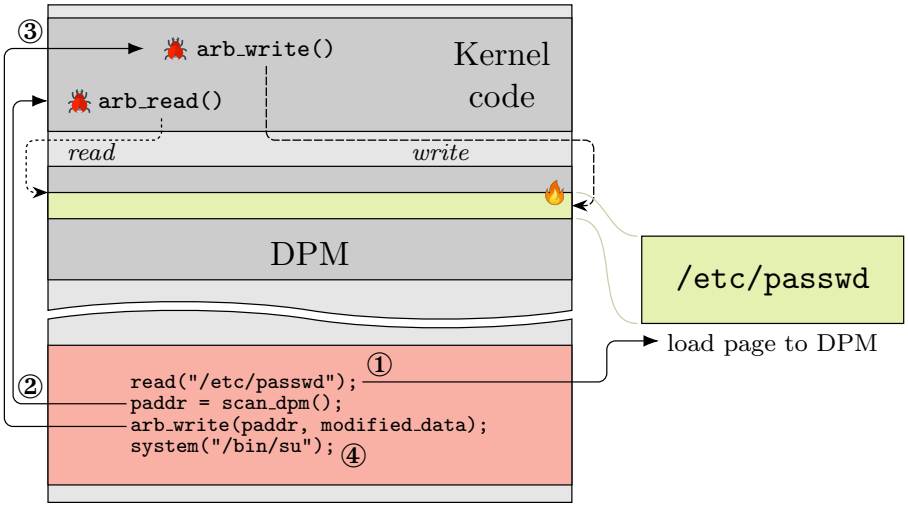


Figure 5.6: DPM-FPATCH attack on `/etc/passwd` file. Step ① loads the read-only file to the DPM, while step ② performs an arbitrary read call for each page read of the DPM to obtain the address where the file content is located. Next, step ③ carefully overwrites sensitive content of the file via the DPM to grant the attacker root privileges without authentication ④.

may add virtual addresses that map to arbitrary physical addresses with arbitrary permissions.

Filesystem mount. Song et al. [71] showed a data-oriented attack in which an attacker tampers with mount flags. The attacker manipulates the flag to mark a mounted file system as no longer read-only. Hence, the attacker can write to files within the read-only file system. The target is the system partition, which is read-only mounted on most Android devices.

12. DPM-FPATCH attack

Operating systems have the important task of managing privilege levels and ensuring the isolation of processes. It is crucial to prevent low-privilege processes from tampering with the data of other processes. However, since the entire physical memory is mapped via the Direct Physical Mapping (DPM), an attacker can use an arbitrary write primitive in the kernel to manipulate data on the DPM, in particluar with including user-accessible data. This

introduces a new variant of data-oriented attacks called DPM-FPATCH, which current state-of-the-art mitigations are unable to protect against.

12.1. Attack

The DPM-FPATCH data-oriented attack variant overwrites data of any user-accessible file. With DPM-FPATCH, we demonstrate an attack on the `/etc/passwd` file, as illustrated in Figure 5.6.

In step ①, an attacker opens and reads the entire content of the high-privilege but user-accessible file. Hence, the kernel loads the content from the disk into the DPM. In step ②, the attacker scans the entire DPM with the arbitrary read and obtains the address where the file content is stored. In step ③, the attacker can use the arbitrary write to overwrite the content via the DPM. One possible modification of a high-privilege file is to change the first line of `/etc/passwd` from `root:x:0:0:root:/root:/usr/bin/zsh` to `root::00:0:root:/root:/usr/bin/zsh`. This change indicates that a root login does not require authentication ④.

In summary, an attacker can perform DPM-FPATCH to modify the data of any user-accessible file. For our demonstration, the attacker modifies `/etc/passwd` to illegally indicate to the system that root does not require authentication on login, elevating the attacker's privileges.

12.2. Potential mitigation

Kemerlis et al. [40] showed the devastating outcome of user data accessibility via the DPM for control-flow hijacking attacks. To prevent this accessibility, their proposed mitigation, XPFO, prevents all accesses to user data via the DPM by either mapping a page in user space or the DPM, but never both. Since DPM-FPATCH does not rely on this mapping, XPFO does not prevent this attack variant. Moreover, according to Linux kernel developers [4, 19], who have extensively tested XPFO, the runtime overhead is above 30 %. Therefore, the XPFO patch was never merged into the Linux kernel.

13. Detailed Evaluation Results

Binary size and compile time overhead. To enforce domain protection, our LLVM pass inserts functions for domain switching and validation. These inserted functions increase the binary size and the compile time. To illustrate both overheads, we compile an unmodified Linux kernel version 5.19 with clang version 15.0.1 as a baseline. We then compile our modified Linux kernel with our LLVM pass enabled for our proof-of-concept implementation. The results are that the binary size and compile time increase by 0.27 % and $1.5 \pm 0.3\%$, respectively.

Micro-benchmarks. We use LMbench to evaluate the latency and bandwidth overhead for various benchmarks of our proof-of-concept implementation. We consider the baseline kernel version 5.19, DOPE-light, and DOPE, where DOPE-light protects the same data objects as our case study DOPE, except for user-accessible pages. We include DOPE-light to highlight the overhead caused by protecting user-accessible pages. To achieve stable results, we run each benchmark 80 times and compute the mean and standard deviation.

Table 5.4 illustrates the evaluation results. The null syscall benchmark indicates that DOPE does not add any syscall entry or exit latency. All syscalls interacting with user-accessible pages have an increased runtime overhead because DOPE switches domains on every user-accessible data access. The micro-benchmarks open, read, and write, show the increased overhead with between 98 % to 128 %. Except for open, DOPE-light reduces the syscall's overhead to about 0 % compared to DOPE. The open syscall has a performance overhead of $110 \pm 3\%$ because of the inserted validation checks and domain switches for the credential domain. Similar to open, fstat also performs validation checks as illustrated with about 15 % runtime increase for DOPE and DOPE-light. Since the pipe syscall interacts with user-accessible data, it has an elevated overhead of $21 \pm 0.1\%$ for DOPE. The overhead for all three process operations, fork, fork+exec, and shell, are 29 % to 37 % for DOPE and 24 % to 30 % for DOPE-light. The synchronization syscall, sem, has negligible runtime overhead. Since, during the pagefault benchmark, a thread accesses page tables, DOPE and DOPE-light increase the performance overhead by about 11 % due to page table domain switches. We show that the overhead caused by a pagefault is negligible when considering the access time to a page from DRAM, as the overhead of the dram page benchmark is between 1 % to 2.1 %. A signal fault has an overhead of about 37 % for both DOPE and

DOPE-light. The overhead of signal install and catch is negligible. We compute the total overhead by averaging over all overheads, resulting in an overhead of 32 % for DOPE and 17 % for DOPE-light.

With all four memory and mmap bandwidth benchmarks, Table 5.4 shows that DOPE does not add any runtime overhead on normal memory accesses, independent of how the memory is allocated. We run the file read benchmark with two parameters: open2close and io_only, and observe an overhead of 72 % to 87 % for a filesize of 4 kB. The overhead decreases to about 4 % by running the benchmark with a size of 1 MB. For DOPE-light, file read with the open2close parameter has an overhead of 43 % caused by the open syscall.

Phoronix Test Suite macro-benchmarks. Our benchmarks from Phoronix Test Suite split up into stress tests and real-world applications, as shown in Figure 5.5. Among the stress tests are one inter-process communication, one kernel scheduler, two filesystem, and one threaded I/O benchmarks. For the inter-process communication benchmark (IPC-benchmark), we observe that DOPE elevates the overhead by about 2 % independent of the used pipe (unnamed or named FIFO) and message size (1 024 Byte or 4 096 Byte). We run Schbench, which evaluates our scheduler, with two worker threads, each creating two, four, or six messenger threads. DOPE has an overhead between 2 % to 3.7 % for the scheduler benchmark, decreasing with more messenger threads. With Dbench, we perform two benchmark tests resulting in an overhead of $3 \pm 0.1\%$ for one and $1 \pm 0.1\%$ for six client threads. DOPE has an elevated performance overhead for LevelDB fill of $9.3 \pm 1.5\%$ due to extensive write syscall usage and, thus, extensive user-accessible domain switches. Since the LevelDB read benchmark caches read data in software, there are fewer syscalls and domain switches. The threaded I/O stress test (TIObench) has an overhead of $4.3 \pm 3.1\%$ for the read and $5.5 \pm 1.8\%$ the write benchmark.

Among the real-world applications are two web-server, two database, and four user application benchmarks (cf. Figure 5.5). The two web-server benchmarks, Apache and NGINX, has an runtime overhead of about 2 %, with the number of Apache requests having little effect on the overhead. DOPE affects the SQLite benchmark with an overhead of $3.4 \pm 1.2\%$, while the in-memory Redis benchmark is affected with a low overhead of $0.4 \pm 0.2\%$. Lastly, all four user applications, PHPBench, compress-lzma, and OpenSSL sha and rsa, results in a low overhead between 0.5 % to 1.3 %.

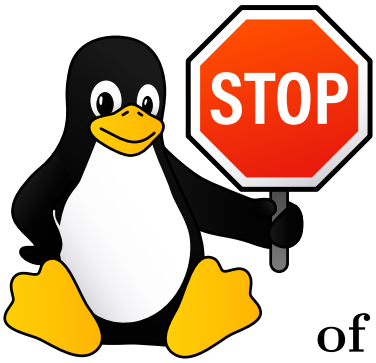
We observe an elevated standard deviation for various benchmarks, especially for those with multiple threads and high kernel execution time. However, the baseline and DOPE-enhanced kernel binary show similar levels of high standard deviation. Therefore, the noise properties of the kernel, such as hardware interrupts or context switches, contribute to the high standard deviation.

The average performance overhead of the Phoronix Test Suite macro-benchmarks is 2.3 %.

SPEC CPU 2017. We perform multiple speed macro-benchmarks of SPEC CPU 2017, as illustrated in Figure 5.5. All measured overheads of the speed macro benchmarks are below 1.8 %, which is in line with the user application macro-benchmarks from Phoronix Test Suite. We compute the overall overhead by averaging all results, leading in an overhead of 0.4 %.

Table 5.4: Micro-benchmark results.

	Benchmarks	Baseline	Overhead in %	
			DOPE-light	DOPE
Latency in µs	syscall null	0.08	-0.1 ± 0.4	0.0 ± 0.4
	syscall open+close	1.03	107.7 ± 2.9	127.8 ± 2.9
	syscall read	0.15	0.6 ± 2.3	98.0 ± 2.6
	syscall write	0.11	0.0 ± 0.7	128.0 ± 0.7
	syscall fstat	0.16	16.0 ± 1.6	15.0 ± 0.4
	syscall pipe	3.65	1.1 ± 0.4	21.0 ± 0.1
	proc procedure	0.002	-1.6 ± 4.4	0.5 ± 1.4
	proc fork	62.5	27.0 ± 1.6	34.0 ± 1.8
	proc fork+exec	211	29.9 ± 1.0	37.0 ± 0.8
	proc shell	458	24.0 ± 0.5	29.0 ± 0.5
	sem	0.46	-7.7 ± 6.3	-5.0 ± 4.3
	pagefault	0.15	12.0 ± 0.5	11.0 ± 0.5
	dram page	1.63	1.0 ± 3.3	2.1 ± 2.5
	signal fault	0.42	37.0 ± 1.4	38.0 ± 0.7
	signal install	0.14	0.4 ± 1.0	0.5 ± 1.1
	signal catch	0.88	0.4 ± 0.4	0.6 ± 0.6
Bandwidth in MB/s	mem rd 4 k	160	0.0 ± 0.2	0.0 ± 0.2
	mem wr 4 k	110	-0.1 ± 0.1	0.0 ± 0.1
	mmap rd 4 k	51.4	0.3 ± 0.5	0.2 ± 0.4
	mmap rd 1 M	48.0	-0.7 ± 1.6	-3.0 ± 1.2
	file_rd o2c 4 k	2.71	43.0 ± 3.7	72.0 ± 1.1
	file_rd o2c 1 M	17.6	1.5 ± 1.1	4.7 ± 0.7
	file_rd io 4 k	6.70	6.5 ± 0.4	87.0 ± 0.5
	file_rd io 1 M	18.1	0.3 ± 0.9	4.1 ± 0.7



6

Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI

Publication Data

Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In: AsiaCCS. 2024

Contributions

The author of this thesis is the main author of this work. The author's contributions are the proposal of *kernel control-data integrity* and *HEK-CFI*, *POC*, *case study*, and *evaluation*, as well as most of the written text.

Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI

Lukas Maar[1] Pascal Nasahl[2] Stefan Mangard[1]

[1] Graz University of Technology [2] Independent Researcher

Abstract

Over the past decade, vulnerabilities in the Linux kernel have more than doubled, allowing control-flow hijacking attacks that compromise the entire system. To thwart these attacks, Control-Flow Integrity (CFI) has emerged as state-of-the-art. However, existing kernel CFI schemes are still limited in providing protection against these attacks, e.g., during system events and for return addresses.

In this paper, we introduce Hardware-Enforced Kernel Control-Flow Integrity (HEK-CFI), a novel approach that protects control-flow-related data during system events, as well as function pointers and return addresses, effectively mitigating control-flow hijacking attacks. HEK-CFI leverages Intel CET, specifically write-protected pages used by its shadow stack design, along with signature-based CFI to safeguard this data. To demonstrate its effectiveness, we implement a proof-of-concept and perform a case study on the Linux kernel v5.18. In our case study, HEK-CFI eliminates all illegal backward-edge targets and reduces forward-edge targets by more than 50 % compared to all existing kernel CFI schemes. We evaluate our proof-of-concept on real hardware and observe a performance overhead of 12.3 % for micro benchmarks and 1.85 % for macro benchmarks. In summary, HEK-CFI is the first solution to provide protection for both system events and return addresses. HEK-CFI also generically reduces forward control-flow targets and the performance overhead compared to existing solutions.

1. Introduction

Over the past decade, the number of vulnerabilities in the Linux kernel has increased dramatically, from 114 CVEs in 2012 to 318 in 2022, according

to the NIST NVD. These vulnerabilities can be devastating, allowing to hijack the control flow to gain arbitrary code execution and compromise the entire system.

Despite efforts by processor vendors [14, 19, 51] and kernel developers [20, 34, 35] to mitigate these attacks, adversaries continued to devise more sophisticated code reuse attacks [5, 7, 10, 28]. For instance, Return Oriented Programming (ROP) [7] redirects the control flow to a chain of gadgets, each ending with `ret`. To counter control-flow hijacking attacks, Control-Flow Integrity (CFI) [1] has emerged as a state-of-the-art mitigation by restricting the control flow to an approximated Control-Flow Graph (CFG).

Kernel CFI-based mitigations face two critical challenges in providing security. First, kernel programs must handle system events [6], i.e., context switches, interrupts, exceptions, and syscalls. During these events, the current thread state (which refers to the set of registers that store its run-time information) is stored in memory, ensuring that it can be restored when the thread resumes execution. If these stored states are not fully protected on any of these events, an adversary can corrupt them to hijack the control flow when the state is restored, as described in prior research [16, 23] and exploited by security researchers at Google Project Zero (cf. CVE-2022-42703) [33]. Second, it is difficult to establish an accurate representation of the CFG for kernel programs due to their large size. Coarse approximations leave the system vulnerable to bypass attacks. This is particularly critical for the backward edges, as static determination leaves the system vulnerable to Control-Flow Bending (CFB) [9], re-enabling control-flow hijacking attacks.

Existing kernel CFI-based mitigations [3, 16, 18, 22, 23, 43, 52, 64, 65, 70] are still limited in addressing these challenges as they do not provide sufficient protection for both the thread state during all system events and backward edges, i.e., return addresses. For example, defenses based on ARM's Pointer Authentication (PA), e.g., PAL [64] and Camouflage [18], inadequately protect the stored thread state, allowing attackers to tamper with it and hijack the control flow. Other approaches, e.g., KCoFI [16] and Fine-CFI [43], statically determine backward edges, which makes their systems vulnerable to CFB, bypassing the applied mitigation. Additionally, the Code Pointer Integrity (CPI) [39] solution combined with the user-space memory isolation scheme CETIS [68] safeguards code pointers and return addresses. However, this combination cannot address the first challenge because it is not designed for kernel-level system events that

require special thread state handling. Consequently, it cannot protect the kernel during these critical events.

In summary, existing kernel CFI schemes and CPI combined with CETIS have limitations in providing protection for the kernel during system events and for return addresses. As a consequence, adversaries can exploit these limitations to bypass the applied CFI scheme, thereby re-enabling control-flow hijacking attacks.

In this paper, we introduce Hardware-Enforced Kernel Control-Flow Integrity (HEK-CFI) to fill the gap in protecting against kernel control-flow hijacking attacks. HEK-CFI consists of three key mechanisms. First, HEK-CFI ensures kernel control-data integrity by retrofitting write-protected pages from Intel CET SHadow StacK (SHSTK) for the *supervisor*. It does this by providing write-protected local and global safe areas within the kernel, where control data is securely stored. In particular, our write-protected local safe area extends well beyond the original purpose of Intel CET SHSTK. While SHSTK protects backward edges, our approach safeguards any local control data, such as during low-level environments like system events. Second, HEK-CFI utilizes our control-data integrity scheme to protect the thread state during all system events, including protection for the SHSTK state. This prevents attackers with memory write capabilities from tampering with the thread and SHSTK state. With these two mechanisms, we are the first to provide comprehensive protection for both system events and return addresses. Third, HEK-CFI combines our control-data integrity with signature-based CFI to protect forward control-flow edges, particularly control data, e.g., function pointers. Signature-based CFI efficiently protects function pointers with rare signatures, while control-data integrity fully safeguards any control data, albeit with a potentially higher performance overhead. To balance this trade-off, HEK-CFI automatically selects the optimal scheme for each control data

While HEK-CFI's true contributions rely on efficient control data (including thread state) protection, it demonstrates its practical use with the third mechanism to protect forward edges as well.

We are the first to implement the Intel CET SHSTK for the *supervisor* in the Linux kernel. This allows us to implement a HEK-CFI proof-of-concept, realized as a compiler-assisted software framework consisting of a Linux kernel extension, a code analyzer [24], and an LLVM pass [40].

Our framework automatically hardens the Linux kernel using a user-configurable CFI precision level as input. To demonstrate its effectiveness, we perform a case study where we eliminate all illegal backward edges and reducing forward control-flow edge targets by 99.98 % [16], 93.3 % [3, 50], 76.4 % [23], and 50.4 % [43] compared to existing kernel CFI schemes.

We evaluate HEK-CFI's security, demonstrating strong security guarantees against control-flow hijacking attacks. We run our proof-of-concept on Ubuntu 22.04.1 LTS on a recent Intel Alder Lake processor supporting Intel CET SHSTK. We observe a performance overhead of $12.3 \pm 1.5\%$ for the LMbench [49] micro benchmarks. For macro benchmarks, the performance overhead is $1.85 \pm 1.02\%$ for Phoronix Test Suite [53] and $0.17 \pm 0.23\%$ for SPEC CPU 2017 [15]. In summary, HEK-CFI is the first solution to provide protection for both system events and return addresses. It also generically reduces forward control-flow targets and performance overhead compared to existing CFI schemes. As a result, HEK-CFI improves kernel security to protect against control-flow hijacking attacks.

The main contributions of this work are:

(1) Kernel control-data integrity: We provide kernel control-data integrity, including a secure approach to protect system events and return addresses, establishing ourselves as the first kernel solution to safeguard both.

(2) HEK-CFI: We introduce HEK-CFI, a novel design that combines our kernel control-data integrity with signature-based CFI, ensuring robust protection for control-flow-related data.

(3) POC: We are the first to implement Intel CET SHSTK for the *supervisor* privilege level in the Linux kernel, allowing us to implement a HEK-CFI proof-of-concept as a compiler-assisted framework, both of which we provide open-source.

(4) Case study: We perform a case study to demonstrate the effectiveness of HEK-CFI in protecting forward and backward edges, as well as system events. We show that HEK-CFI provides enhanced efficacy compared to existing solutions.

(5) Evaluation: We evaluate HEK-CFI's security and performance, showing strong security guarantees with an overhead of 12.3 % and 1.85 % for micro and macro benchmarks.

Outline

Section 2 provides background. In Section 3, we present a systematization of existing works. Section 4 introduces HEK-CFI. In Section 5, we implement our proof-of-concept, while Section 6 conducts a case study. Section 7 shows the strong security guarantees. In Section 8, we evaluate the performance overhead. Section 9 discusses related work. Lastly, Section 10 concludes our work.

2. Background

This section provides background on control-flow hijacking attacks. At its core, a control-flow hijacking attack comprises two stages. First, an attacker exploits a vulnerability to obtain a Control-Flow Hijacking Primitive (CFHP) [66] allowing them to deviate from the legal Control-Flow Graph (CFG), e.g., memory safety vulnerability to overwrite a function pointer. Second, the attacker utilizes the CFHP to redirect the control flow to an attacker-controlled instruction sequence performing attacker-controlled execution.

CFI. Control-Flow Integrity (CFI) [1] has been established as a state-of-the-art mitigation against these attacks by restricting the control flow to the CFG. However, since complete CFI induces prohibitive runtime overhead, existing CFI schemes restrict the control flow to an approximated CFG. This approximation is achieved through various methods, including static determination (e.g., signature-based [21]) or dynamic techniques (e.g., ARM's Pointer Authentication (PA) [44] or by ensuring integrity of code pointers [39]).

Intel CET. Intel's Control-flow Enforcement Technology (CET) [30] is a recent hardware extension aiming to mitigate scenarios of hijacking attacks. CET consists of the Indirect-Branch Target (IBT) and a SHadow STacK (SHSTK) feature, where CET supports a user and supervisor SHSTK. The IBT feature introduced the `endbr` instruction as a landing pad for indirect branches. When the control flow of an indirect branch is redirected to any other instruction, the hardware raises a control-protection exception. In CET, a shadow stack protects the return path from being maliciously corrupted [60]. On a function call, the hardware pushes the return address onto both the data and the shadow stack. On a function

return, both addresses are compared, and a mismatch causes a control-protection exception. Additionally, the hardware pushes sensitive registers (i.e., `rip`, `cs`, and `ssp`) to the supervisor shadow stack on exceptions and interrupts, which are validated on an `iret` instruction. To protect the shadow stack from attackers, it is write-protected using the dedicated page permissions setting dirty and non-writable.

CETIS. CETIS [68] is a user space intra-process memory isolation scheme that uses Intel CET SHSTK's write-protected shadow pages to create a global safe area. This safe area ensures that adversaries cannot tamper with code pointers. By providing write protection for such data, CETIS enhances the efficiency and practicality of Code Pointer Integrity (CPI) [39] on x86_64 systems.

3. Threat Model and Systematization

In this section, we first present the threat model and then explore the variety of attack vectors used to obtain a CFHP in the Linux kernel. Additionally, we demonstrate that existing CFI-based mitigations do not fully prevent these CFHPs from violating control-flow restrictions, allowing them to bypass the applied defenses. To illustrate these limitations, we provide example exploits in Appendix 12 and an end-to-end attack exploiting CVE-2019-2215 in Appendix 13.

3.1. Threat Model

We assume that an attacker can arbitrarily execute code in user space. Moreover, the kernel contains a vulnerability that can be exploited to obtain an arbitrary memory read-and-write primitive. This primitive is accessible through the user space without violating the kernel's control-flow integrity. We assume that kernel defense mechanisms are enabled, i.e., the `W^X` [19], SMEP, SMAP [14], and page-table protection [17, 56]. With these mitigations enabled, kernel sections cannot be both writable and executable, the kernel cannot execute or access user space memory, and page tables cannot be manipulated [45]. Our threat model aligns with the threat model of existing kernel CFI-based mitigations [18, 23, 43, 64].

Attack goal. We assume that the primary goal of an attacker is corrupting kernel control-flow-related data discussed in Section 3.2 to hijack the kernel's control-flow.

Out of scope. Data-oriented attacks are another class of attacks that need to be addressed. This work focuses on mitigating control-flow hijacking attacks, while other orthogonal defenses [47, 55, 63] are necessary to address data-oriented attacks. While we acknowledge the existence of side-channel [8, 26, 45, 71], microarchitectural [27, 36], and software fault injection [12, 58] attacks, as well as malicious operating systems, these are out of the scope.

3.2. Attack Vectors

In the Linux kernel, various control-flow-related data (we refer as control data) can be exploited to obtain a CFHP.

Function pointers. Function pointers in writable sections are a well-known security concern [13]. The Linux kernel frequently uses function pointers for various tasks, e.g., calling device functions.

Operation table pointers. To reduce the attack surface of overwriting function pointers, the Linux kernel stores multiple function pointers in operation tables mapped as read-only [13]. However, since the operation table pointers are stored in writable sections, an attacker can tamper with the table pointers instead of function pointers [18, 32, 57]. For instance, each `inode` object has an operation table pointer `i_op` pointing to a read-only `inode_operations` struct containing function pointers for `inode` interaction. The attacker can obtain a CFHP by overwriting the table's pointer with a previously crafted `inode_operations` table.

Return addresses. Another way to redirect the control flow is by manipulating the return address stored on the stack. On function return, the kernel interprets the tampered return address as the execution path for the resumption.

Thread state. The thread state refers to the set of registers that store the runtime information about its thread, e.g., `rip` and `rsp` on x86_64, `pc` and `sp` on arm64, as well as general-purpose registers. During system events [6], i.e., context switch, interrupt, exception, and syscall, the current thread state is stored in memory, ensuring that it can be restored later when the thread resumes execution. This allows the kernel to switch

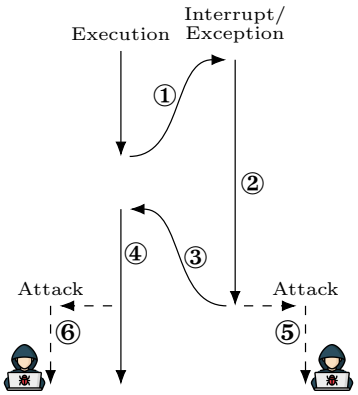


Figure 6.1: Interrupt or exception disrupts execution.

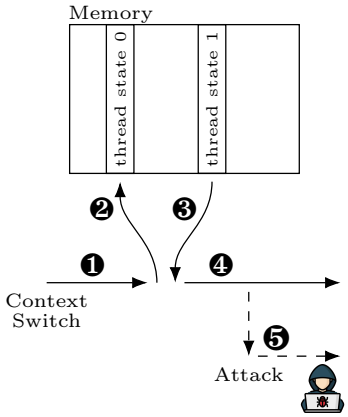


Figure 6.2: A context switch from thread 0 to thread 1.

between threads and handle event requests. Since these memory locations are writable, an attacker can manipulate them. If the state is restored from the tampered memory, the thread continues execution based on the tampered state, hijacking the kernel's control flow.

Figure 6.1 shows the system events interrupt and exception, which disrupt the thread's execution and store its state in memory ①. Consequently, the kernel's control flow is legally redirected to handle the invoked request ②. The thread state is restored upon completion ③ to continue its execution ④. By tampering with the stored state, an attacker can perform two attack scenarios. First, they manipulate stored registers, e.g., `rip` for x86_64 and `pc` for arm64, which are interpreted by the return, i.e., `iret` for x86_64 and `eret` for arm64, as the continuing execution ⑤. Second, they tamper with register values later used to redirect the control flow ⑥, e.g., `rax` if the execution performs `call *rax`. We provide motivational exploits in Appendix 12.1. In addition, security researchers at Google Project Zero [33] have exploited CVE-2022-42703 to obtain an uncontrolled arbitrary write, allowing them to corrupt thread state on interrupts and thus hijack control flow.

Another state-changing event is the context switch, where we refer to the Linux kernel design of a context switch: When a thread performs a context switch, it calls into the scheduler to select a new thread to run next, switches the memory descriptor, jumps to the `switch_to` function, and performs a cleanup of the previous thread. The `switch_to` function

Table 6.1: Systematization of existing kernel mitigations.

Mitigations	Attack Vector			
	Thread state	Return addresses	Operation table pointers	Function pointers
Ge et al. [23]	○	○	■	●
kCFI [3]	○	○	□	○
Fine-CFI [43]	○	○	■	●
PATTER [70]	○	●	◎	●
Camouflage [18]	○	●	●	○
PAL [64]	○	●	◎	●
FineIBT [50]	○	○	□	○
KCoFI [16]	●	○	□	○
Intel CET SHSTK [30]	○	●	○	○
CPI [39] + CETIS [68]	○	●	◎	●
HEK-CFI	●	●	●	●

● Protection ○ Insufficient protection
■ Implicit protection □ Implicit insufficient protection
◎ Does not protect but can be extended.

stores the current thread state in memory and then loads the stored state from the next thread. Figure 6.2 illustrates the storing and restoring of the state on a context switch ❶-❹. Since the state is stored in writable sections, e.g., the thread's stack frame, an attacker can manipulate it, and gain control of the registers when the state is restored. Taking control of these registers results in hijacking the control flow ❺. For instance, by tampering with callee-saved registers that store a function pointer, the attacker hijacks the control flow on the indirect branch instruction to its tampered register value. We provide motivational exploits in Appendix 12.2 and an end-to-end exploit in Appendix 13.

3.3. Systematization of Existing Works

We investigate the limitations of existing kernel CFI-based mitigations [3, 16, 18, 23, 43, 50, 64, 70] in protecting against kernel control-flow hijacking attacks, where we refer to Section 9 for detailed information. We include Intel CET SHSTK [30] and CPI [39] combined with CETIS [68] to highlight their limitations in protecting specific attack vectors. To illustrate our

findings, we use a classification scheme in Table 6.1. Mitigations marked with ● provide protection for the attack vector, while those marked with ○ provide no protection or can be bypassed by attacks we present in the following. For mitigations that implicitly protect the attack vector or implicitly protect it insufficiently, we use ■ or □, respectively.

Thread state. Various kernel mitigations either do not (i.e., Camouflage [18], FineIBT [50], kCFI [3], and PATTER [70]) or inadequately (i.e., PAL [64], Fine-CFI [43], and the proposal from Ge et al. [23]) protect the stored thread state. As a result, attack scenarios, such as ❺ from Figure 6.2, and ⑤ and ⑥ from Figure 6.1, can redirect the control flow and, thereby, bypass control-flow restrictions (see Appendices 12.1 and 12.2 for exploitation examples).

Intel CET SHSTK for supervisor [30] also fails to adequately protect the thread state, as they do not mitigate attack scenarios ⑥ and ❺ from Figure 6.1 and Figure 6.2. Moreover, since the shadow stack pointer is part of the thread state, Intel CET SHSTK can be compromised using ❺ as well. CPI [39] combined with CETIS [68] suffers from a similar issue, as it lacks a low-level protection primitive, like protected local storage, to safeguard the thread state.

Return addresses. Kernel mitigations (i.e., kCFI [3], KCoFI [16], Fine-CFI [43], and the proposal from Ge et al. [23]) that solely rely on static analysis to protect backward control-flow edges leave the system vulnerable to the notorious Control-Flow Bending (CFB) [9] attack. CFB involves exploiting dispatcher functions to corrupt the return address using malicious arguments, bypassing the applied CFI protection. As emphasized by Carlini et al. [9] a shadow stack is necessary to fully protect return addresses and mitigate CFB. Besides shadow stack, Lilijestrand et al. [44] emphasized that ARM's Pointer Authentication (PA) [4] could also enhance protection against CFB attacks with dynamic runtime information, such as the current stack pointer, to validate return addresses.

Operation table pointers. Since PA-based kernel mitigations (i.e., PAL [64] and PATTER [70]) do not protect operation table pointers they are susceptible to pointer-to-pointer attacks, where an attacker corrupts operation table pointers instead of function pointers. However, their design can be extended to also protect operation table pointers. Mitigations that provide static control-flow integrity (marked with ■ or □ in Table 6.1) implicitly protect operation table pointers on indirect branches. Hence, if

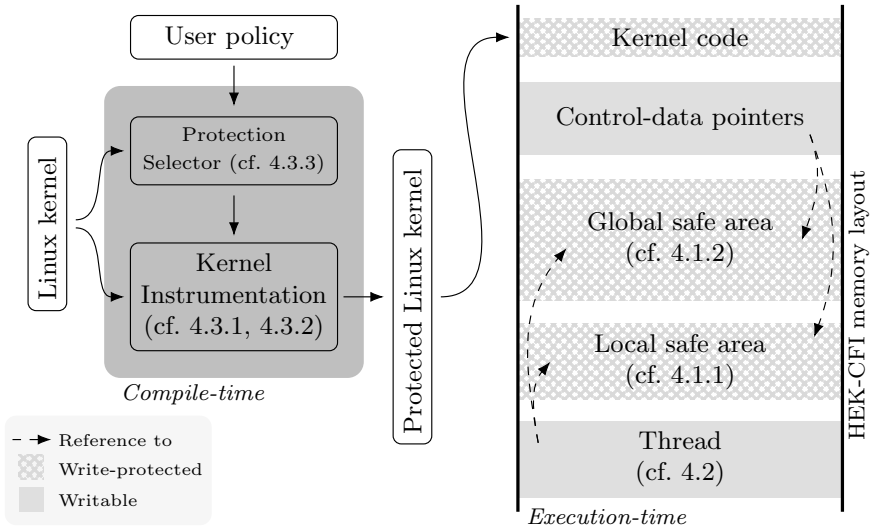


Figure 6.3: HEK-CFI instruments the kernel to perform run-time validation checks ensuring protection for control data.

the function pointers are protected, the operation table pointers are also protected.

Function pointers. Mitigations that provide coarse-grained (i.e., KCoFI [16]) or signature-based (i.e., Fine-CFI [22] and kCFI [3]) protection for forward control-flow edges offer weak security guarantees. The large set of targets matching the function signature in the kernel space enables privilege escalation while not violating the over-approximated CFG, as demonstrated in Appendix 12.3. Moreover, Camouflage [18] only protects a selected set of function pointers, leaving unprotected vulnerable for privilege escalation.

Summary. Existing kernel mitigations exhibit limitations in providing protection for control data, particularly the thread state during system events and return addresses. As a consequence, attackers can exploit these limitations to bypass the applied CFI-based countermeasure. This results in a gap in kernel security.

4. Design

We introduce Hardware-Enforced Kernel Control-Flow Integrity (HEK-CFI) which consists of three key mechanisms. Figure 6.3 depicts a high-level overview. First, HEK-CFI provides kernel control-data integrity (see Section 4.1) by retrofitting write-protected pages from Intel CET SHSTK for the *supervisor*. This retrofitting enables write-protected local and global safe areas within the kernel where control data is securely stored. Notably, these safe areas extend well beyond the original purpose of Intel CET SHSTK. Second, HEK-CFI utilizes our control-data integrity to protect the thread state (see Section 4.2) during all state-changing system events, i.e., interrupt, exception, syscall, and context switch, as well as protect the SHSTK state. This allows HEK-CFI to mitigate our motivational exploit examples and Google Project Zero's thread state exploit [33]. With these two key mechanisms, we are the first to provide comprehensive protection for both system events and return addresses. Third, HEK-CFI combines our control-data integrity with signature-based CFI to protect (see Section 4.3) forward control-flow edges, particularly control data, i.e., function pointers and operation table pointers. While signature CFI efficiently protects function pointers with rare signatures, control-data integrity offers full protection for any control data, albeit at a potentially higher performance overhead. To optimize the trade-off, it automatically selects the optimal scheme for each control data based on a user policy as input.

While the real contribution of HEK-CFI lies in the efficient protection of control data (including thread state), the third mechanism demonstrates the practicality of protecting forward edges as well.

4.1. Kernel Control-Data Integrity

Its main concept is to store control data (outlined in Section 3.2) in a hardware-enforced write-protected safe area, preventing any malicious tampering attempt. To access the control data legally, the safe area needs to be designed differently depending on the context, such as local or global. For this reason, HEK-CFI includes a per-thread local and a global safe area storing and protecting the local and global control data, respectively.

Write protection. Our control-data integrity scheme relies on the security of the write-protected safe areas. To achieve this protection, all

```
1 struct ptls {
2   u64 tos; /* top of stack */
3   u64 data[]; /* data array */
4 };
5 #define wr_ptls(tos, d) \
6   asm("wrssq %0,(%1)"::"r"(d),"r"(tos))
7 #define rd_ptls(ptls) \
8   asm("rdsspq %0\n" \
9     "andq $~(LSA_SZ-1),%0":"=r"(ptls))
10
11 void ptls_init(void) {
12   struct ptls *ptls;
13   rd_ptls(ptls);
14   wr_ptls(&ptls->tos, (u64)ptls->data);
15 }
16 void ptls_push(u64 data) {
17   struct ptls *ptls;
18   rd_ptls(ptls);
19   wr_ptls(ptls->tos, data);
20   wr_ptls(&ptls->tos, ptls->tos+8);
21 }
22 u64 ptls_pop(void) {
23   struct ptls *ptls;
24   rd_ptls(ptls);
25   wr_ptls(&ptls->tos, ptls->tos-8);
26   return *ptls->tos;
27 }
```

Listing 6.1: PTLS's provided routines.

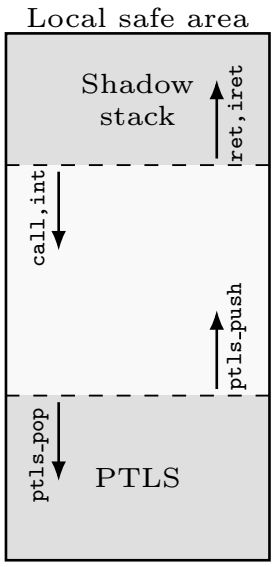


Figure 6.4: Memory layout of the local safe area for each thread, where Intel CET SHSTK implicitly resides on the top and PTLS on the bottom.

safe area pages are marked as shadow pages, retrofitting the approach followed by CETIS [68] to the kernel. This enforces Intel CET SHSTK to write-protect them. To legally write to the safe areas, HEK-CFI utilizes the `wrssq` instruction introduced by Intel CET, which permits writes to shadow pages. If a memory write operation to a shadow page is propagated by a non-shadow stack instruction, e.g., `movq`, it causes a control-protection exception raised by Intel CET. There is also no `wrssq`-gadget allowing an attacker to illegally write to shadow pages as we later discuss in Section 7. To conclude, with this approach, it not possible to illegally manipulate control data stored in the safe areas using the arbitrary write primitive.

4.1.1. Local Safe Area

HEK-CFI introduces a local safe area for each thread in kernel space. The local safe area consists of a shadow stack via Intel CET and a

Protected Thread Local Storage (PTLS), as illustrated in Figure 6.4. Intel's hardware feature CET SHSTK utilizes the shadow stack to implicitly push and pop specific control data onto the shadow stack, as explained in Section 2. However, since Intel CET does not provide explicit push or pop operations [29], HEK-CFI introduces the software-based approach PTLS, which is an efficient and secure local storage. The PTLS is located at the bottom of the local safe area and provides `ptls_push` and `ptls_pop` routines corresponding to a safe push and pop operation. Hence, PTLS provides strong protection for locally accessible control data.

Listing 6.1 illustrates the `ptls` struct (PTLS) and the associated initializing, pushing, and poping routines. At its core, all of these routines use the macros `rd_ptls` (Line 7) and `wr_ptls` (Line 5) to obtain the `ptls` struct's location and to write to the underlying shadow page memory of the `ptls` struct, respectively. The macro `rd_ptls` does so by performing a logical AND operation of the read shadow stack pointer (Line 8) with `(LSA_SZ-1)`, where `rdsspq` reads the current shadow stack pointer, and `LSA_SZ` represents the local safe area size. The macro `wr_ptls` executes the `wrssq` instruction to write an 8 byte word to the top of the `ptls`. The `ptls_init` routine initializes the `ptls` struct by obtaining its current location with `rd_ptls`. Then, `ptls_init` sets the top of stack member variable `ptls->tos` to the `&ptls->data[]`, indicating that the `ptls` is empty and is ready for the first push operation. The other two routines, `ptls_push` and `ptls_pop`, perform push and pop operations to or from the `ptls`, respectively.

4.1.2. Global Safe Area

HEK-CFI presents a global safe area for control data that are accessible in a global context. These globally accessible control data store an index referencing the global safe area where the actual data is stored. We design our global safe area as a one-to-one mapping, meaning that each globally accessible control data has its own safe storage within the global safe area.

HEK-CFI provides allocation and deallocation routines to allocate and deallocate a global safe storage for the control data. To mitigate forgery attempts, we uniquely bind the control data to its safe storage by storing its address along with the actual protected data within the safe storage on allocation. On deallocation, we unbind the control data with the safe storage and mark the safe storage as free. Figure 6.5 exemplifies a globally

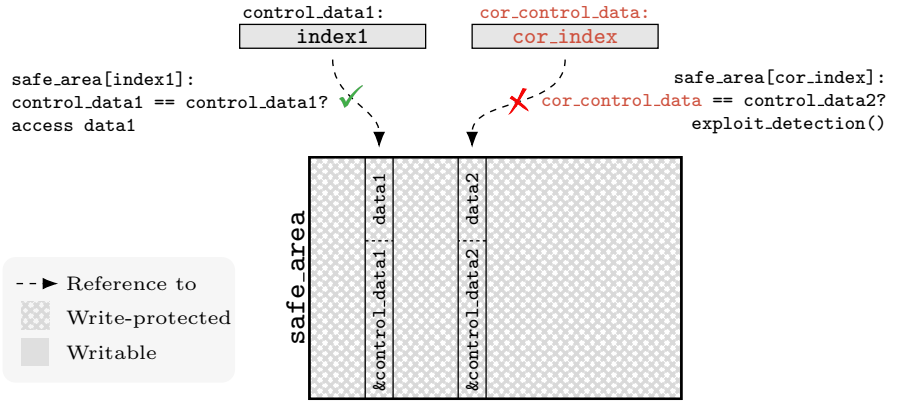


Figure 6.5: Two accesses from a control data stored on the global `safe_area`. HEK-CFI validates that the address of the accessed control data matches the stored address referenced by the index. Since the validation of `control_data1` succeeds, `data1` access is granted. Contrary, the validation of the corrupted `cor_control_data` fails, detecting an exploit attempt.

accessible control data with `control_data1`, storing an `index1` index that references the safe storage (`safe_area[index1]`) within the global safe area. This storage comprises the control data address `&control_data1` and the actual protected data `data1`.

HEK-CFI provides read and write routines to access the global protected data. It ensures protection against malicious access by verifying the control data's integrity before the access. This verification involves checking whether the accessed control data's address matches the stored address of the referenced safe storage. Access is granted on match; otherwise, it is considered an exploitation attempt. We illustrate both cases in Figure 6.5, where the validation check for the valid control data `control_data1` succeeds, while it fails for the corrupted control data `cor_control_data`.

4.2. Thread State Protection

Compared to user space programs, the kernel processes synchronous (i.e., exceptions and syscalls) and asynchronous (i.e., interrupts) system events which store the thread state in memory and handles invoked execution request (see Sections 4.2.1 and 4.2.2). Furthermore, at each context switch (see Section 4.2.3), the kernel stores the thread state in memory while

restoring the state of the next thread to execute. By not fully protecting the stored state, an attacker can perform attack scenarios described in Section 3.2, as demonstrated in the examples given in Appendix 12 and the end-to-end attack exploiting CVE-2019-2215 in Appendix 13. To mitigate these scenarios, HEK-CFI uses our kernel control-data integrity to effectively protect the stored thread state. More precisely, HEK-CFI protects the thread state for interrupts, syscalls, and exceptions by storing it within the local safe area and for the context switch within both the global and local safe area.

4.2.1. Interrupt and exception

The hardware pushes `ss`, `rsp`, `rflags`, `cs`, and `rip` to the current stack or value specified in the Interrupt Stack Table (IST) on kernel entrance and re-entrance. We refer to the current stack and the value specified in the IST as data stack. The kernel then pushes the general-purpose registers (except `rsp`) to the data stack. With Intel CET SHSTK enabled, the hardware pushes `cs`, `rip`, and `ssp` to the shadow stack atomically to the data stack push. On `iret` instructions, the hardware validates that `cs` and `rip` are equal to the ones stored on the shadow stack, where a mismatch causes a control-protection exception.

Protecting `cs` and `rip` is insufficient to ensure protection as an attacker may tamper general-purpose registers stored on the data stack, illustrated in attack scenario ⑥ in Figure 6.1. To mitigate this, HEK-CFI stores the register values within the PTLS on interrupt and exception entrances before pushing them to the data stack. On interrupt and exception exits, the registers are popped from the data stack and then compared with those stored in the PTLS. HEK-CFI interprets a mismatch as an exploitation attempt.

To mitigate potential TOCTTOU attacks, HEK-CFI never stores the registers to unprotected memory during state storing (① in Figure 6.1). HEK-CFI protects all register values, as shown in Listings 6.2 and 6.3. On the request entrance routine `rq_entry`, HEK-CFI first temporarily stores `r15` to a per-CPU storage (Line 3), also write-protected with Intel CET SHSTK. Temporarily storing `r15` is essential because HEK-CFI requires one register for the upcoming execution. HEK-CFI then interprets the bottom of the current local safe area as `ptls` (Lines 5-6) and loads `ptls-`

```
1 rq_entry:
2  /* safe store r15 */
3  wrssq  r15, R15(gs)
4  /* r15 = ptls->tos */
5  rdsspq r15
6  andq   $~(LSA_SZ-1), r15
7  movq   r15, (r15)
8  /* store all regs to ptls */
9  wrssq  r14, R14(r15)
10 wrssq  r13, R13(r15)
11 ...
12 wrssq  rdi, RDI(r15)
13 /* store r15 to ptls */
14 wrssq  r14, R14(gs)
15 movq   r15, r14
16 movq   R15(gs), r15
17 wrssq  r15, R15(r14)
18 movq   R14(gs), r14
```

Listing 6.2: Safe stores all registers to the current `ptls`.

```
1 rq_exit:
2  /* safe store r15 */
3  wrssq  r15, R15(gs)
4  /* r15 = ptls->tos */
5  rdsspq r15
6  andq   $~(LSA_SZ-1), r15
7  movq   r15, (r15)
8  /* validate all regs */
9  cmpq   r14, R14(r15)
10 jne    .fault
11 ...
12 cmpq   rdi, RDI(r15)
13 jne    .fault
14 /* validate r15 */
15 movq   r14, R14(gs)
16 movq   r15, r14
17 movq   R15(gs), r15
18 cmpq   r15, R15(r14)
19 jne    .fault
20 movq   R14(gs), r14
```

Listing 6.3: Validates all registers to be equal to the stored ones on the `ptls`.

`>tos` to `r15` in Line 7. Between Lines 9-12 all registers are stored to the `ptls`. Lastly, HEK-CFI protects register `r15` (Lines 14-18).

On request exit `rq_exit`, HEK-CFI performs a validation process that requires `r15` for the upcoming execution. Between Lines 9-13, HEK-CFI compares all registers with the protected register values stored within the `ptls`, jumping to `.fault` on a validation failure. We relax the constraints on interrupts and exceptions from user space: We guarantee to return to the exact location in user space but do not protect general-purpose registers. Hence, we apply `rq_entry`/`rq_exit` only if the thread was disrupted during kernel-space execution. Crucially, our relaxation is stricter than comparable CFI-based mitigations [23, 43, 64].

4.2.2. Fast syscall

The `syscall` instruction invokes a fast syscall to request kernel-space execution with supervisor privileges. The hardware saves the current `rip` to `rcx` and `rflags` to `r11` on this instruction. Next, it loads the `rip` from `MSR_IA32_LSTAR`, indicating the syscall entry location. On entry, the

kernel software stores both register values, `rcx` and `r11`, on the data stack, while on completion, it restores both values and executes `sysret`. This instruction returns to user space by loading `rip` from `rcx`, `rflags` from `r11`, and user `cs` and `ss` from `MSR_IA32_STAR`.

To protect `rcx` and `r11` from being tampered with, HEK-CFI also stores both values within our provided PTLS, on kernel entrance. During kernel execution, when these user registers are legally modified, HEK-CFI also modifies the protected one. On completion, HEK-CFI validates that `rcx` and `r11` have not been tampered with.

In rare cases, the Linux kernel returns from a fast syscall to user space with `iret` instead `sysret`. HEK-CFI stores, for these cases, the `USER_CS` and `rcx` which represents `cs` and `rip`, to the shadow stack. On `iret` Intel CET SHSTK compares both, `cs` and `rip`, from the data and shadow stack, where a mismatch raises an exception. Crucially, HEK-CFI never trusts any value stored in unprotected memory, as `USER_CS` is a constant and `rcx` was protected within PTLS. Since an attacker cannot corrupt either register, HEK-CFI protects the fast syscall event from being exploited.

4.2.3. Context switch

When a thread performs a context switch, it calls into the scheduler that selects a new thread to run next. Since the Linux kernel does not have a dedicated scheduler, the current thread switches the memory descriptor (including `cr3`) with the new descriptor and jumps to `switch_to`. This function stores the current and restores the next stack pointer, callee-saved registers, `fs`/`gs` registers (if required), and additional non-general-purpose registers, e.g., for debugging. Storing and restoring the instruction pointer is not needed as it was implicitly stored on the stack when the context switch function was called and is restored on return.

HEK-CFI also mitigates the attack scenario outlined ❺ from Figure 6.2 by protecting the shadow stack pointer with control-data integrity within the global safe area. Hence, the attacker cannot forge the shadow stack pointer. Since the control data of the data stack must match the control data on the shadow stack on every `ret` and `iret`, and the shadow stack pointer's integrity is ensured, corrupting the data stack pointer cannot be exploited.

In addition to the shadow stack pointer, HEK-CFI protects callee-saved registers as well as `fs` and `gs` within our PTLS upon a context switch event. HEK-CFI does not require explicit protection for instruction pointers on a context switch because Intel CET SHSTK implicitly protects the stored instruction pointer as a return address.

4.3. Control-Flow Integrity

In this section, we explain how HEK-CFI combines kernel control-data integrity with function signature CFI to protect the thread state, return addresses, and control data pointers, i.e., operation table and function pointers. Function signature CFI (see Section 4.3.2) provides efficient protection for function pointers with rare signatures. In contrast, control-data integrity (see Section 4.3.1) offers full protection for any control data, albeit with a potentially higher performance overhead. To optimize the trade-off, we present the control-data protection selector (see Section 4.3.3), which automatically selects what to protect with signature CFI and what with control-data integrity based on a user policy as input. This way, we achieve strong security without compromising performance.

4.3.1. Safe Area Usage

To protect control data pointers with the kernel control-data integrity approach, HEK-CFI instruments the kernel as follows: When control data pointers are generated, HEK-CFI allocates a safe storage and binds it to the control data. On control data destruction, the corresponding safe storage is unbound and deallocated. Depending on the context of the control data (i.e., global or local), HEK-CFI utilizes either the global or local safe area as safe storage. For the global safe area, HEK-CFI uses the provided allocation and deallocation routines directly. For the local safe area, HEK-CFI pushes a local safe storage to the PTLS, consisting of the actual protected data and the address of the local control data pointer, to uniquely bind the local safe storage to the control data. On deallocation, HEK-CFI pops the local safe storage from the PTLS.

When accessing control data pointers, HEK-CFI validates whether the index stored in the control data has been tampered with. For the global safe area, HEK-CFI uses the provided read and write routines directly, as shown in Figure 6.5. For the local safe area, HEK-CFI uses read and

write routines similar to the global ones, but they reference the PTLS. We illustrate the instrumentation in Appendix 14.

Since HEK-CFI uses a one-to-one mapping to bind the control data pointers to the safe storage uniquely, functions such as `memcpy` may cause false positive exploitation attempts, as the copied stored index mismatches with its referenced address. To prevent false positives, we provide an instrumentation routine performing `realloc`. We carefully designed this routine to only copy the control data if the old safe storage is validated. Since the old safe storage was validated, this routine cannot be used to forge a safe storage.

4.3.2. Function-Signature Control-Flow Integrity

HEK-CFI provides control-flow transfer restriction with a function signature granularity by applying FineIBT [22] to the kernel. FineIBT leverages Intel IBT for coarse-grained control-flow integrity and builds a software-based control-flow restriction with function signature granularity on top of it. It stores the hash of a function pointer's signature into a register before redirection and validates the hash on function entry. A hash mismatch is interpreted as an exploitation attempt.

4.3.3. Control-Data Protection Selector

How much performance overhead is acceptable and how much is prohibitive depends on the use case. For high-security context, i.e., protecting all control data with control-data integrity, an elevated performance overhead may be acceptable. On the other hand, high-efficiency systems hardly accept any performance overhead. Since we envision HEK-CFI being suitable for all use cases, we provide a user policy that allows to choose the desired CFI precision level and, hence, overhead.

To accomplish this, our control-data protection selector analyzes the Linux kernel to identify control data pointers, i.e., function and operation table pointers, that require protection with control-data integrity based on the user policy. Dynamically allocated and global control data pointers are stored in the global safe area (see Section 4.1.2), while local control data pointers are stored in the PTLS (see Section 4.1.1). Regardless of the user policy, protection for the thread state and return addresses is always enabled.

User policy. The user policy comprises the number of permitted control-flow targets with function signature granularity. Our selector analyzes the Linux kernel and determines the number of possible control-flow targets matching the signature for each control data pointer. For function pointers, if the determined number exceeds the permitted control-flow targets, the selector annotates to protect the pointer with control-data integrity. For operation table pointers, the selector annotates to protect the pointer if one of its containing function pointers exceeds the permitted targets.

5. Implementation

In this section, we present the proof-of-concept implementation of HEK-CFI. We extend the Linux kernel and implement an LLVM pass [40] for instrumentation and the control-data protection selector using CodeQL [24] and Python. Our selector and instrumentation work automatically based on a user policy.

5.1. Linux Kernel

We first enhance the Linux kernel to support all instrumentation routines. Next, we integrate write protection for the safe areas enforced with CET SHSTK. Lastly, we provide thread state protection.

Kernel instrumentation. We implement thread-safe functions for all kernel instrumentation routines described in Sections 4.1.1, 4.1.2 and 4.3.1, except for the read-access routines, as our compiler extension directly inserts an instruction sequence for the read-access routines, as we explain in Section 5.3. Moreover, we implement a routine that handles all fault scenarios of HEK-CFI, where our proof-of-concept detects and prevents exploitation attempts.

Protection through shadow stack. At the time of writing, there was no Linux kernel patch for the *supervisor* shadow stack[1]. Therefore, we integrate shadow stack as follows: Intel CET SHSTK supports per-thread supervisor and per-CPU IST shadow stacks. The supervisor shadow stack is used with most executions, while the IST shadow stack is used when the system requires the IST stack. To enable shadow stack usage, we

[1]Xen hypervisor v4.15-unstable [11] has integrated Intel CET SHSTK for supervisor, but it significantly varies from the Linux kernel integration.

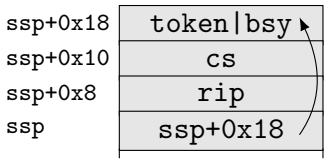


Figure 6.6: Required shadow stack layout for a valid `iret`.

first set the per-CPU IST shadow stacks (`MSR_INT_SSP_TAB`) to a table of shadow stacks and the supervisor shadow stack (`MSR_PL3_SSP`) to the per-thread shadow stack page. We then set bit `X86_CR4_CET` in the `cr4`, and `CET_SHSTK_EN | CET_WRSS_EN` in the `MSR_S_CET`. Next, we prepare all shadow pages for the `setssbsy` instruction accordingly [31], which sets the shadow stack pointer register to the value specified in the `MSR_PL0_SSP`. The `setssbsy` instruction requires the shadow stack to be not busy, while it marks the stack as busy. The busy flag is stored in the memory location `MSR_PL0_SSP` points to.

To mark a page as a shadow stack, its permissions must be dirty and non-writable. If a page is marked as shadow stack, only shadow stack write instructions, e.g., `wrssq`, are permitted to write this page. Otherwise, Intel CET raises a control-protection exception on a write operation propagated from a non-shadow stack instruction, e.g., `movq`. Moreover, if a shadow stack instruction writes to a non-shadow page, Intel CET also raises a control-protection exception.

Our proof-of-concept enables the shadow stack during kernel initializations before SMP is enabled, and only the `init_task` runs. Hence, the shadow stack is only set for the `init_task`. Enabling the shadow stack must occur in the early stage of kernel initialization, as our control-data integrity scheme relies on its security mechanism.

We extend the user thread creation routine to allocate one shadow stack page, mark its permission to dirty and non-writable, initialize the PTLS, and prepare the shadow stack for the `iret` to start execution in user space. Figure 6.6 depicts the required shadow stack layout to perform a valid `iret` [29], where `ssp` is the shadow stack pointer, `token` is the supervisor token, and `bsy` is the busy flag indicating whether the shadow stack is in use. On `iret`, the hardware validates that `rip` and `cs` are equal between data and shadow stack. If they are equal, the hardware sets the `rip` and `cs` accordingly. It then sets the shadow stack pointer to `ssp+0x18` and

resets the `bsy` flag on the user space transition. We also extend the kernel thread creation, which closely assembles the user thread creation.

Since each thread has its shadow stack, the shadow stack pointer has to be stored and restored on every context switch. Hence, we implement the context switch in assembly as described in Section 4.2.3. When restoring the next shadow stack pointer, we store a non-busy token temporarily in the next shadow stack, which is used to switch the shadow stack. The switch occurs with the `setssbsy` instruction. Afterward, we reverse the temporary storing of the token.

Protection through IBT. We use the inofficial FineIBT patch [22, 50] for the control-flow restriction with signature granularity.

Interrupts and exceptions. We implement a safe storing routine for all general-purpose registers on interrupt and exception entries, as illustrated in Listing 6.2. Crucially, `rq_entry` is executed before the Linux kernel pushes the registers to the data stack via `PUSH_ALL_REGS`. Since `rq_entry` requires the register `r15` for the upcoming execution, HEK-CFI temporarily stores it on a per-CPU page in the `per_cpu` section. The per-CPU page's permissions are set as a shadow stack page. To support nested interrupts, HEK-CFI increases the `ptls->tos` by `sizeof(struct regs)` after both `rq_entry` and `PUSH_ALL_REGS`. We implement the validation routine `rq_exit` on interrupt and exception exits, shown in Listing 6.3.

Fast syscalls. Intel CET resets the supervisor shadow stack pointer on a privilege level change from user to kernel space via a fast syscall (i.e., `syscall` instruction) [31]. Hence, we extend the kernel entrance for a fast syscall to set the shadow stack accordingly. On rare occasions where the kernel returns via the `iret` instruction instead of `sysret` to the user space, HEK-CFI prepares the shadow stack to perform a valid `iret`, shown in Figure 6.6.

5.2. Control-Data Protection Selector

Our control-data protection selector has two main components: A code analyzer and a parser. We use CodeQL [24] as our code analyzer and a Python script as our parser. CodeQL compiles the Linux kernel to create a database that stores essential meta information. We run our CodeQL queries using the database to retrieve relevant information for our mitigation. Then, our parser interprets the query results based on the

user policy and generates an output file containing all the information for kernel instrumentation.

Code analyzer. Our CodeQL queries first find all control data pointers, e.g., members within a struct or its containing struct or standalone pointers. The queries then determine the number of functions matching the function pointer's signature or any within an operation table. Each function pointer retrieves a score of matching functions, while each operation table pointer retrieves the highest number of its containing function pointers. The queries also determine all occurrences of control data pointers via allocation, deallocation, global variable, or local variable.

Parser. Our parser inputs the query results and the user policy (i.e., maximum permitted control-flow targets with signature granularity). It then filters out control data pointers with fewer targets than the user policy allowed and saves the remaining pointers to a file. This file represents all pointers protected with control-data integrity. At this point, a user can modify the file if necessary.

5.3. Compiler Extension

Our compiler extension (LLVM pass) inserts validation checks into the kernel, ensuring HEK-CFI's functionality.

Control-data integrity. To ensure control-data integrity, the LLVM pass first examines kernel code for operations such as allocations, deallocations, reallocation, reads, and writes of control data pointers, annotated in the file generated by our control-data protection selector. Subsequently, the pass modifies the code to include instrumentation routine accessing the control data, where Appendix 14 demonstrates the actual instrumentation. For allocation, deallocation, reallocation, and write access, the pass inserts function calls provided by our kernel extension. The code is modified for read access to perform a performance-trimmed instruction sequence executing the safe storage read-access. To protect global variables, HEK-CFI identifies all protected global control data pointers, whether standalone or within structs, and allocates their safe storage during the early stage of kernel initialization.

Similar to global variables in kernel code, global variables within modules must also be initialized so that protected control data pointers reference a

safe area. Hence, HEK-CFI identifies these global variables and inserts initialization routines on module insertion.

Signature CFI. We utilize the inofficial FineIBT patch [50] in our LLVM pass to ensure control-flow restriction with signature granularity. This patch instruments the caller and callee sites involved in each indirect forward-edge transfer.

6. Case Study

This section demonstrates HEK-CFI's effectiveness in reducing forward control-flow targets. HEK-CFI achieves this by protecting operation table pointers and function pointers with a common function signature from being overwritten through our kernel control-data integrity scheme (see Section 4.1). Additionally, it restricts the control-flow targets of function pointers with rare function signatures at the granularity of signatures. By reducing the permitted targets of control-data-integrity-protected pointers to 1 and limiting signature-restricted pointers to the set of functions matching the pointers' signature, HEK-CFI effectively lowers the overall average forward control-flow targets. This case study demonstrates that HEK-CFI achieves a forward target reduction of more than 50 % over comparable kernel CFI schemes [3, 16, 23, 43, 50] while having a lower performance overhead than these schemes, as we later evaluate in Section 8. Notably, HEK-CFI is also the first to protect return addresses and thread states comprehensively.

In the following, we describe how we compute HEK-CFI's target reduction with other kernel CFI schemes. We first discuss the *Average Indirect targets Allowed* (*AIA*) metric, which allows us to quantify CFI precision levels. We then determine the *AIA* for various CFI precision levels, including HEK-CFI. Finally, we demonstrate that HEK-CFI outperforms the security guarantees of forward control-flow target reduction compared to CFI schemes.

Quantify CFI precision level. Zhang et al. [72] proposed the metric *Average Indirect Reduction* (*AIR*) to evaluate the effectiveness of CFI-based mitigations. However, for large binaries, e.g., the Linux kernel, the *AIR* may be misleading [9, 23] because in these cases, an *AIR* of more than 99 % stills permits 10 k of control-flow targets for each indirect control-flow redirection. Subsequently, Ge et al. [23] proposed *AIA* as

Table 6.2: Function pointers protected with kernel control-data integrity (see Section 4.1).

	Total	**Within operation tables**[1]	**Plain**[2]
Total	6487	3033	3454
Protected	1662	1235	427

Stored in [1]read-only sections and [2]writable sections.

an improved evaluation metric to quantify the effectiveness of CFI-based mitigations.

$$AIA = \frac{1}{n} \sum_{i=1}^{n} |T_i| \tag{6.1}$$

Equation (6.1) shows the quantifier, where n is the number of indirect calls within the entire binary, and T is the set of permitted control-flow targets. We consider three cases for our CFI precision analysis: Coarse-grained (reachable function granularity) and fine-grained (function signature granularity) CFI policy, and our HEK-CFI.

Analysis. The setup of our case study is the Linux kernel v5.18 running with the configuration of Ubuntu 22.04.1 LTS. We perform a manual analysis revealing that the entire kernel code, including all modules, comprises 6487 function pointers (outlined in Table 6.2). 3454 of them are stored in writable sections and 3033 are stored within read-only operation tables, whereas the kernel has 300 operation table pointers. We determine that the kernel comprises a total of 140534 control-flow targets for coarse-grained CFI (without any unaligned `endbr64`). Next, we obtain information for each indirect call and the function pointer's permitted targets with signature granularity. Using these insides and Equation (6.1), we calculate a coarse-grained AIA_{cg} of 140534 and a fine-grained AIA_{fg} of 325.

Target reduction with HEK-CFI. We configure HEK-CFI via user policy to enforce an $AIA_{hek-cfi}$ that is lower than comparable kernel CFI schemes [3, 16, 23, 43, 50]. To achieve this, a developed tool (see Appendix 15) determines the user policy as 190 maximum permitted control-flow targets. With this user policy, HEK-CFI's control-data protection selector automatically determines that out of 3454 function pointers stored in writable sections, 427 exceed the permitted control-flow targets of 190. Hence, HEK-CFI protects these function pointers with control-data integrity, as outlined in Table 6.2. Moreover, among the 300 operation

Table 6.3: Permitted control-flow target reduction of HEK-CFI over state-of-the-art mitigations.

	AIA -	***AIR*** %	**HEK-CFI's improvement**[1] %
KCoFI [16]	140534	-	99.98
FineIBT [50], kCFI [3]	325	-	93.3
Ge et al. [23]	92	-	76.4
Fine-CFI [43]	-[2]	99.999740	50.4
HEK-CFI	22	99.999871	-

[1] HEK-CFI's control-flow target reduction over the other mitigation.

[2] Not enough information provided to compute the *AIA*.

table pointers, 88 contain at least one function pointer exceeding 190 targets. Consequently, HEK-CFI ensures the integrity of these 88 operation table pointers. Since the operation tables referenced by the 88 protected pointers are read-only, their containing 1235 function pointers are implicitly protected from manipulation. Overall, our case study protects 1662 function pointers reducing their control-flow target to 1.

$$AIA_{hek-cfi} = \frac{1}{n}\left(\sum_{i=1}^{p} 1 + \sum_{i=p+1}^{n} |T_i|\right) \tag{6.2}$$

We adapt Equation (6.1) to Equation (6.2), where p is the number of protected function pointers, i.e., 1662. We then compute $AIA_{hek-cfi}$, resulting in 22 average permitted control-flow targets. To evaluate HEK-CFI's effectiveness, we compute the improvement over coarse- and fine-grained CFI, showing a permitted target reduction of 99.98 % over coarse- and 93.3 % over fine-grained CFI.

Comparison to existing CFI schemes. As described by Ge et al. [23], directly comparing CFI precision metrics is inaccurate for several reasons, such as varying kernels and kernel configurations. Hence, for the following comparison, we relatively compare various designs [3, 16, 23, 43, 50] as if their mitigation would have protected our kernel binary, with the results shown in Table 6.3.

KCoFI provides coarse-grained CFI, resulting in an AIA_{KCoFI} of 140534. Due to our $AIA_{hek-cfi}$ of 22, we reduce targets by 99.98 %. In both fine-grained mitigations, FineIBT and kCFI, average permitted targets (AIA_{fg}) are 325, while our mitigation reduces these targets by 93.3 %. Ge

et al. claimed that their proposal eliminates 71.8 % of signature-based CFI schemes. We assess the CFI precision level of their proposal by reducing the AIA_{fg} by 71.8 %, resulting in an AIA_{ge} of 92. With an $AIA_{hek-cfi}$ of 22, we improve by 76.4 %.

Since Fine-CFI only provides AIR as precision level, we compute the AIR for HEK-CFI and Fine-CFI for our kernel binary.

$$AIR_{hek-cfi} = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \frac{|T_i|}{S}\right) \tag{6.3}$$

$$AIR_{fine-cfi} = 1 - \frac{|T|}{S \cdot n} \tag{6.4}$$

We use Equation (6.3) [72] to compute HEK-CFI's $AIR_{hek-cfi}$ and their adapted Equation (6.4) [43] for Fine-CFI's $AIR_{fine-cfi}$, where T is the set of permitted control-flow targets, n is the number of indirect calls, and S is the kernel binary size. The results are 99.999 871 % for $AIR_{hek-cfi}$ and 99.999 74 % for $AIR_{fine-cfi}$, showing HEK-CFI's permitted target reduction of 50.4 %.

PAL [64] provides too little information to perform a reasonable comparison. Since we neither have any information about their AIA nor AIR, a direct comparison with our case study is not possible.

Manual efforts. In rare cases, LLVM's frontend may not store variable type information, which can cause our implemented LLVM pass to miss control data pointer uses. We manually inserted validations into the kernel code to address these rare false negatives (less than 0.3 %). We emphasize that these false negatives come from neither our design nor our implementation but rather from the limitations of the LLVM frontend. Therefore, we still refer to our framework as automated.

Even if the compiler missed inserting an instrumentation routine, this would result in a fault as the control data contains an index instead of the data. Similarly, if the protection selector failed to find a control-data instance, the instrumentation routines would interpret the control data as an index, resulting in a false exploit detection. During our evaluation, we encountered no such scenarios except for the false negatives caused by the LLVM frontend, which we manually fixed. Even if false negatives were present, we anticipate they could not be exploited, as they would result in a fault.

7. Security Discussion

Improvement over existing works. HEK-CFI improves security over existing works, as outlined in Table 6.1. For instance, only KCoFI provides comprehensive thread state protection among the kernel CFI schemes. Thus, all schemes except KCoFI fail to mitigate all of our motivational exploitation attacks presented in Appendix 12, as well as the Google Project Zero's thread state exploit (cf. CVE-2022-42703) [33]. In contrast, HEK-CFI successfully mitigates them. Although a combination of solutions such as KCoFI, Fine-CFI, and Intel CET SHSTK could theoretically offer similar protections for thread state and return addresses, HEK-CFI still improves forward edge protection (see Table 6.3) with substantially lower performance overhead. In particular, HEK-CFI reduces the CFI targets by over 50 % (see Section 6) compared to the combined forward CFI precision of these mitigations, represented by Fine-CFI's enhanced precision. Due to the unavailability of these solutions for empirical comparison, we estimate their combined overhead to be significantly higher than that of HEK-CFI. Specifically, HEK-CFI's performance overhead for macro benchmarks is about 1.85 %, as we will evaluate later in Section 8, whereas the individual overheads for KCoFI and Fine-CFI are around 10 % each. Similarly, when comparing HEK-CFI with a theoretical combination of KCoFI, Ge et al.'s solution, and SHSTK, HEK-CFI reduces targets by more than 76 %, representing the enhancement over Ge et al.'s solution. Like the previous example, we expect the combined overhead of these solutions to be significantly higher, with KCoFI and Ge et al.'s solutions individually contributing overheads of 10 % and 2 %, respectively.

Attacking control-data integrity. An attacker attempts to perform the following attack scenarios on kernel control-data integrity. Firstly, they overwrite the index stored in place of control data with an index referencing other control data. When the control data is accessed, HEK-CFI verifies that the control data's address matches the stored address in the safe storage. HEK-CFI interprets a mismatch as an exploit attempt. Secondly, they overwrite the index to reference outside the safe area. HEK-CFI performs a bounds check and, hence, detects the exploit attempt. Thirdly, they tamper with the global or local safe areas directly. However, the safe area is write-protected by Intel CET and any write access by non-shadow stack operations results in a control-protection exception.

Confused deputy attack. A confused deputy attack [41] tricks a high-privilege routine to perform a write operation to protected data. Since

only shadow stack instructions, i.e., `wrssq`, are permitted for writing to shadow stack pages, we identify the following high-privilege routines that may be targeted: Instrumentation routines for the safe areas, i.e., `rq_*` and `ptls_*`. However, these routines cannot be exploited for a deputy attack as they only write to the safe area if the access was validated. Furthermore, these routines only write to the current shadow stack or per-CPU storage. Since an attacker can neither control the shadow stack pointer nor the per-CPU storage is mapped, these routines can also not be used for a deputy attack. Overall, HEK-CFI effectively mitigates the confused deputy attack, preventing the existence of a `wrssq`-gadget.

Thread state. The Linux kernel processes system events that store the thread state in memory, as illustrated in Figures 6.1 and 6.2. Directly overwriting the state, e.g., `rip` or `cs`, on the data stack ⑤ will cause a control-protection exception on `iret` as Intel CET validates that the `rip` on data and shadow stack is equal. Directly tampering with pages marked as shadow stack also leads to a control-protection exception. Moreover, an attacker may manipulate general-purpose registers ⑥ stored in memory. For instance, the attacker overwrites a register stored on the data stack that will be used for a control-flow transfer later during execution. However, HEK-CFI stores all general-purpose registers within the PTLS and validates the registers to be equal on state restoration. Hence, HEK-CFI detects the tampering attempt. In attack scenario ❺, an attacker attempts to tamper with the thread state restored on a context switch event. However, since HEK-CFI protects the stored thread state within PTLS and the shadow stack state to the global safe area, each corruption attempt is detected. By manipulating any of these safe areas, Intel CET raises a control-protection exception which HEK-CFI interprets as an exploitation attempt.

Architectural-defined control data. The Linux kernel has various architectural-defined control data that may be targeted for control-flow hijacking attacks. x86 CPUs contain Interrupt Descriptor Tables (IDT), Global Descriptor Tables (GDT), and Local Descriptor Tables (LDT) storing security-critical system configurations. The IDT stores sensitive information, such as interrupt entry locations, CPU privilege level on interrupt entry, and used stack. However, corrupting the IDT is not possible on recent Linux versions because the kernel maps it as read-only. Both descriptor tables, GDT and LDT, store information about memory segmentations, such as the code segment. The code segment determines the CPU privilege level on interrupt (and exception) entry and exit, as well

as the base address used by indirect and direct control-flow transfers [23]. By tampering with the descriptor tables, an attacker may change data of segments, e.g., code segment. HEK-CFI maps the GDT as a shadow stack page to prevent descriptor table corruption and uses the `wrssq` instruction for write operations. The LDT could be protected similarly to the GDT, with its address protected with control-data integrity. However, in our proof-of-concept, we compiled the Linux kernel with `CONFIG_MODIFY_LDT_SYSCALL` reset, not supporting LDT. This may affect compatibility with older user applications, but we encountered no issues during the evaluation.

Control-Flow Bending. Control-Flow Bending (CFB) [9] is an attack that exploits so-called dispatcher functions that corrupts their own return address using malicious arguments. Even with a fully-precise static CFI scheme [9], commonly used functions have a large set of permitted backward edges that can be used for a CFB attack. Thus, CFI-based mitigations that do not fully protect return addresses can be bypassed by CFB attacks [23]. However, HEK-CFI includes a shadow stack, which prevents dispatcher functions from corrupting their return address and effectively mitigates CFB.

Pointer-to-pointer corruption. HEK-CFI is designed to protect control data from being corrupted. However, an attacker may corrupt a non-control data pointer to a control data, e.g., the attacker may corrupt a pointer within a list of inodes (`list_head`) to forge an `inode`. Fortunately, our control-data integrity scheme provides generic protection that can also be applied to protect non-control data pointers. We acknowledge that identifying an appropriate set of non-control data pointers to protect is a subject that requires further research and is an area we plan to explore in future work.

Stack tampering. Xu et al. [69] recently presented their novel WarpAttack attack targeting CFI schemes. They demonstrated that compiler optimizations may introduce double-fetch vulnerabilities when registers containing control data are spilled to the stack, making them vulnerable to corruption attacks. Fortunately, HEK-CFI's PTLS can prevent against WarpAttack by storing these registers within PTLS instead of pushing them to the stack. While WarpAttack protection is possible, it is currently out of scope as extending the compiler stage would require significant engineering effort.

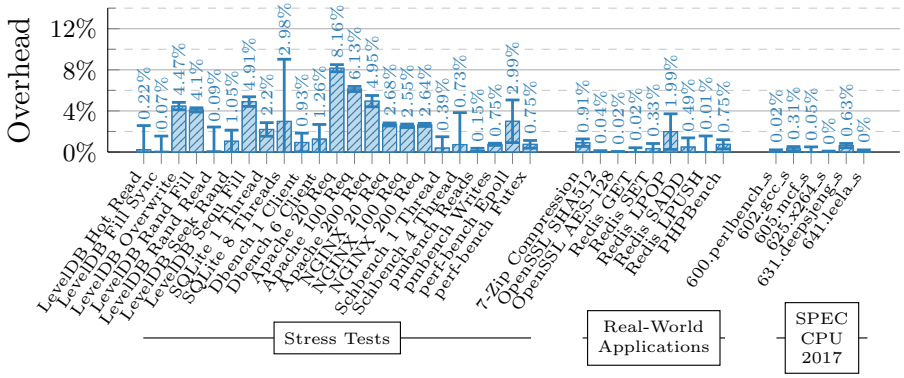


Figure 6.7: Macro benchmark results.

8. Performance Evaluation

We evaluate our proof-of-concept's runtime overhead by performing micro benchmarks with LMbench [49], and macro benchmarks with Phoronix Test Suite [53] and SPEC CPU 2017 [15]. We observe an overhead of $12.3 \pm 1.5\%$ for micro benchmarks, while macro benchmarks from Phoronix and SPEC increase the overhead by $1.85 \pm 1.02\%$ and $0.17 \pm 0.23\%$. We run our proof-of-concept on Ubuntu 22.04.1 LTS on the Intel Alder Lake processor i7-12700k, supporting Intel CET. We also evaluate the compile time, binary size, and analyzer overhead in Appendix 16.

Micro benchmarks. We use LMbench to evaluate the latency and bandwidth overhead. We consider the Linux kernel v5.18 baseline and our proof-of-concept HEK-CFI enhanced one. We run each benchmark 80 times and compute the geometric mean and standard deviation. Table 6.4 illustrates the evaluation results for HEK-CFI, with the geometric mean being $12.3 \pm 1.5\%$.

Phoronix Test Suite. We split our benchmarks from Phoronix Test Suite into stress tests and real-world applications, as illustrated in Figure 6.7. Among the stress tests are three database, two webserver, one scheduler, one virtual paging, and one performance tool benchmarks. Since the webserver and database benchmarks use a lot of control-data pointers with common signatures, e.g., `void (*)(struct sk_buff *)` and `void (*)(struct inode *)`, these benchmarks elevate the overhead between 0.01 % to 8.16 %. The Schbench benchmark illustrates that HEK-CFI has little impact on scheduling performance as the caused overhead is between

0.39 % to 0.73 %. Among the real-world applications are one in-memory database, and three user applications, having a performance overhead between 0.01 % to 1.99 %. The computed geometric mean of all Phoronix Test Suite macro benchmarks is $1.85 \pm 1.02\%$.

We observe an elevated standard deviation for benchmarks, particularly multi-threaded ones. However, since these are present in both kernel binaries, the baseline and HEK-CFI-enhanced, they are caused by the noisy kernel properties, e.g., interrupts.

SPEC CPU 2017. We perform various speed SPEC CPU 2017 macro benchmarks, as illustrated in Figure 6.7. The resulting overheads are with a geometric mean of $0.17 \pm 0.23\%$, in line with the results of the real-world applications from Phoronix Test Suite.

9. Related Work

FineIBT [22, 50] provides effective protection for function pointers with rare function signatures, as they enforce control-flow integrity with function signature granularity while having little impact on the performance [22, 61]. However, FineIBT does not protect return addresses. kCFI [3] protects forward-edge control-flow transfers similarly to FineIBT, but instead of relying on Intel's IBT hardware feature, it uses Clang's software solution. PATTER [70] uses ARM's Pointer Authentication (PA) [4] to sign and authenticate function pointers and return addresses and, hence, protects against tampering of these two control data but does not protect operation table pointers. Camouflage [18] protects selected function and operation table pointers, and return addresses, with ARM PA. All four proposed mitigations [3, 18, 50, 70] do not protect the thread state.

KCoFI [16] is a coarse-grained CFI scheme with protection for the thread state. For forward-edge control-flow transfers, they only reduce the targets by 98.18 %, and for backward transfers, KCoFI enforces that a function return must land at one of the call sites that could have called it. Moreover, KCoFI has a performance overhead of above 10 % and 100 % for macro and micro benchmarks, respectively.

Ge et al. [23] proposed a method of retrofitting kernel software to provide fine-grained CFI. They demonstrated that their FreeBSD proof-of-concept reduces performance overhead and control-flow targets compared to a

function signature-based CFI scheme. Their proof-of-concept implementation has a reasonable performance overhead of around 2 % for macro benchmarks. To protect against attack scenarios ⑤ and ⑥ from Figure 6.1, they disable preemption during kernel space execution. The kernel may also raise a page fault exception on common occasions, e.g., `copy_from_user`, which they address by storing the `rip` to an unused debug register on exception entry and validating on exit. However, as this debug register is stored in writable memory on a context switch (and they do not protect the thread state on a context switch ❺), an attacker can corrupt the memory to gain control of the register on restoration. Subsequently, the instruction pointer is set to the corrupted debug register (⑤) on exception exit, hijacking the control flow.

Li et al. [43] proposed Fine-CFI that reduces the indirect control-flow targets over previous CFI schemes [16, 23]. It induces the overhead by about 10 % and 8 % for macro benchmarks from Phoronix and SPEC, respectively. Moreover, Fine-CFI insufficiently protects the thread state on an `iret` instruction as it only validates the `rip` and `cs`. This leaves the stored state unprotected for attack scenarios ⑥ and ❺ from Figures 6.1 and 6.2, respectively.

PAL [64] is a kernel CFI-based defense that uses ARM PA to protect function pointers and return addresses with 1 % to 5 % overhead for macro benchmarks. However, their measured overheads must be interpreted cautiously, as they fluctuate by up to 200 %, as reported in their appendix. PAL does not protect operation table pointers and insufficiently protects the thread state. On a preemption, the kernel stores the registers in memory, and PAL computes a signature of all registers, including a time-based nonce. Then, PAL stores the nonce and signature in memory. After preemption, it verifies the registers to ensure they have not been corrupted. We identify three security concerns. First, it only validates the registers on preemption, not on a context switch event. Second, PAL is susceptible to replay attacks, where an attacker manipulates the nonce, signature, and register values to match a previously authenticated version. Third, storing the registers in memory before signing them makes PAL vulnerable to TOCTTOU attacks, where an attacker corrupts the stored register values before the signature is computed. Overall, PAL is vulnerable to attack scenarios ⑤, ⑥, and ❺.

Carlini et al. [9] demonstrated that CFB can bypass CFI schemes that determine the backward edges statically. Since these mitigations, except

PATTER, Camouflage, and PAL, do so, their scheme is vulnerable to CFB, re-enabling control-flow hijacking attacks.

While CFI is a powerful approach, there are other approaches [2, 25, 37, 38, 48, 54, 67] to prevent kernel control-flow hijacking attacks. Although we donot discuss these, their existence highlights the ongoing efforts to enhance kernel security.

10. Conclusion

In this paper, we introduced HEK-CFI, which provides hardware-enforced protection for control data, effectively mitigating control-flow hijacking attacks. Our HEK-CFI was established as the first kernel CFI-based countermeasure to provide protection for both thread state during system events and return addresses. At the same time, it generically reduces the forward control-flow targets and performance overhead compared to existing kernel mitigations. Overall, HEK-CFI increases kernel security.

11. Acknowledgements

We thank the anonymous reviewers and shepard for their valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE and AWARE project (FFG grant number 888087 and 891092). Any opinions, findings, 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 (pp. 152, 155).

[2] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In: USENIX Security Symposium. 2021 (p. 185).

[3] Android. Kernel Control Flow Integrity. 2022. URL: https://source.android.com/docs/security/test/kcfi (pp. 152, 154, 159–161, 175–177, 183, 197).

[4] ARM. Arm Architecture Reference Manual for A-profile architecture. Feb. 2022 (pp. 160, 183).

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In: AsiaCCS. 2011 (p. 152).

[6] Daniel P Bovet and Marco Cesati. Understanding the Linux Kernel. O'Reilly Media, Inc., 2005 (pp. 152, 157).

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: ACM Conference on Computer and Communications Security (CCS). 2008 (p. 152).

[8] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (p. 157).

[9] 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. 2015 (pp. 152, 160, 175, 181, 184).

[10] Nicholas Carlini and David A. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In: USENIX Security. 2014 (p. 152).

[11] Andrew Cooper. Xen CET Supervisor Shadow Stacks. Feb. 2021. URL: https://xenbits.xen.org/people/andrewcoop/Xen-CET-SS.pdf (p. 171).

[12] Jonathan Corbet. Defending against Rowhammer in the kernel. Oct. 2016. URL: https://lwn.net/Articles/704920/ (p. 157).

[13] Jonathan Corbet. Kernel security: beyond bug fixing. Oct. 2015. URL: https://lwn.net/Articles/662219/ (p. 157).

[14] Jonathan Corbet. Supervisor mode access prevention. Sept. 2012. URL: https://lwn.net/Articles/517475/ (pp. 152, 156).

[15] Standard Performance Evaluation Corporation. SPEC CPU 2017. 2017. URL: https://www.spec.org/cpu2017/ (pp. 154, 182).

[16] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In: S&P. 2014 (pp. 152, 154, 159–161, 175–177, 183, 184).

[17] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In: NDSS. 2017 (p. 156).

[18] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Camouflage: Hardware-assisted CFI for the ARM Linux kernel. In: DAC. 2020 (pp. 152, 156, 157, 159–161, 183).

[19] Jake Edge. Extending the use of RO and NX. 2011. URL: `https://lwn.net/Articles/422487/` (pp. 152, 156).

[20] Jake Edge. Kernel address space layout randomization. 2013. URL: `https://lwn.net/Articles/569635/` (p. 152).

[21] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the Effectiveness of Type-Based Control Flow Integrity. In: ACSAC. 2018 (p. 155).

[22] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: arXiv:2303.16353 (2023) (pp. 152, 161, 170, 173, 183).

[23] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In: Euro S&P. 2016 (pp. 152, 154, 156, 159, 160, 167, 175–177, 181, 183, 184).

[24] GitHub. CodeQL. 2021. URL: `https://codeql.github.com/` (pp. 153, 171, 173).

[25] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. IskiOS: Lightweight Defense Against Kernel-Level Code-Reuse Attacks. In: arXiv:1903.04654 (2019) (p. 185).

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 157).

[27] Daniel Gruss, Michael Schwarz, and Moritz Lipp. Meltdown: Basics, Details, Consequences. In: Black Hat USA. 2018 (p. 157).

[28] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: USENIX Security. 2009 (p. 152).

[29] Intel. Control-flow Enforcement Technology Preview. Revision 2.0. June 2017 (pp. 164, 172).

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (pp. 155, 159, 160).

[31] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers. May 2019 (pp. 172, 173).

[32] Seth Jenkins. Analyzing a Modern In-the-wild Android Exploit. 2023. URL: https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html (p. 157).

[33] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack. 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html (pp. 152, 158, 162, 179).

[34] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security. 2014 (p. 152).

[35] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In: USENIX Security. 2012 (p. 152).

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 157).

[37] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In: NDSS. 2013 (p. 185).

[38] Anil Kurmus and Robby Zippel. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In: CCS. 2014 (p. 185).

[39] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In: OSDI. 2014 (pp. 152, 155, 156, 159, 160).

[40] 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 (pp. 153, 171).

[41] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In: NDSS. 2022 (p. 179).

[42] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A Hybrid Alias Analysis and Its Application to Global Variable Protection in the Linux Kernel. In: USENIX Security. 2023 (p. 197).

[43] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. In: IEEE Transactions on Information Forensics and Security (2018) (pp. 152, 154, 156, 159, 160, 167, 175–178, 184).

[44] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In: USENIX. 2019 (pp. 155, 160).

[45] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 156, 157).

[46] Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In: AsiaCCS. 2024 (p. 149).

[47] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (p. 157).

[48] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In: NDSS. 2022 (p. 185).

[49] Larry McVoy and Carl Staelin. Lmbench: Portable Tools for Performance Analysis. In: USENIX ATC. 1996 (pp. 154, 182, 202).

[50] Joao Moreira. Kernel FineIBT Support. Apr. 2022. URL: `https://lwn.net/Articles/891976/` (pp. 154, 159, 160, 173, 175–177, 183).

[51] James Morse. arm64: kernel: Add support for Privileged Access Never. 2015. URL: `https://lwn.net/Articles/651614/` (p. 152).

[52] PaX Team. Rap: Rip rop. 2015 (p. 152).

[53] Phoronix. OpenBenchmarking. 2022. URL: https://openbenchmarking.org (pp. 154, 182).

[54] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. kR^ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In: EuroSys. 2017 (p. 185).

[55] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In: S&P. 2020 (p. 157).

[56] Samsung Knox News. Real-time Kernel Protection (RKP). 2016. URL: https://www.samsungknox.com/de/blog/real-time-kernel-protection-rkp (p. 156).

[57] 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 (p. 157).

[58] Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. Mar. 2015. URL: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (p. 157).

[59] INetCop Security. New Reliable Android Kernel Root Exploitation Techniques. Nov. 2016. URL: http://powerofcommunity.net/poc2016/x82.pdf (p. 197).

[60] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS. 2007 (p. 155).

[61] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In: HASP. 2019 (p. 183).

[62] Di Shen. Defeating Samsung KNOX with Zero Privilege. July 2017. URL: https://infocondb.org/con/black-hat/black-hat-usa-2017/defeating-samsung-knox-with-zero-privilege (p. 197).

[63] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In: NDSS. 2016 (p. 157).

[64] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In: USENIX Security. 2022 (pp. 152, 156, 159, 160, 167, 178, 184).

[65] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: S&P. 2010 (p. 152).

[66] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: USENIX Security. 2019 (p. 155).

[67] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In: USENIX Security. 2018 (p. 185).

[68] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In: CCS. 2022 (pp. 152, 156, 159, 160, 163).

[69] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini†, Bing Mao, and Mathias Payer. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In: S&P. 2023 (p. 181).

[70] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels. In: arXiv:1912.10666 (2019) (pp. 152, 159, 160, 183).

[71] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (p. 157).

[72] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In: USENIX Security. 2013 (pp. 175, 178).

[73] Xiaochen Zou. CVE-2022-27666 Writeup. Mar. 2022. URL: `https://etenal.me/archives/1825` (p. 197).

Appendix

12. Motivational Exploit Examples

In this section, we demonstrate various attack scenarios where an attacker obtains a Control-Flow Hijacking Primitive (CFHP), allowing them to deviate from the legal Control-Flow Graph (CFG) in an attacker-controlled manner. In the cases of Appendices 12.1 and 12.2, these CFHP scenarios also allow adversaries to bypass the control-flow restrictions imposed by applied kernel CFI-based countermeasures. As a result, attackers can redirect the control flow to an attacker-controlled code location. Furthermore, in Appendix 12.3, we show that even being within the approximated CFG with signature granularity is insufficient to mitigate this attack, leaving the system vulnerable to compromise by adversaries. Our approach, HEK-CFI, addresses these attack scenarios by protecting the thread state during all system events and control-data pointers, i.e., function and operation table pointers.

12.1. Attacking Thread State on Exceptions and Interrupts

Suppose an interrupt or exception request disrupts a thread. In that case, the system stores the thread state, including general-purpose registers that may hold control data, to a writable memory location, e.g., thread stack frame. This allows the system to resume the thread's execution once the interrupt or exception handling is completed. However, there is a security issue. If an attacker tampers with the memory holding control data in the saved thread state, they gain control over the control data when the system restores the thread state. As a result, the attacker obtains a CFHP.

Figure 6.8 illustrates this attack scenario by pivoting the `dev_rel` function (shown in C, x86_64 assembly, and arm64 assembly). This function makes an indirect branch to the address stored in `dev->rel`. To exploit `dev_rel` as a CFHP, an attacker can initiate an asynchronous interrupt or exception, such as using the high-precision `hrtimer` interface in the Linux kernel. This interrupt or exception may happen right before the assembly code's indirect branch in Line 4.If the interrupt or exception occurs precisely at this point, the attacker can manipulate the register `rbx` or `x20` stored on the stack frame. When the interrupt or exception handling finishes, the execution continues with the manipulated register. Hence, the indirect

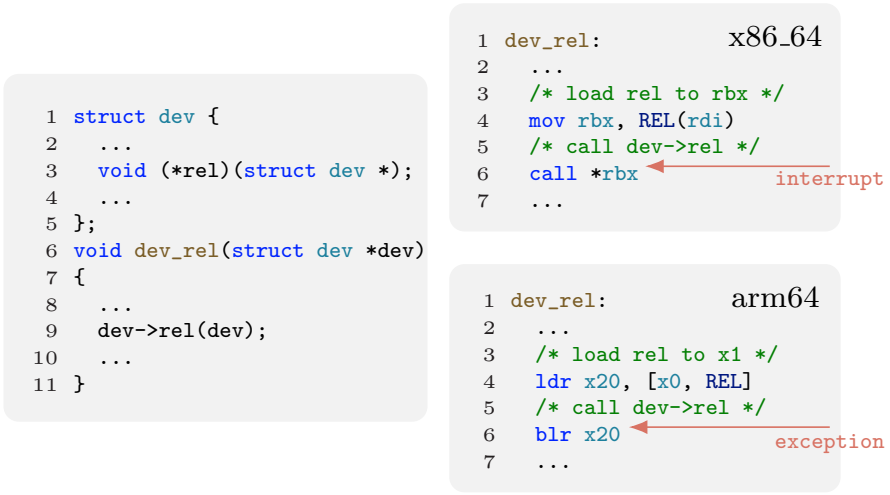


Figure 6.8: `dev_rel` performs an indirect branch (for x86_64 and arm64). If an interrupt/exception is triggered shortly before the branch, the register (`rbx` or `x20`) is stored to writable memory. By overwriting this memory location, an attacker can gain control over the register, resulting in a CFHP.

branch redirects the control flow to the value stored in the tampered register, effectively hijacking the control flow.

In Figure 6.8, we exemplify this attack on the thread state during the system events exception and interrupts with the function `dev_rel`. This can be generalized for every function, performing an indirect branch which can be disrupted by an exception, e.g., page fault on `copy_from/to_user`, or interrupt, e.g., timer interrupt.

12.2. Attacking Thread State on Context Switch

Exploiting instruction sequence of `ret_from_fork`. In this attack scenario, an attacker tampers with callee-saved registers (part of the thread state) of a thread currently not running to obtain a CFHP. In Figure 6.9, we illustrate the assembly instruction sequence, which is executed as first instruction sequence for a just created process using `fork` to return to the user space. For x86_64, the kernel executes the `ret_from_fork` function as shown in Listing 6.4. If an attacker tampers with the callee-saved registers between the `fork` and execution of `ret_from_fork`, they gain control over `rbx` and `r12` (which will be `rdi`). Since `rbx` is not zero (Line 4), the

```
1 ret_from_fork:
2   mov rax, rdi
3   call schedule_tail
4   test rbx, rbx
5   jne 1f
6 2:
7   mov rsp, rdi
8   call syscall_exit_to_user_mode
9   jmp __irqentry_text_end
10 1:
11  mov r12, rdi
12  call *rbx
13  jmp 2f
```

Listing 6.4: For x86_64.

```
1 ret_from_fork:
2   bl schedule_tail
3   cbz x19, 1f
4   mov x0, x20
5   blr x19
6 1:
7   get_current_task tsk
8   mov x0, sp
9   bl asm_exit_to_user_mode
10  b ret_to_user
```

Listing 6.5: For arm64.

Figure 6.9: Instruction sequence `ret_from_fork` that can be exploited for a CFHP by tampering with the callee-saved register, `rbx` for x86_64 and `x19` for arm64.

indirect call at Line 12 is made with the attacker-controlled registers. This control over the indirect call is what allows the attacker to achieve a CFHP. The function `ret_from_fork` for arm64 (see Listing 6.5) closely resembles the one for x86_64. By corrupting the memory location, where `x19` is stored, (i.e., `taxk_struct`) an attacker gains control over it, which is then used for a CFHP in Line 5.

For a more detailed exploitation, we refer to Appendix 13, where we exploit the CVE-2019-2215 to perform an end-to-end attack.

Exploiting generic code patterns. Another way to achieve a CFHP is through code patterns that involve storing a function pointer in a callee-saved register and then calling functions leading to a context switch, such as `schedule` or `mutex_lock`. On the context switch, the thread state including the callee-saved register is stored in memory, and the thread is marked as not running. Similar to the previous example, an attacker interferes with the memory location where the callee-saved register is stored while the thread is not running. This manipulation allows the attacker to gain control over the function pointer, resulting in a CFHP when the thread resumes execution to call the function pointer.

We illustrate in Listing 6.6 an example of such a code pattern. The function `rq_qos_wait` loads a function pointer into a register in Line 5 and uses it in an indirect branch in Line 13. In between, it may perform a context switch

```
1 void rq_qos_wait(...)
2 {
3   struct rq_qos_wait_data data = {
4     ...
5     .cb = acquire_inflight_cb,
6   };
7   ...
8   do {
9     /* break out of while loop */
10    if (data.got_token)
11      break;
12    /* perform indirect branch */
13    data.cb(...);
14    ...
15    /* perform context switch */
16    schedule();
17  } while (1);
18 }
```

Listing 6.6: Code pattern example that can be exploited for a CFHP. An attacker tampers with the callee-saved register, containing the function pointer `data.cb` (Line 5), during the `schedule` function (Line 16) to have control over the indirect branch (Line 13).

in Line 16. Crucially, whether the function uses a callee-saved register or not depends on the compiler. For our observations, we compiled the Linux v5.18 with gcc version 10.2.1 for both x86_64 and arm64 architectures using the default configuration of Ubuntu 22.04.1 LTS. In both cases, the kernel used callee-saved registers to store the function pointer, resulting that `rq_qos_wait` can be exploited to obtain a CFHP.

12.3. Attacking Signature-based CFI

CFI with signature granularity provides weak security guarantees for the Linux kernel because the set of control-flow targets matching the signature is too large. In Figure 6.10, we demonstrate an example of how an attacker can bypass signature-granular CFI. To explain, the attacker begins by opening a file where they have read- but not write-permission, e.g., `/etc/passwd`. This action causes the kernel to allocate a `file` object containing an operation table pointer to a `file_operations` object. Within the `ext4` filesystem, its members `read_iter` and `write_iter` point to functions `ext4_file_read_iter` and `ext4_file_write_iter`. By performing a read or write operation to or from the opened file,

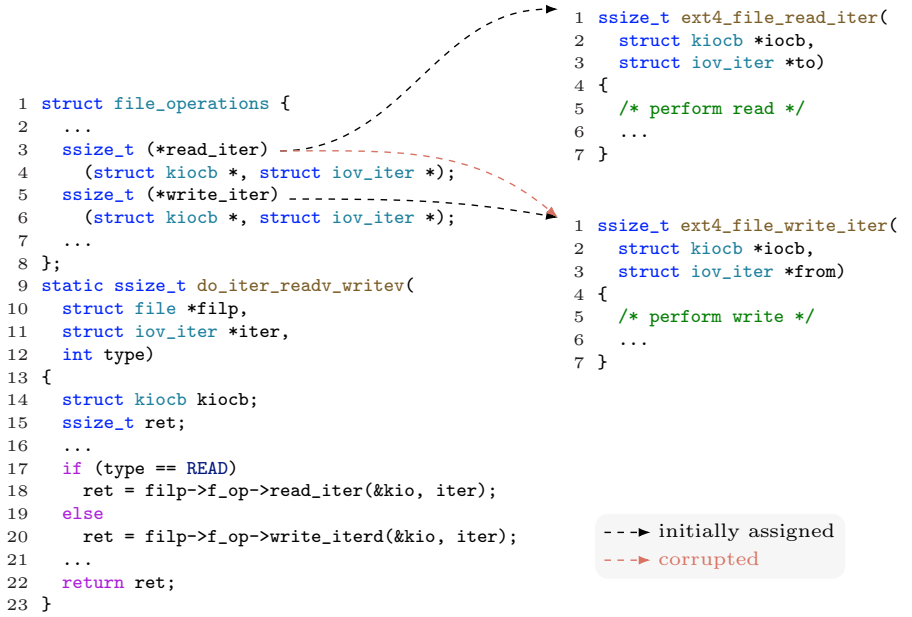


Figure 6.10: Exploitation to break signature-granular CFI. By overwriting the function pointer `read_iter` with `ext4_file_write_iter` an attacker enforce to call the write function on a read request.

the kernel calls `do_iter_readv_writev`. Since both signatures match, the attacker can overwrite `read_iter` member with the address of `ext4_file_write_iter` and still resides in the over-approximated CFG determined based on signatures. However, this manipulation causes a read operation to perform a write operation. Consequently, the attacker calls the `read` syscall to write content to the file where they do not have write-permissions, e.g., `/etc/passwd`. This results in a persistent privilege escalation, which CFI with signature granularity cannot prevent.

In fact, since the operation table `file_operations` in Figure 6.10 is mapped as read-only, an attacker cannot directly manipulate the function pointer `read_iter`. However, they can forge an operation table and tamper with the operation table pointer of the `file` object, pointing to the forged operation table. In this way, they can bypass the read-only mapping and perform this privilege escalation attack.

```
1 void find_process(void)
2 {
3   size_t kernel_base = leak_kaslr();
4   printf("[*] looking for the process \"t1\"...\n");
5   size_t task = INIT_TASK_OFFSET + kernel_base;
6   size_t task_stack;
7   while (1) {
8     arb_read(task + TASKS_OFFSET, (size_t)&task);
9     task = task - TASKS_OFFSET;
10
11    char name[8] = {0};
12    arb_read(task + COMM_OFFSET, (size_t)name);
13    if (!strcmp((char *)name, "t1")) {
14      arb_read(task + STACK_OFFSET, (size_t)&task_stack);
15      printf("[+] we found the process at %lx stack %lx\n", task,
  -> task_stack);
16      break;
17    }
18  }
19 }
```

Listing 6.7: Obtaining `task_struct`'s kernel address and the kernel stack address of the process named `"t1"`.

13. Motivational End-to-End Exploit

This section provides a motivational end-to-end attack that exploits the CVE-2019-2215 vulnerability. The vulnerability allows an arbitrary read-and-write primitive that matches our threat model in Section 3.1. This end-to-end attack shows the severity of not protecting the thread state, as this attack can bypass almost all kernel CFI schemes (mitigations that do not protect the thread state, see Table 6.1) including the one used by Android [3], i.e., kCFI. Therefore, this attack can be used through Android kernel exploitation to perform control-flow hijacking attacks, which often relied on using the arbitrary read-and-write primitive to tamper with global control data pointers [42], e.g., `ptmx_fops` [59, 62] or `modprobe_path` [73]. The high-level exploitation strategy is a three-step plan as follows.

As the *first* step, we create a new process by calling `fork`, which prompts the kernel to allocate a `task_struct` and an associated kernel stack. The `task_struct` is allocated via the dedicated allocation cache `task_struct_cachep`, while the kernel stack is allocated via `vmalloc`, both of which reuse freed objects for future allocations. We change the name of the just

```
__switch_to_asm:
  // Save callee-saved regs
  pushq rbp
  pushq rbx
  pushq r12
  pushq r13
  pushq r14
  pushq r15

  // switch stack
  movq  rsp, TASK_sp(rdi)
  movq  TASK_sp(rsi), rsp

  // restore callee-saved regs
  popq  r15
  popq  r14
  popq  r13
  popq  r12
  popq  rbx
  popq  rbp

  jmp __switch_to
```

Listing 6.8: For x86_64.

```
cpu_switch_to:
  mov x10, #THREAD_CPU_CXT
  add x8, x0, x10
  mov x9, sp

  // store callee-saved regs
  stp x19, x20, [x8], #16
  stp x21, x22, [x8], #16
  stp x23, x24, [x8], #16
  stp x25, x26, [x8], #16
  stp x27, x28, [x8], #16
  stp x29, x9, [x8], #16
  str lr, [x8]
  add x8, x1, x10

  // restore callee-saved regs
  ldp x19, x20, [x8], #16
  ldp x21, x22, [x8], #16
  ldp x23, x24, [x8], #16
  ldp x25, x26, [x8], #16
  ldp x27, x28, [x8], #16
  ldp x29, x9, [x8], #16
  ldr lr, [x8]

  // restore stack pointer
  mov sp, x9
  msr sp_el0, x1
  ret
```

Listing 6.9: For arm64.

Figure 6.11: Context switch assembly code of the Linux kernel.

created process to a unique one, i.e., `"t1"`, and use our read primitive to obtain the address of its `task_struct`. The way to do this is by leaking `init_task` (Line 5 of Listing 6.7), which is the `task_struct` of the initial, i.e., first, high-privilege process. Since the Linux kernel stores all processes as a linked list (via the `struct list_head tasks` member), we iterate through all processes at Line 8 and use the arbitrary read to obtain the process's names at Line 12. We check whether its name (via the `char comm[TASK_COMM_LEN]` member) is the same as the unique one at Line 13, we previously set to `"t1"`. If the stored name is the same, we leak the stored stack (via the `void *stack` member variable) at Line 14[2].

[2]Leaking the kernel stack's address is only necessary for x86_64 as arm64 stores the callee-saved registers to its `task_struct`

As the *second* step, we kill the process named `"t1"` prompting the kernel to free its `task_struct` and kernel stack. By creating a new process via `fork` called `"t2"` shortly after we reclaim both the previously leaked `task_struct` and kernel stack[3]. Immediatly after this `fork`, we use our arbitrary write primitive to overwrite the stored callee-saved registers (as part of the thread state) stored either on the kernel stack for x86_64 or the `task_struct` for arm64 (denoted as `THREAD_CPU_CXT`). For x86_64 these are `rbp`, `rbx`, and `r12-15`, while for arm64 `x19-29` and `x9`. If process `"t2"` is scheduled, it loads these manipulated register values from the stack or `task_struct` as shown with `__switch_to_asm` in Listing 6.8 and `cpu_switch_to` in Listing 6.9, respectively. Since it is the first time scheduled, the Linux kernel prompts this process to execute `ret_from_fork` shown in Figure 6.9. With control over the register, we just created a CFHP with control over its arguments (Lines 11 and 12 for x86_64 and Lines 4 and 5 for arm64), allowing an adversary to perform a powerful control-flow hijacking attack.

As the *third* step, we perform a classical control-flow hijacking attack such as executing an instruction sequence equivalent to `commit_creds(prepare_kernel_cred(0))` to escalate privileges. This powerful control-flow hijacking attack depends on a race window, to tamper with the process's (`"t2"`) kernel stacks or `task_struct` before it gets scheduled. However, this race window is large and does not affect the practicality of this attack.

14. Instrumentation

This section demonstrates the instrumentation process of HEK-CFI, where we illustrate how HEK-CFI ensures control-data integrity for globally and locally accessible control data.

Globally accessible control data. Listing 6.10 depicts an allocation and deallocation of `dev`, and a read from and write to its member variable `void (*rel)(struct dev *)`. We assume that HEK-CFI provides protection for the function pointer `rel`. Although this example protects a function pointer, the same applies to protected operation table pointers.

[3]To bypass the free-list randomisation of the kernel heap allocator and reclaim the leaked `task_struct`, we massage the dedicated allocator `task_struct_cachep`. For the sake of simplicity, we assume that we successfully reclaim it.

```
struct dev {
  /* protected fn pointer */
  void (*rel)(struct dev *);
  ...
};
void dev_rel(struct dev *);
void function(void) {
  struct dev *dev;
  dev = kmalloc();
  ...
  dev->rel = &dev_rel;
  ...
  if (dev->rel)
    dev->rel(dev);
  ...
  kfree(dev);
}
```

Listing 6.10: Original code of accessing a struct dynamically allocated.

```
void function(void) {
  struct dev *dev;
  dev = kmalloc();
+ ss_alloc(dev->rel);
  ...
- dev->rel = &dev_rel;
+ ss_wr(dev->rel, &dev_rel);
  ...
- if (dev->rel)
-   dev->rel(dev);
+ reg = ss_rd(dev->rel);
+ if (reg)
+   reg(dev);
  ...
+ ss_free(dev->rel);
  kfree(dev);
}
```

Listing 6.11: Instrumented code of accessing a struct dynamically allocated.

```
struct delayed_call {
  /* protected fn pointer */
  void (*fn)(void *);
  void *arg;
};
void dc_fn(void *);
void function2(void) {
  struct delayed_call dc;
  ...
  dc.fn = &dc_fn;
  ...
  dc.fn(dc.arg);
  ...
}
```

Listing 6.12: Original code of a local struct.

```
void function2(void) {
  struct delayed_call dc;
+ ptls_alloc(dc.fn);
  ...
- dc.fn = &dc_fn;
+ ptls_wr(dc.fn, &dc_fn);
  ...
- dc.fn(dc.arg);
+ reg = ptls_rd(dc.fn);
+ reg(dc.arg);
  ...
+ ptls_free(dc.fn);
}
```

Listing 6.13: Instrumented code of a local struct.

For the instrumentation, we refer to the function `ss_alloc`, `ss_free`, `ss_rd`, and `ss_wr`, to globally accessible safe storage allocation, deallocation, read, and write.

Listing 6.11 demonstrates how HEK-CFI protects the function pointer `rel` when the struct it belongs to (`dev`) is dynamically allocated. HEK-CFI inserts `ss_alloc` to allocate a safe storage for `rel` shortly after dynamically allocating `dev`. HEK-CFI replaces the write access to `rel` with `ss_wr`, which writes to the protected data within the allocated safe storage. On read access, HEK-CFI replaces the read with `ss_rd`, which reads the protected data from the safe storage and stores it in the register `reg`. HEK-CFI inserts `ss_free` shortly before freeing the `dev` to free the safe storage.

Locally accessible control data. Listing 6.12 demonstrates the local variable `delayed_call` containing a function pointer `void (*fn) (void *)` protected by HEK-CFI. For the instrumentation, we refer to the functions `ptls_alloc`, `ptls_free`, `ptls_rd`, and `ptls_wr`, to locally accessible safe storage allocation, deallocation, read, and write, respectively. As illustrated in Listing 6.13, HEK-CFI inserts `ptls_alloc` and `ptls_free` on function entry and exit. Additionally, HEK-CFI replaces the write and read access to or from the protected function pointer `fn` with `ptls_wr` and `ptls_rd`.

15. Determine Ideal User Policy

We developed a tool that takes the average permitted control-flow targets value, i.e., $AIA_{hek-cfi}$ illustrated in Equation (6.2), as input and determines the user policy, i.e., maximum permitted forward control-flow targets, as output. It does so by first finding the value of p, the number of permitted function pointers of Equation (6.2), to match the given input. This value p resides within the range of 1 to n. After determining p, the tool identifies the set $|T_i|$ with the largest number of permitted control-flow targets. This largest number is the output value of the user policy.

Table 6.4: Micro benchmark results.

	Benchmarks	**Baseline**	**Overhead** in %
Latency in µs	fcntl lock	1.51	18.9 ± 0.9
	pagefault	0.17	18.3 ± 1.4
	proc call	0.0027	-1.9 ± 3.3
	proc fork	81.5	8.7 ± 1.9
	proc fork+exec	90.5	8.9 ± 1.5
	proc shell	359	7.6 ± 0.8
	signal install	0.15	15.4 ± 1.9
	signal catch	1.12	8.8 ± 0.3
	signal fault	0.48	8.1 ± 1.0
	syscall null	0.079	25.8 ± 0.3
	syscall open+close	1.19	18.7 ± 2.3
	syscall read	0.12	20.6 ± 1.6
	syscall write	0.095	24.8 ± 3.9
	syscall stat	0.44	9.4 ± 1.1
	syscall fstat	0.15	18.2 ± 0.9
Bandwidth in MB/s	pipe 1 k	0.13	24.3 ± 4.2
	pipe 128 k	3.74	20.8 ± 0.2
	unix 1 k	337	11.2 ± 0.3
	unix 128 k	82	11.3 ± 0.8
	file rd 4 k o2c	2.69	15.4 ± 1.1
	file rd 1 M o2c	15	6.2 ± 0.7
	file rd 4 k ip	7.71	13.6 ± 0.8
	file rd 1 M ip	16.2	7.5 ± 0.4
	mmap rd o2c 4 k	1.34	10.3 ± 0.4
	mmap rd o2c 1 M	14.8	6.3 ± 0.5
	mmap rd 4 k	43.7	0.1 ± 0.2
	mmap rd 1 M	41.6	1.4 ± 2.0
	geo mean	-	12.3 ± 1.5

16. Detailed Evaluation Results

In this section, we illustrate the results of our micro benchmark performance evaluation in Table 6.4, performed with LMbench [49]. Moreover, we evaluate the binary and compile-time overhead of the Linux kernel enhanced with HEK-CFI. Lastly, we evaluate the time taken by our code analyzer and parser.

Binary size and compile time overhead. Since HEK-CFI inserts validation checks to ensure its functionality, the inserted checks increase both the binary size and the compile time. To illustrate their increase, we

compile an unmodified Linux kernel v5.18 with clang version 15.0.0 as a baseline. We then compile our modified Linux kernel with our LLVM pass and the user policy described in our case study. The results are that the binary size and compile-time increase by 0.97 % and $47.2 \pm 1.6\%$, respectively.

Code analyzer and parser. We measure the time taken to generate a database and query requests for CodeQL. Moreover, we measure the required time for parsing the results from CodeQL to output the file for our LLVM pass. We perform each measurement eight times and calculate the geometric mean and standard deviation, where the results are 2h53m39s±6s, 59m4s±6s, and 17.8s±0.4s for database generation, database queries, and parser, respectively.

7

SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

Publication Data

Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024

Contributions

The author of this thesis is the main author of this work. The author's contributions are the proposal of *side-channel supported recycling and reclaiming* and *novel exploitation method*, and *comprehensive analysis and attack evaluation*, as well as most of the written text.

SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

Lukas Maar Stefan Gast Martin Unterguggenberger
Mathias Oberhuber Stefan Mangard

Graz University of Technology

Abstract

While the number of vulnerabilities in the Linux kernel has increased significantly in recent years, most have limited capabilities, such as corrupting a few bytes in restricted allocator caches. To elevate their capabilities, security researchers have proposed software cross-cache attacks, exploiting the memory reuse of the kernel allocator. However, such cross-cache attacks are impractical due to their low success rate of only 40 %, with failure scenarios often resulting in a system crash.

In this paper, we present SLUBStick, a novel kernel exploitation technique elevating a limited heap vulnerability to an arbitrary memory read-and-write primitive. SLUBStick operates in multiple stages: Initially, it exploits a timing side channel of the allocator to perform a cross-cache attack reliably. Concretely, exploiting the side-channel leakage pushes the success rate to above 99 % for frequently used generic caches. SLUBStick then exploits code patterns prevalent in the Linux kernel to convert a limited heap vulnerability into a page table manipulation, thereby granting the capability to read and write memory arbitrarily. We demonstrate the applicability of SLUBStick by systematically analyzing two Linux kernel versions, v5.19 and v6.2. Lastly, we evaluate SLUBStick with a synthetic vulnerability and 9 real-world CVEs, showcasing privilege escalation and container escape in the Linux kernel with state-of-the-art kernel defenses enabled.

1. Introduction

Operating system kernels, such as Linux, are susceptible to memory safety vulnerabilities due to their size and complexity. However, most of these vulnerabilities have limited capabilities, such as corrupting a few bytes in restricted allocator caches. These limitations make exploitation difficult in practice. To make these vulnerabilities even more difficult to exploit, researchers and kernel developers have included defenses such as SMAP, KASLR, and kCFI [38]. In addition, the kernel's allocator is designed to restrict exploits that propagate from heap vulnerabilities. One particular hardening strategy is to enforce coarse-grained heap separation. This separation places objects in distinct allocator caches that maintain blocks of adjacent pages, called slabs, and separate security-critical objects from frequently used objects. Hence, vulnerabilities in frequently used caches cannot be directly exploited to manipulate security-critical objects, such as credentials.

To circumvent coarse-grained heap separation, security researchers [51] presented software cross-cache attacks, which have been used by several kernel exploits [2, 13, 17, 19, 28, 29, 30, 48]. Software cross-cache attacks exploit the memory reuse of the kernel allocator as follows: Initially, an adversary triggers a heap vulnerability to obtain and hold on to a write capability for a victim object. They then free all memory slots on the slab page containing the victim object and allocate a different (sensitive) object type. This triggers the allocation of new slab pages, presumably reclaiming the previously freed and recycled slab page. The adversary then continues to overwrite the victim object, which now resides in the same memory location as the newly allocated sensitive object, corrupting it.

The Linux kernel has two types of allocator caches: dedicated and generic caches. While dedicated caches can be reliably exploited for cross-cache attacks [2, 17, 19, 28, 48], generic caches cannot [28, 51]. In particular, exploitation of generic caches has a success rate of only 40 % [51], with failure scenarios often leading to system crashes. To increase the reliability of generic cache exploitation, security experts [13, 29, 30, 48] have used stabilization objects, e.g., `msg_msg` or `pipe_buffer`. However, these objects cannot be used in newer kernel versions due to more refined heap separation, i.e., v5.14 introduced `kmalloc-cg-*`. Therefore, for newer kernel versions, cross-cache attacks on generic caches do not provide the reliability required in practice [28, 51].

In this paper, we present SLUBStick, a novel kernel exploitation technique that converts a limited kernel heap vulnerability into an arbitrary read-and-write primitive. At its core, SLUBStick exploits timing side-channel leakage of the kernel's allocator to reliably trigger the recycling and reclaiming process for a specific memory target. Exploiting this side-channel leakage significantly enhances the success rate of software cross-cache attacks, exceeding 99 % for generic caches with a single slab page and 82 % for multiple slab pages. With this substantial increase, our approach overcomes the prior unreliability and makes cross-cache attacks practical for exploitation. Using our reliable side-channel supported approach, SLUBStick performs a cross-cache attack to recycle a slab page that contains a write capability. SLUBStick then reclaims the slab page as a page table, i.e., Page Upper Directory (PUD), used for userspace address translation. By triggering the write capability, SLUBStick overwrites page table entries, obtaining arbitrary read and write capabilities.

To perform SLUBStick, we overcome the following technical challenges: First, we present reliably exploitable primitives for our timing side channel that are accessible to unprivileged users. Second, hardly any kernel heap vulnerabilities provide the capability to modify kernel data directly. Therefore, we present techniques that exploit code patterns prevalent in the Linux kernel. These techniques convert heap vulnerabilities before the recycling phase to allow a write capability after reclamation as a page table. Third, manipulating page table entries to obtain an arbitrary memory read-and-write primitive is challenging because the physical memory layout is randomized due to KASLR, and we do not assume address information leakage. Hence, we introduce a reliable solution that obtains such a primitive from an overwrite.

We conduct a systematic analysis for two Linux kernel versions, v5.19 and v6.2, providing a comprehensive list of primitives to successfully execute SLUBStick for all generic caches from `kmalloc-8` to `kmalloc-4096`. We also evaluate SLUBStick with a synthetic vulnerability as well as with 9 real-world CVEs for both kernel versions on x86_64 as well as aarch64, demonstrating its architecture and kernel version independence. Based on these findings, we conclude that SLUBStick poses a significant threat to kernel security.

Contributions. The main contributions of SLUBStick are:

(1) Side-Channel Supported Recycling and Reclaiming: We present a novel approach to reliably trigger the recycling process of a specific memory target and reclaim it by using a software timing side channel. Our approach shows success rates exceeding 99 % for frequently used generic caches, making cross-cache attacks practical.
(2) Novel Exploitation Method: Leveraging our reliable side-channel supported recycling and reclaiming approach, we present a novel exploitation technique to convert kernel heap vulnerabilities with limited capabilities into an arbitrary memory read-and-write primitive with state-of-the-art kernel defenses enabled.
(3) Comprehensive Analysis and Attack Evaluation: We systematically analyze two Linux kernel versions, v5.19 and v6.2, showing that SLUBStick can be executed for generic cache from `kmalloc-8` to `kmalloc-4096`. We also evaluate SLUBStick using a synthetic vulnerability and 9 real-world CVEs to escalate privileges.

Outline. Section 2 describes the background and threat model. Section 3 presents SLUBStick. Section 4 introduces our reliable recycling and reclaiming process. Section 5 describes pivoting heap vulnerabilities. Section 6 details how to gain arbitrary read-and-write capabilities. Section 7 comprehensively evaluates our attack. Section 8 discusses valuable insights and kernel defenses. Section 9 concludes this work.

2. Background and Threat Model

2.1. Buddy and SLUB Allocator

Linux's page allocator is based on the *Binary Buddy Allocator* [23], mainly referred to as buddy allocator. It allocates physical contiguous memory in chunks of page order size, i.e., $2^{\mathtt{n}} \cdot \mathtt{PAGE\_SIZE}$, where `n` is the page order. Moreover, it combines this page-order allocation with free chunk merging.

SLUB allocator. As the buddy allocator only provides page-order allocations, the slab allocator caches available objects with a predefined size in a multi-level free-list hierarchy, using pages obtained from the buddy allocator. There are three main implementations: SLUB is the default choice for several Linux distributions [22], while SLOB has become obsolete, and SLAB will be deprecated soon [7].

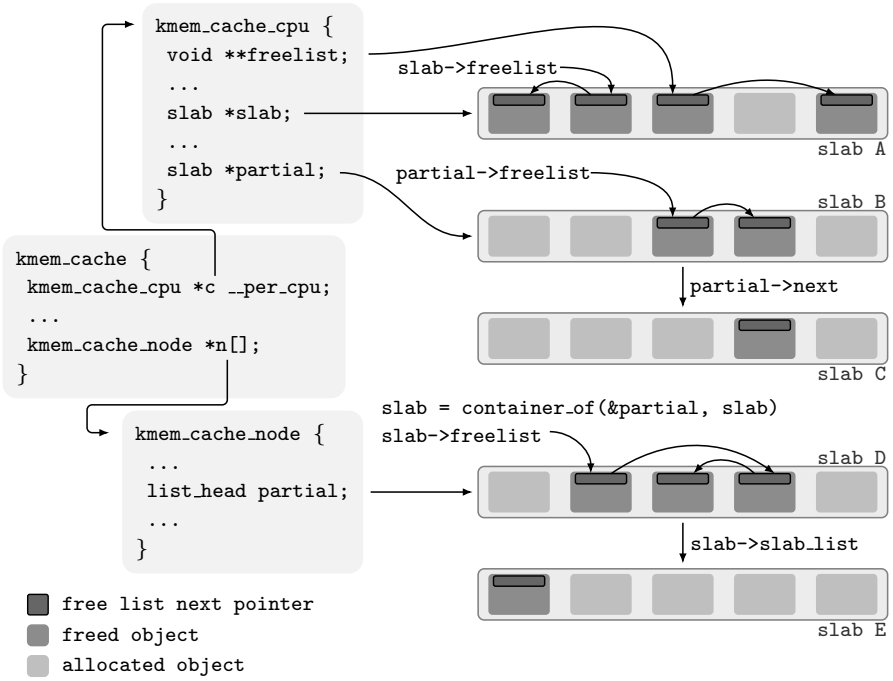


Figure 7.1: `kmem_cache` layout for the SLUB implementation.

In the Linux kernel, the SLUB allocator provides two primary types of allocator caches: dedicated and generic caches. Dedicated caches are employed for frequently used fixed-size objects, such as `cred` or `task_struct`. Generic caches are utilized for generic object allocation and deallocation or for objects whose sizes are not known during compile-time, e.g., elastic objects [5]. Both types of caches utilize `kmem_cache`, with each dedicated cache having its own `kmem_cache`, while generic caches have multiple `kmem_cache`s matched to different sizes. When allocating memory from a generic cache, the kernel matches the requested size to one of these caches and allocates an object from the corresponding `kmem_cache`.

The Linux kernel incorporates cache aliasing to optimize memory management. Cache aliasing merges freed objects stored in distinct `kmem_cache`s with similar characteristics, e.g., object size and allocation properties. For security reasons, `kmem_cache`s of dedicated or generic caches considered security-critical are marked as accounted to prevent aliasing. Essentially, these accounted `kmem_cache`s separate accounted objects from the non-accounted ones. Security-critical caches include those that store sensitive

information, such as `cred`, and objects commonly used for exploitation (allocated using `kmalloc-cg-*`), such as elastic objects [5].

Architecture. The architecture of a `kmem_cache` [22], shown in Figure 7.1, includes a `kmem_cache_cpu` for each logical CPU and an array of `kmem_cache_node`s. The `kmem_cache_cpu` comprises various free lists: a CPU free list (`c->freelist`), a slab free list (`slab->freelist`), and additional free lists of partial slabs (`partial->freelist`, maintained as a single-linked list). Despite each slab having its free list, the separate CPU free list allows lockless allocation, improving performance. The `kmem_cache_node` has a double-linked list of slabs (`partial`) also containing freed objects. In the context of this work, we refer to a list (i.e., free list, and single- and double-linked list) as full when it reaches its capacity of containing objects. It is considered empty when no object is present in it. A list is classified as partial when it is neither full nor empty.

Allocation and deallocation. `kmem_cache` stores objects in a multi-level free-list hierarchy. As shown in Figure 7.2, the allocation process starts by searching for an available object in the lower free-list levels [22]. This process continues throughout the hierarchy until an available object is found. These levels include the CPU free list ①, slab free list ②, CPU partial slab list ③, and node partial slab list ④, with each level taking more allocation time. If no object is available in any of these free lists, the SLUB allocator falls back to the buddy allocator ⑤, which allocates a memory chunk.

When deallocating, the SLUB allocator attempts to place the object in the lower free-list levels, e.g., CPU free list ① [22], as shown in Figure 7.3. Upon deallocation, the kernel may check the number of free slabs, i.e., the number of slabs with full free list stored in the node partial slab list ③. If this number exceeds a particular capability (see Table 7.4), the SLUB allocator deallocates the slab's memory chunks ⑤, returning them to the buddy allocator. Memory chunks returned in such a recycling phase are reused for future allocations.

Timing attacks on allocation. Lee et al. [26] demonstrated with PSPRAY the feasibility of performing a timing side channel on the SLUB allocator. PSPRAY deduces when the allocator allocates a fresh memory chunk (see ⑤ in Figure 7.2). This insight increases the likelihood of successful kernel heap exploitation, which primarily relied on heap spraying, i.e., for Use-After-Free (UAF) and Double-Free (DF), or heap grooming, i.e., for Out-Of-Bounds (OOB) [4, 26, 54].

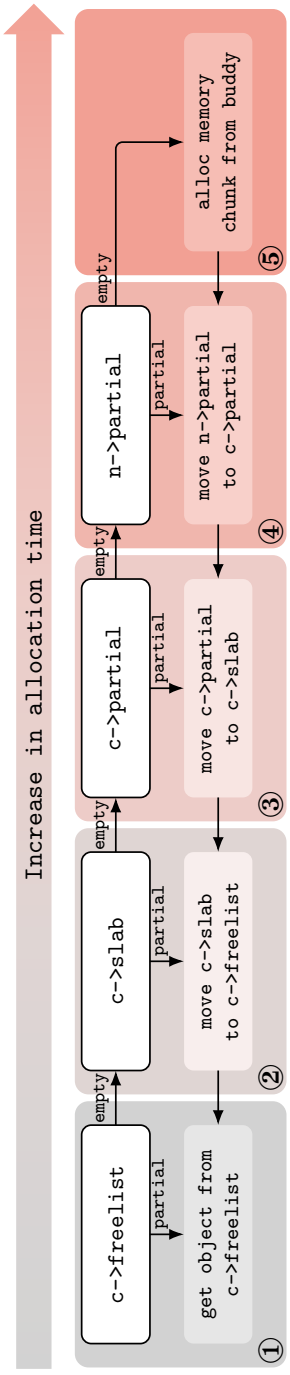


Figure 7.2: SLUB allocation of an object, where the terms `c` and `n` refer to the `kmem_cache_cpu` and `kmem_cache_node`, respectively. The free lists (i.e., `c->freelist` and `c->slab->freelist`) and slab lists (i.e., `c->partial` and `n->partial`) are checked to be either `empty` or `partial`.

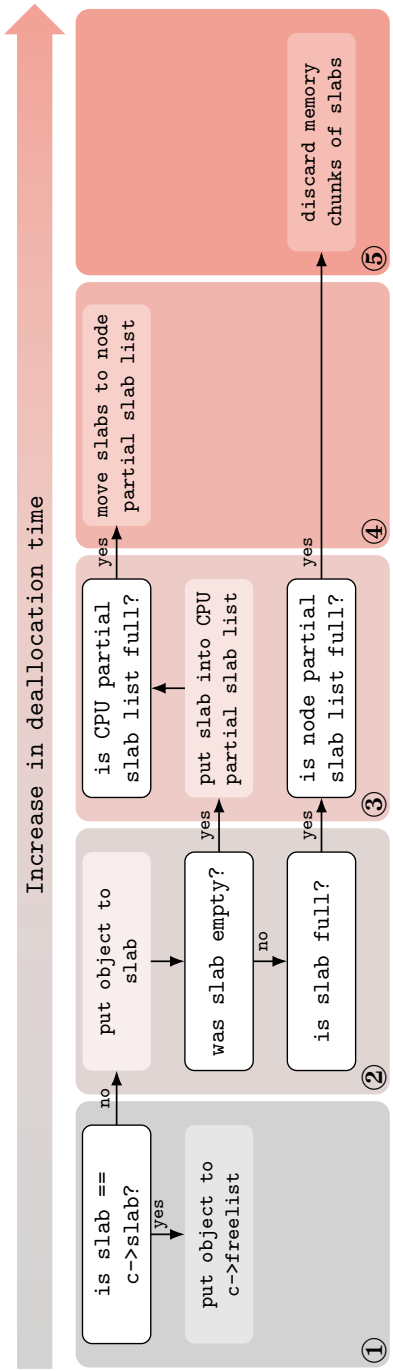


Figure 7.3: SLUB deallocation of an object, where the term `slab` refers to the slab that contains the object to be freed. The term `c` represents the `kmem_cache_cpu` associated with this slab. The `slab` is either active (i.e., slab stored as `c->slab`), or stored in the partial slab list (i.e., slab located within `c->partial`) or node partial slab list (i.e., slab located within `n->partial`).

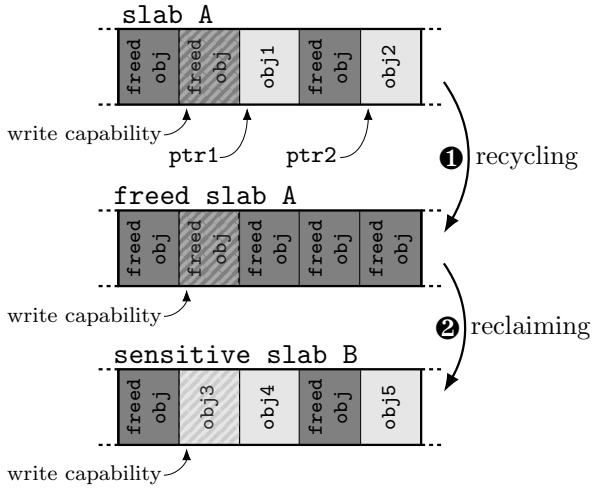


Figure 7.4: Software cross-cache attack with an initial state, where a write capability refers to a freed object. An attacker enforces a recycling ❶ of `slab A`'s memory chunk by freeing `obj1/2`. By allocating sensitive objects, the attacker presumably reclaims ❷ the chunk for a sensitive `slab B`, resulting in write capability referring `obj3`. Lastly, `obj3` is overwritten.

However, their method relies on a precise measurement primitive that is no longer available in recent kernel versions. Their primary proposed primitive uses `msg_msg`. Since it is allocated via the segregated `kmalloc-cg-*` for kernel versions v5.14 or higher, it is limited to scenarios where the vulnerable object is also allocated from the segregated generic cache. Other proposed primitives are limited because the computational overhead of non-allocation tasks primarily masks allocation timing (e.g., `read`). Furthermore, their approach fails to identify suitable measurement primitives, e.g., for `kmalloc-8/16`. For other identified primitives, we could not reproduce the allocation of a single data object (e.g., `fchown`), or the primitives are privileged and therefore unusable (e.g., `kexec_load`). We contacted the authors about the applicability of using their identified syscalls (apart from `msg_msg`) to determine the timing difference. They confirmed that the overhead of the syscalls they identified limits their applicability. In summary, while their work demonstrates feasibility, its applicability is limited to older kernel versions.

2.2. Software Cross-Cache Attacks

When the SLUB allocator frees memory chunks using the buddy allocator, as shown with ⑤ in Figure 7.3, these chunks are reused. Classic cross-cache attacks [2, 17, 19, 28, 29, 48, 51] exploit this reusing behavior. Initially, an adversary compels the SLUB allocator to recycle a memory chunk containing a write capability due to vulnerabilities. Subsequently, they allocate numerous sensitive objects from another allocator cache, hoping to reclaim the previously freed chunk. If successful, the memory that was previously occupied with the write capability will now be occupied by sensitive objects. Lastly, they trigger the write capability to this memory, corrupting a sensitive data object. The recycling ❶ and reclaiming ❷ phases are shown in a simplified setting in Figure 7.4.

Xu et al. [51] demonstrated the feasibility of cross-cache attacks, and subsequent research [28, 29] has further explored their impact. However, executing such attacks is notably challenging, particularly for frequently used generic caches. One significant hurdle is the introduction of noise through uncontrolled allocations, complicating to achieve state ⑤ in Figure 7.3. For example, unknown allocations from a kernel thread can thwart the freeing of the slab's memory chunk, thereby preventing the reclamation [28]. Adding to the complexity, the unpredictable occurrence of both phases, recycling ❶ and reclaiming ❷, introduces instability to the exploit.

In summary, mounting cross-cache attacks is complex and fraught with challenges. Although these attacks are compelling, in practice, they have a limited success rate, as low as 40 % [51]. Importantly, this percentage only represents the success rate of the cross-cache attack, excluding additional stages of an end-to-end exploit, e.g., vulnerability triggering and memory manipulation, which further reduces the overall success rate. The process of repeatedly triggering vulnerabilities carries its risks. Traces left in the kernel often make it difficult to trigger the same vulnerability again. For instance, an OOB write may corrupt lists when triggered. Hence, repeated activation of the vulnerability can result in a crash, severely limiting the attack's practicality.

2.3. Threat Model

We assume that an unprivileged user has code execution. Additionally, we consider the presence of a heap vulnerability in the Linux kernel. We

assume that the Linux kernel incorporates all defense mechanisms available in version 6.4, the most recent Linux kernel version when we started our work. These mechanisms include features such as `W^X`, KASLR, SMAP, and kCFI [38]. We do not assume any microarchitectural vulnerabilities, e.g., transient execution [24, 31], fault injection [44], or hardware side channels [3, 52].

In this work, we primarily focus on heap vulnerabilities (most common type of software vulnerability according to Microsoft [37]) that result in a Double-Free (DF), or a Use-After-Free (UAF) or an Out-Of-Bounds (OOB) allowing for a limited writing capability. For instance, CVE-2023-21400 enables the double free of an object within the `kmalloc-32` generic cache, while CVE-2023-3609 permits a write operation at offset 0x18 on an object allocated from `kmalloc-64`.

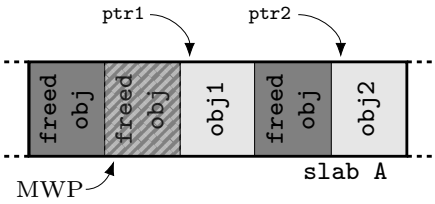
3. Technical Overview and Challenges

This section outlines SLUBStick's capability to overcome several technical challenges when exploiting a limited heap vulnerability to obtain an arbitrary read-and-write primitive.

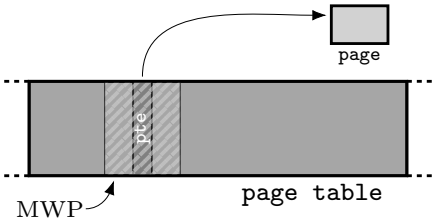
3.1. Overview

Obtaining an arbitrary read-and-write primitive with SLUBStick involves *three stages*, as depicted in Figure 7.5. In the *first stage* (see Figure 7.5a), SLUBStick exploits a heap vulnerability to acquire a Memory Write Primitive (MWP). This MWP provides an adversary with a write capability at an adversary-determined time, with the primitive referring to the memory of a freed object. Subsequently, SLUBStick triggers the recycling process of its slab's memory chunk by deallocating all objects of the slab. This chunk is then returned to the buddy allocator for future allocations. Crucially, even after the recycling, the MWP still refers to the recycled chunk.

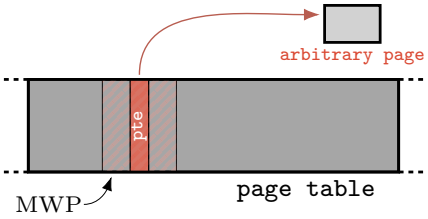
In the *second stage* (see Figure 7.5b), SLUBStick allocates page tables to reclaim the recycled memory chunk for a page-table page used to translate a userspace address. The entries in this table refer to user-accessible pages, storing crucial information, e.g., their page frame number and access permissions.



(a) *Stage 1* deallocates objects `obj1` and `obj2` to trigger the recycling process of `slab A`'s memory chunk, with a Memory Write Primitive (MWP) referring to it.



(b) *Stage 2* reclaims the recycled memory chunk for a page table used by the userspace address translation, containing one entry, `pte`, which refers to the user-accessible `page`.



(c) *Stage 3* triggers the MWP to manipulate the page table entry `pte`. This manipulated entry indexes then the `arbitrary page`, allowing it to be overwritten from the userspace.

Figure 7.5: High-level overview of SLUBStick subdivided into three stages exploiting an MWP.

In the *third stage* (see Figure 7.5c), SLUBStick triggers the MWP to overwrite the referenced memory of the page table. This manipulation allows an adversary to change the page frame number and permission bits, granting access to any physical page and adjusting user access permissions. As a result, the virtual address now refers to the selected page with permission to modify it from userspace. For example, by referring to kernel code, this capability allows the adversary to alter the kernel's behavior. As another example, the adversary can reference data of privileged files such as `/etc/passwd`, thus modifying it to bypass authentication checks.

SLUBStick does not violate the kernel's control flow or any of the existing kernel defenses outlined in Section 2.3. Hence, these existing kernel defenses cannot mitigate SLUBStick, highlighting its severe threat to current systems. Furthermore, efforts to separate generic caches [9] are ineffective in mitigating SLUBStick. We discuss the limitations of existing kernel defenses in Section 8.

3.2. Technical Challenges

This section briefly describes how SLUBStick overcomes several technical challenges to escalate privileges, while the subsequent sections discuss our solutions in more detail.

Triggering recycling/reclaiming. SLUBStick performs a cross-cache attack on generic caches. As described in Section 2.2, cross-cache attacks have been challenging to execute because they are unstable and unpredictable [28, 51]. Uncontrolled allocations and deallocations introduce noise, making it difficult to determine when the recycling and reclaiming phase will occur. However, we present a side-channel supported approach to reliably trigger the recycling process of a targeted memory chunk, which is then reclaimed. This significantly increases the success rate, exceeding 99 % for slabs with single page-size chunks and 82 % for multi-page-size chunks, which were previously considered impractical. In Section 4, we explain our novel approach, which exploits the side-channel timing leakage of the SLUB allocator.

Pivoting heap vulnerabilities. Pivoting a kernel heap vulnerability to establish an MWP presents two sub-challenges: First, SLUBStick obtains a dangling pointer from the vulnerability. DF vulnerabilities provide us with this capability, while OOB and UAF write vulnerabilities do not. Hence, SLUBStick requires an appropriate pivoting approach to utilize

them fully. Second, in most cases, a dangling pointer does not directly allow for controlled overwriting with controlled timing. Hence, SLUBStick must pivot the dangling pointer, performing an adversary-controlled write, i.e., MWP. In Section 5, we explain the process of adequately pivoting capabilities from kernel heap vulnerabilities to obtain the MWP.

Arbitrary read-and-write. By triggering the MWP, SLUBStick can overwrite a page table used for address translations of a userspace address. As the kernel memory layout is randomized (for physical addresses) by KASLR, and we do not assume any address information leakage, pivoting an MWP to an arbitrary memory read-and-write primitive is not straightforward. In Section 6, we explain how we achieve this pivot to an arbitrary read-and-write primitive.

4. Triggering Recycling and Reclaiming

In this section, we present a novel approach that reliably triggers the recycling process of a targeted memory chunk and reclaims it, making cross-cache attacks practical for exploitation. Our approach leverages a software timing side-channel attack on the SLUB allocator. Through this side channel, we gain valuable information about when a memory chunk is allocated and deallocated using the buddy allocator. These insights allow us to reliably trigger the recycling process of a targeted chunk (see Section 4.1). To ensure the return of this chunk during the reclamation phase, we massage the buddy allocator's internal state (see Section 4.2). By combining these two strategies, our novel approach triggers the recycling and reclaiming process reliably. To demonstrate the effectiveness, we provide a proof-of-concept (including experiments), offering comprehensive details on deploying generic caches from `kmalloc-8` to `kmalloc-4096` (see Section 4.3).

4.1. Side-Channel Supported Recycling

To trigger the recycling of a targeted memory chunk, our approach first groups allocated objects according to their slab by using a software timing side channel on the allocation. It then deallocates these grouped objects, prompting the kernel to recycle the slabs' chunks, including our targeted chunk.

```
1 ssize_t __do_sys_add_key(const char __user *_desc) {
2   ssize_t ret;
3   size_t len = strnlen_user(_desc) + 1;
4   char *desc = kmalloc(len, GFP_KERNEL);
5   size_t n = copy_from_user(desc, _desc, len);
6   if (n) goto ERR;
7   desc[len - 1] = 0;
8   if (IS_INVALID(desc)) goto ERR;
9   ... /* add_key code execution */
10 ERR:
11   kfree(desc);
12   return ret;
13 }
```

Listing 7.1: The `add_key` syscall serves as a measurement primitive to determine whether Line 4 allocates a new chunk.

The principal approach to grouping allocated objects involves timing measurements during standalone object allocation from userspace. A fast to medium timing indicates that the object was allocated from one of the slab's lists (①, ②, ③, or ④ in Figure 7.2). Hence, we group it with the previously allocated object in the same slab. Conversely, if the allocation time is notably higher (⑤), this indicates that the object originates from a newly allocated memory chunk and thus a new slab, which prompts us to group it separately. As a result, we obtain a list of grouped objects organized by their slabs.

One challenge with this approach is the lack of primitives in the Linux kernel to accurately measure standalone allocation times with minimal non-allocation tasks from userspace. This is because a single object allocation involves notable non-allocation tasks, e.g., allocating an object by opening its device. Using primitives with notable non-allocation tasks, as proposed by PSPRAY [26] for non-separated generic caches, leads to inaccurate and unusable results. To address this challenge, we separate the timing measurement from the persistent object allocation. While an allocation primitive persistently allocates an object, the timing measurement primitive is used to group this allocated object based on its slab.

Measurement primitive. We measure the timing of primitives that involve both allocation and deallocation within a single syscall while only performing minimal non-allocation tasks. During the deallocation process within this primitive, the previously allocated object is consistently set to the active CPU free list (state ① in Figure 7.3), which only minimally

affects the measured timing. The minimal non-allocation tasks executed during the syscall exhibit high consistency, which also minimally affects the results. Thus, the measured timing depends primarily on the allocation. This allows us to determine when a new memory chunk is allocated from userspace, indicated by a notable slow allocation timing. To illustrate the notable difference between fast and slow allocation timings, we provide detailed experiments in Section 10.2.

An example of a measurement primitive is the `add_key` syscall, shown in Listing 7.1. To ensure minimal interference from non-allocation tasks, our approach invokes the `add_key` syscall with a `_desc`, which fails the validation check at Line 8. As a result, the syscall primarily involves two privilege switches, allocation and deallocation, with the execution time mainly depending on the allocation. By measuring the timing of this syscall, our approach determines whether Line 4 allocates a new memory chunk. Crucially, while the `add_key` syscall is one example, measurement primitives are common in the Linux kernel, as we later illustrate in Section 4.3.

Persistent allocation primitive. A persistent allocation refers to a situation where the allocated kernel object remains allocated until the corresponding deallocation counterpart is executed. Crucially, the deallocation of the object must occur within the deallocation syscall, not at a later time. Therefore, objects released inside a Read-Copy-Update (RCU) [36] callback function cannot be used for persistent allocation. Also, the de-/allocation syscalls must be unprivileged.

Triggering the recycling process. Using our method of accurately measuring allocation timing and persistently allocating objects, our approach to reliably triggering the recycling process consists of *two stages*.

In the *first stage*, we allocate and group kernel objects based on their slabs. We start emptying all free lists within the cache's multi-level hierarchy by allocating many objects. Next, we persistently allocate objects to fill the free memory slots of the slab while using our measurement primitive to exploit the timing side channel for grouping the allocated objects. Listing 7.2 shows an example of combining a measurement primitive (e.g., `add_key`) with a persistent allocation primitive (e.g., `snd_ctl_file`). The `timed_alloc` function provides two outputs: the measured timing at Line 9 and the allocated object as an identifier at Line 11. Using the measured `*time`, we determine whether the object belongs to the same slab as the previously allocated object (i.e., indicated by a low value) or whether it

```
1 size_t timed_alloc(int *time) {
2   /* allocate the 64 byte struct snd_ctl_file */
3   int fd = open("/dev/snd/controlC0", O_RDONLY);
4   /* timed allocation with invalid add_key */
5   char _desc[64] = INVALID_DESC;
6   size_t t0 = rdtsc_begin();
7   add_key(_desc);
8   size_t t1 = rdtsc_end();
9   *time = t1 - t0;
10  /* return allocated object */
11  return fd;
12 }
```

Listing 7.2: An example of a 64 byte allocation from userspace, where Line 3 allocates a `snd_ctl_file` object. Between Lines 6 and 8, we use the `add_key` measurement primitive to determine whether a new chunk was allocated.

originates from a new memory chunk and hence a new slab (i.e., indicated by a notably high value). This allows us to group objects identified with `fd` by their slabs.

In the *second stage*, we proceed to free the grouped objects, prompting the Linux kernel to deallocate their slabs via the buddy allocator, as denoted by the state ⑤ in Figure 7.3. The release of slabs occurs when the number of free slabs exceeds the capacity of the node's partial slab list ③, detailed in Table 7.4. Since we know the objects of the slabs and the capacity of the partial slab list, we retain control over when the memory chunks, including our target, are recycled. While uncontrolled deallocations may introduce noise, we show remarkable noise resilience in our experiments in Section 4.3.

4.2. Massaging the Buddy Allocator

This section determines the reclaiming phase of a targeted memory chunk based on the recycling phase, where our approach is tailored to the generic cache size. A memory chunk can consist of a single or multiple pages, as shown in Table 7.4. Our goal is to ensure the reliable reclaiming of either the recycled chunk or a page-size part of it. Notably, the reclamation chunk's size is required to be one page, as SLUBStick reclaims it as a page table, as we later show in Section 6.

For generic caches from `kmalloc-8` to `kmalloc-256`, the memory chunk consists of a single page. When releasing slabs' memory chunks, the buddy allocator follows a Last In First Out (LIFO) principle to return recycled memory chunks. Therefore, to obtain a targeted chunk, one can request a page-size chunk shortly after the recycling process in the opposite order of the deallocation process.

In contrast, for generic caches from `kmalloc-512` to `kmalloc-4096`, the memory chunk consists of multiple pages. To ensure the reclaiming of the recycled memory chunk as page size, we need to put the allocator in a state where it splits the targeted memory chunk into smaller ones. To accomplish this, we take the following strategy: First, we empty smaller memory chunks, reducing the number of available smaller chunks. Next, we perform the recycling process, preparing for subsequent chunk splitting and reclaiming. Finally, we allocate multiple page-size chunks. This compels the kernel to split the targeted memory chunk into smaller ones, making it available for allocation. Through these steps, we ensure that the buddy allocator splits the targeted memory into smaller chunks and returns them for page-size allocation.

4.3. Proof-of-Concept and Experiments

For our Proof-Of-Concept (POC), we first systematically analyze the Linux kernel to identify kernel objects suitable for a measurement and allocation primitive. We then combine our approaches discussed in Sections 4.1 and 4.2 to reliably trigger the recycling and reclamation phase. Our POC demonstrates the feasibility and effectiveness of our approach with a success rate of between 99.3 % to 99.9 % for single page-size chunks and 82.1 % to 93.5 % for multi-page-size chunks.

Systematic analysis. Our approach requires a *measurement* (e.g., `add_key` in Listing 7.1) and an *allocation* (e.g., `snd_ctl_file` in Listing 7.2) primitive. We scan the kernel code for suitable code patterns using the query language CodeQL [15], as we explain in detail in Sections 10.3.2 and 10.3.3.

Many kernel execution paths in the kernel allow *measurement* primitives, as any snippet that copies user data to a dynamically allocated buffer and subsequently performs validation checks can be leveraged. This includes functions that delegate the allocation and copying process to commonly used routines like `strndup_user` and `memdup_user`. Hence, our analysis

yields a measurement primitive for each generic cache from `kmalloc-8` to `kmalloc-4096` (see Table 7.8).

Similarly, the kernel has many execution paths that allow persistent *allocation* primitives, as any unprivileged syscall that persistently allocates an object and has a freeing counterpart that instantly releases the object can be leveraged. As a result, our analysis also yields an allocation primitive for each of these generic caches (see Tables 7.6 and 7.7).

We conduct this analysis for Linux kernel versions v5.19 and v6.2. Our results, presented in Table 7.8 for measurement primitives and Tables 7.6 and 7.7 for allocation primitives, underline the high applicability of our approach across versions.

Experiments. We demonstrate the feasibility and effectiveness of our approach by showcasing how to reliably trigger the recycling process of a targeted memory chunk and reclaim it during the reclaiming phase. To perform this experiment, we use a read device driver `rdd`, shown in Listing 7.6. The driver provides object allocation (`ALLOC`), deallocation (`FREE`), and reading of the object's contents (`READ`). Allocation and deallocation are used to allocate and release an object on a target memory chunk while reading determines if the reclamation was successful. In this experiment, we use a page table as the object that reclaimed the recycled memory chunk. To allocate a page table, we map a page without all page table levels mapped. We consider the experiment successful if we can read the page table entries from our `rdd` after the reclaiming phase since this means that the page table successfully reclaimed our target memory chunk. The experiment is considered unsuccessful if we can not trigger the recycling process for the chunk containing the target object or if the recycled page is not reclaimed as a page table.

We ran this experiment 100 times to determine the success rate, which we repeated 10 times to compute the mean and standard deviation. We tested under an idle system with *CPU pinning* and two noise conditions: *no CPU pinning* and external *noise*, demonstrating our noise resilience. Our experimental setup was Ubuntu 22.04 LTS with the generic Linux kernel v6.2 for x86, where v5.19 gives similar results. We ran this on a machine with Intel i7-1260P and 48 GB RAM.

CPU pinning is the method used to fix a process to a CPU, preventing CPU migration. Since both the slab and buddy allocator maintain per-CPU lists, CPU migration may introduce noise. To examine this effect, we included both scenarios: with and without CPU pinning. To introduce

Table 7.1: Success rate of triggering the recycling and reclamation process for generic caches.

Generic Cache	#Pages	Success Rate		
		Idle	No CPU pinning	External noise
		%	%	%
kmalloc-8	1	99.9 ± 0.1	99.9 ± 0.1	99.6 ± 0.7
kmalloc-16	1	99.4 ± 0.6	98.9 ± 1.2	99.9 ± 0.4
kmalloc-32	1	99.4 ± 0.9	99.7 ± 0.5	99.9 ± 0.3
kmalloc-64	1	99.2 ± 1.3	99.2 ± 0.9	81.0 ± 6.4
kmalloc-96	1	99.9 ± 0.4	99.9 ± 0.1	99.8 ± 0.6
kmalloc-128	1	99.9 ± 0.4	99.8 ± 0.5	99.9 ± 0.3
kmalloc-192	1	99.9 ± 0.4	99.8 ± 0.4	99.3 ± 1.2
kmalloc-256	1	99.9 ± 0.3	99.9 ± 0.3	99.7 ± 0.7
kmalloc-512	2	90.2 ± 5.4	87.2 ± 3.1	65.2 ± 2.8
kmalloc-1024	4	88.1 ± 7.2	79.5 ± 3.3	70.3 ± 8.1
kmalloc-2048	8	83.1 ± 9.2	70.5 ± 16.0	57.8 ± 5.7
kmalloc-4096	8	82.1 ± 3.4	73.3 ± 19.0	53.8 ± 10.0

external *noise*, we ran `stress-ng` concurrently with two workers for each CPU, IO, and VM, saturating approximately 100 % usage on all CPUs. This aligns with prior noise evaluation [26].

Results. Under idle, our experiments (see Table 7.1) show remarkable results, with a more than 99.2 % success rate for generic caches with page-size chunks, i.e., `kmalloc-8` to `kmalloc-256`. The failure scenarios are primarily due to the targeted chunk that can not be reclaimed as a page table. The results of generic caches with multiple page-sized chunks decrease with increasing size from 90.2 % to 82.1 %. This is primarily due to the failure in reclaiming the targeted page from the split chunk. The larger the original chunk, the more error-prone the reclaiming is, as seen by the lowest success rate of about 83 % for generic caches with 8 pages.

Our results demonstrate remarkable noise resilience for almost all generic caches with page-size chunks with more than 98 % for no CPU pinning and external noise. One exception is `kmalloc-64` for the noise evaluation since `stress-ng` performs frequent allocation directly following deallocation for `kmalloc-64`. Hence, it appends a memory chunk to the node partial list, which makes it confusing to reclaim the memory chunk before the actual target chunk as the target chunk. For multiple page-sized chunks, no CPU pinning decreases the success rate to 70.5 %, while external noise decreases it to 53.8 %. Crucially, despite the induced noise, our approach is still notably better than the prior success rate of 40 % with no noise [51].

5. Pivoting Kernel Heap Vulnerabilities

SLUBStick leverages a kernel heap vulnerability to gain a Memory Write Primitive (MWP). This primitive provides an adversary with a write capability to previously freed memory at a controlled time. To achieve this, SLUBStick initially obtains a dangling pointer (see Section 5.1), i.e., a pointer referencing an already freed object from a heap vulnerability. SLUBStick then establishes an MWP from this dangling pointer (see Section 5.2). Crucially, we must extend the time window between converting the dangling pointer to an MWP (i.e., before the recycling) and when it is triggered (i.e., after the reclaiming). These steps provide the adversary with an MWP that can be triggered at the desired time.

5.1. Obtaining a Dangling Pointer

In this section, we pivot DF, and UAF and OOB write vulnerabilities to obtain a dangling pointer.

Double-Free. By exploiting a DF vulnerability and freeing an object twice, we induce a situation where the pointer to an object becomes dangling. However, freeing the same object twice will cause it to reside twice in the CPU free list, corrupting this free list and causing a system crash. Hence, to leverage a DF vulnerability for obtaining a dangling pointer, SLUBStick allocates an object after its initial freeing and then utilizes the subsequent double free to create a dangling pointer pointing to this newly allocated object. Reallocating the object prevents the duplicate free list entry, avoiding free-list corruption and a subsequent system crash.

Out-Of-Bounds and Use-After-Free write. While UAFs that allow write control over dangling pointers might work, they often fall short in practice. For direct use in SLUBStick, a UAF vulnerability would need specific criteria: First, they must retain overwriting capability post-zeroing during recycling. Second, they must be triggerable after reclaiming as a page table. Third, they require the ability to overwrite specific values, i.e., malicious page-table entries. However, most UAFs [2, 19, 40, 53] provide limited write primitives and small race windows, thus failing these criteria.

Hence, SLUBStick leverages significantly weaker write capabilities offered by UAF (and OOB) vulnerabilities commonly found in practice to enforce a double free of an object. It does so by using this capability to manipulate either a *reference counter* or an *object pointer* within an object of the

same cache. By corrupting these members, SLUBStick forces an object into a DF state (see **Double-Free** above).

In Linux, *reference counters* manage the lifetime of an object. At its core, the counter increases with each new reference and decreases with each release. When the counter reaches zero, all the object's resources are released. For exploitation, SLUBStick identifies a victim object within the same cache containing a reference counter located at the write capability. Using heap manipulation [6, 26, 51], it aligns this victim object in memory with the write capability. SLUBStick then utilizes the write capability to corrupt the counter, forcing the release of the object still in use and placing it in a DF state.

To demonstrate the applicability of this approach, we analyze the kernel for the available objects with a reference counter located at each potential 8 byte write capability. We found 873 objects with a reference counter. They cover 100 % of overwrite locations for generic caches up to `kmalloc-32` and more than 80 % for generic caches between `kmalloc-64` and `kmalloc-256`. For caches between `kmalloc-512` and `kmalloc-4096` the availability ranges from 73.4 % to 2.7 %.

Regarding *pointer* corruption, SLUBStick identifies a victim object within the same cache that contains a pointer to an object from a separate cache. SLUBStick then places the victim object to align the overwrite capability and triggers it to corrupt the pointer by zeroing out the two least significant bytes. This creates an additional reference to another object in the separate cache. When the kernel releases this object, SLUBStick obtains a dangling pointer to the other object stored in the separate cache without leaking KASLR.

The uncorrupted pointer must refer to an object in a different slab. Otherwise, the slab that now contains a double-referenced object cannot be recycled, as the object that was originally pointed to has lost its reference and cannot be freed. To ensure that the uncorrupted pointer refers to an object in a different slab, SLUBStick employs our side channel (see Section 4.1). This technique allows for detecting and preventing failure scenarios, enhancing the success rate of pivoting.

Primitive conversion. The success rate of converting a UAF (including DF) vulnerability to a dangling pointer relies on the time window between the free and use stages. Techniques like ExpRace [27] can enhance exploitability by widening the time window. Additionally, UAF and OOB

```
int ipmi_open(void) {
  struct ipmi_file_private *priv;
  /* allocate object */
  priv = kmalloc(sizeof(*priv));
}
long ipmi_ioctl(struct file *f, u64 data) {
  struct ipmi_file_private *priv;
  struct ipmi_timing_parms parms;
  /* copy data from user */
  copy_from_user(&parms, data);
  priv->parms = parms;
}
```

Listing 7.3: Persistent code pattern 1.

```
u64 netlink_sendmsg(struct msghdr *msg, u64 len) {
  /* allocate object */
  struct sk_buff *skb = kmalloc(len);
  /* copy data from user */
  copy_from_user(obj, msg, len);
}
```

Listing 7.4: Persistent code pattern 2.

```
u64 keyctl_pkey_verify(void *uaddr, void *uaddr2, u64 size,
 -> u64 size2) {
  /* allocate and copy data from user */
  void *in = kmalloc(size);
  copy_from_user(in, uaddr, size);
  /* second copy for extending time window */
  void *in2 = kmalloc(size2);
  copy_from_user(in2, uaddr2, size2);
  /* free obj */
  kfree(in2);
  kfree(in);
}
```

Listing 7.5: Temporal code pattern.

Figure 7.6: Examples of code patterns allowing to be exploited as a Memory Write Primitive (MWP).

write vulnerabilities require an object with a reference counter or pointer located at the write capability.

5.2. Establishing a Memory Write Primitive

In this section, we detail how SLUBStick establishes a Memory Write Primitive (MWP) from a dangling pointer, allowing an adversary to overwrite memory reclaimed for a page table (see Section 6). The main challenge is to extend the time window from converting the dangling pointer into an MWP before recycling, while triggering must be done after the reclaim as a page table. To address this challenge, we exploit three distinct code patterns prevalent in the Linux kernel, as demonstrated in our **systematic analysis**.

Persistent code pattern 1. The pattern shown in Listing 7.3 grants SLUBStick an MWP with control over the timing. The process involves allocating the memory slot referred to by the dangling pointer, e.g., as an `ipmi_file_private` object at Line 4, before recycling. After reclamation, SLUBStick triggers the MWP with `ipmi_ioctl` to overwrite page table entries at Line 11. Figure 7.7 shows **P1** with its timeline.

While this pattern allows the MWP to be triggered arbitrarily, it has limitations as the allocated object, e.g., `ipmi_file_private`, is zeroed during recycling. If other code parts access the (unexpectedly) zeroed data, this could lead to a crash. Notably, this limitation does not affect `ipmi_file_private`, but others like `timerfd_ioctl`.

Persistent code pattern 2. The second pattern (see Listing 7.4) allows SLUBStick to trigger an MWP at a controlled time as follows: It allocates the memory slot referred to by the dangling pointer, e.g., as `sk_buff` at Line 3, before the recycling. Within the same syscall, the overwriting step at Line 5 follows after memory reclamation. To extend the time window between allocation and overwriting, SLUBStick can use techniques such as `userfaultfd`, Filesystem for USErspace (FUSE) [16], or slow page fault [20], controlling the duration of a user page fault, e.g., from of `msg` at Line 5.

While prior work [29, 40, 56] controlled the duration of copying from userspace with `userfaultfd`, recent systems restrict `userfaultfd` to privileged users for handling user page faults while in kernel mode [8]. Alternatively, researchers [16] showed that FUSE also allows control of the duration of copying from userspace. FUSE is a framework for userspace filesystems and can be utilized by an unprivileged user [16, 21]. We hence use FUSE to extend the time window between allocation and overwriting,

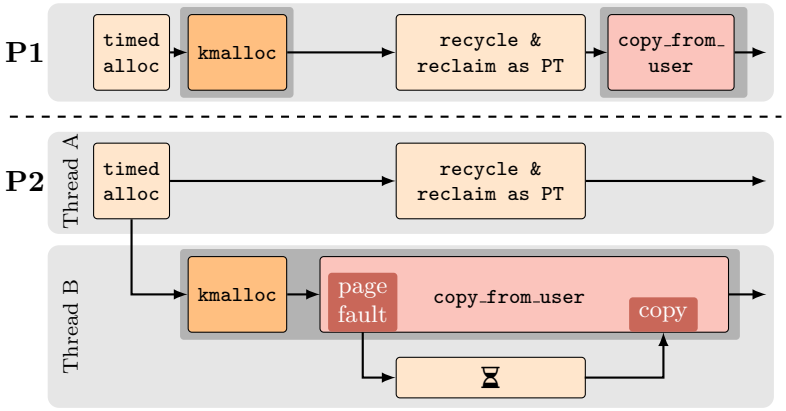


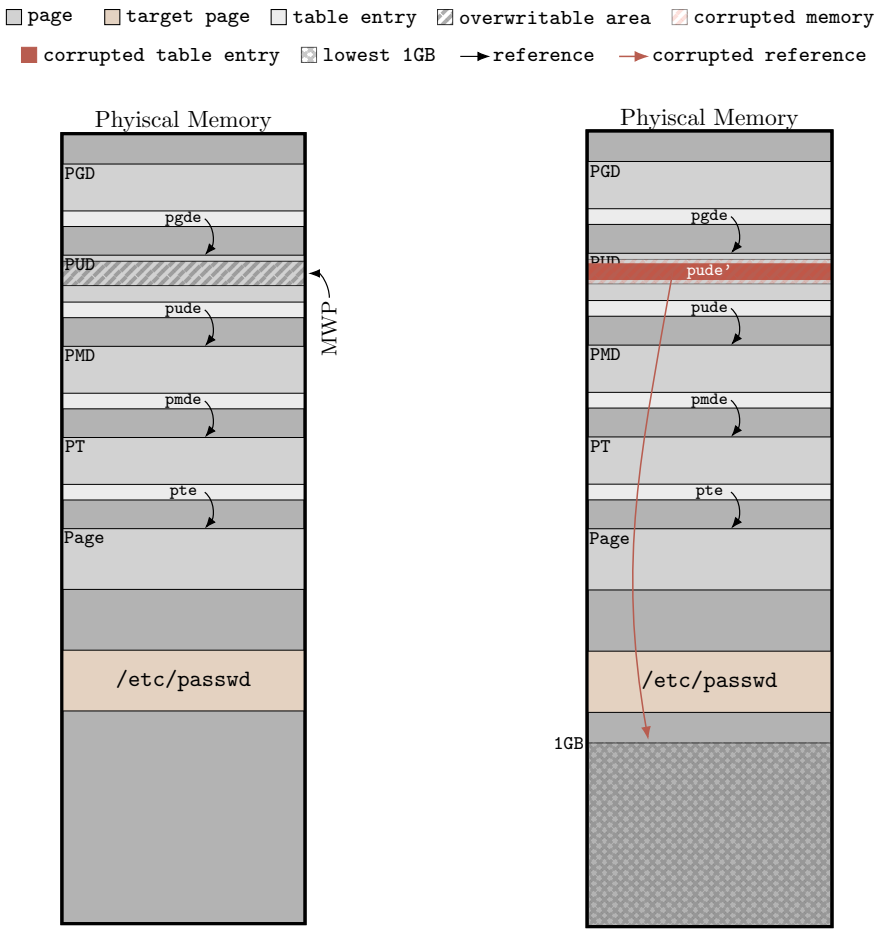
Figure 7.7: The execution timeline of persistent code patterns 1 (**P1**) and 2 (**P2**), where `timed alloc` performs persistent allocations combined with our timing side channel. `kmalloc` allocates the memory slot referenced by the dangling pointer. Since this slot resides on the page table after recycling and reclamation, `copy_from_user` overwrites page table entries.

allowing us to perform the recycling and reclaiming in between. Besides FUSE, SLUBStick can leverage the slow page fault approach [20].

We show **P2** with its execution timeline in Figure 7.7, where thread B's page fault starts before recycling and ends after reclaiming. The hourglass illustrates this time window extension, e.g., with FUSE. Hence, SLUBStick successfully overwrites entries in the reclaimed page table.

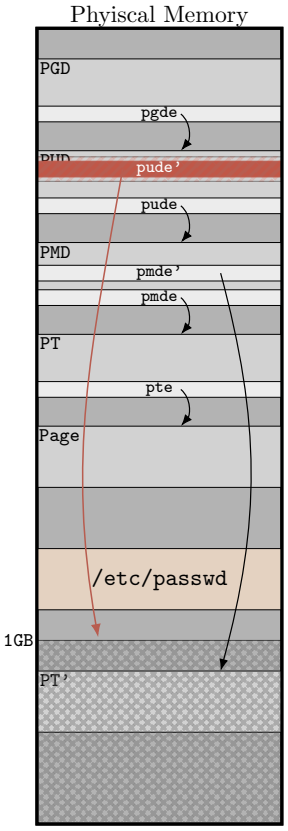
Temporal code pattern. Beyond persistent patterns, SLUBStick can also utilize temporal ones, as exemplified in Listing 7.5. This approach demands the management of two specific timing windows. The first ensures that object allocation at Line 3 and subsequent overwriting at Line 4 occur during the intended phases. The second window is crucial to avert premature object deallocation, leading to a potential failure scenario since the object is concurrently accessed as a page table. To extend these time windows, SLUBStick uses FUSE files as described for persistent pattern 2.

Systematic analysis. We use CodeQL [15] to identify suitable code patterns. We detail this approach in Section 10.3.4. Our results for Linux kernel versions v5.19 and v6.2 are shown in Table 7.9. They indicate that persistent code pattern 1 is with 3 instances (and limited sizes) rare. For persistent code pattern 2, we identify 5 instances with sizes that cover generic caches from `kmalloc-8` to `kmalloc-4096`. A similar stands true

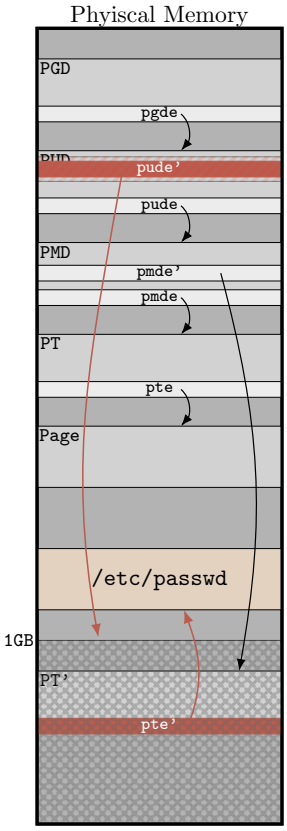


(a) SLUBStick has an MWP to a `PUD` page used for a user address translation. It aims to tamper with the targeted page containing data of `/etc/passwd`.

(b) SLUBStick leverages the MWP to write `pude'` (i.e., `pfn=0`, `size=1`, and `user=1`) to the `PUD` page. As a result, user addresses using `pude'` now reference the first GB of the physical memory.



(c) SLUBStick maps alot of page tables, till a page table (i.e., `PT'`) is mapped within the first GB of the physical memory.



(d) SLUBStick uses first GB mapping to tamper with the entry `pte'` of page table `PT'`. As a result, the user address using `pte'` now references an arbitrary page, e.g., `/etc/passwd`.

Figure 7.8: Converting an MWP to an arbitrary read-and-write primitive to find and tamper with the targeted page `/etc/passwd`.

for temporal code patterns with 7 identified. These findings underline the portability of the existing patterns.

6. Arbitrary Memory Read-and-Write

This section describes the *three-stage* process to obtain an arbitrary memory read-and-write primitive by triggering an MWP once, with an example shown in Figure 7.8. The single triggering of the MWP is crucial since both the persistent pattern 2 and the temporal pattern only allow a single trigger.

Initially, SLUBStick starts with a mapped userspace page using a Page Upper Directory (`PUD`) for address translation, with an MWP associated with it. The goal is to corrupt the content of a targeted page. Figure 7.8a shows an example where the userspace page utilizes the `cr3->pgde->pude->pmde->pte->page` translation, and SLUBStick aims to corrupt the targeted page containing data of `/etc/passwd`. While `/etc/passwd` is the target in this example, any page can be targeted, including kernel code.

In the *first stage* (Figure 7.8b), SLUBStick utilizes the MWP to partially overwrite the `PUD` page with `pude'` entries, each comprising a page frame number of zero and set user-accessible and size bit. As a result, userspace addresses using these entries for the page-table walk (`cr3->pgde->pude'`) are granted read and write access over the first physical GB.

In the *second stage* (Figure 7.8c), SLUBStick maps userspace pages, prompting the kernel to map page tables. It continues this process until a page table (`PT'` in the case of Figure 7.8c) is mapped in the first GB of physical memory. Given SLUBStick's control over reading and writing the first GB, the content of this page table `PT'` can be modified.

In the *third stage* (Figure 7.8d), SLUBStick overwrites an entry in the page table `PT'` with `pte'`. This grants an arbitrary physical memory read and write, using the userspace page with the `cr3->pgde->pude->pmde'->pte'` address translation. Since `kpti` is enabled by default on Ubuntu, SLUBStick uses an invalid syscall to flush this translation from the TLBs and force a page-table walk. Other methods, such as using a TLB eviction set referring to zero pages, would also work. With this primitive, SLUBStick can locate the targeted page and tamper with its content to

escalate privileges, e.g., adding a privileged user with no authentication required.

This example uses the MWP to overwrite a PUD page but can also be used to overwrite a Page Middle Directory (PMD) page. By tampering with a PMD page, SLUBStick has an access space to find `PT'` of 2 MB instead of 1 GB.

In fact, we can use our MWP to overwrite not just a single page table entry but the entire memory slot previously used by the object. For instance, the function `netlink_sendmsg` allows an adversary to overwrite `len` (see Line 5 of Listing 7.4) bytes of the page table page. Thus, to increase the access space after the first stage to find `PT'`, SLUBStick overwrites the page table with entries of increasing page frame number. This approach increases the access space to $\frac{\texttt{sizeof(cache)}}{8} \cdot 1\,\text{GB}$ and $\frac{\texttt{sizeof(cache)}}{8} \cdot 2\,\text{MB}$ for PUD and PMD, respectively.

Reliability of obtaining an MWP to a PMD/PUD page. In order to reliably obtain an MWP primitive to a PMD or PUD page, SLUBStick employs the following strategies.

For generic caches from `kmalloc-8` to `kmalloc-256`, SLUBStick maps a single PUD page, as explained in Section 4.2, with an MWP associating the PUD, as follows: Exploiting the LIFO approach inherent to the buddy allocator for returning recycled memory chunks, SLUBStick initiates the mapping of a user page with an unmapped PUD, PMD, and PT. This procedure guarantees the reliable reclamation of the recycled memory chunk for the PUD page.

Conversely, achieving an MWP to a PMD page involves another strategy. For generic caches from `kmalloc-512` to `kmalloc-4096`, where the memory chunk size exceeds a single page, SLUBStick initially prompts the kernel to split the recycled memory chunk. To reliably reclaim the split part of the memory chunk containing the MWP, SLUBStick employs an extensive mapping strategy for PMD pages, accomplished by mapping 2 MB huge pages. This extensive mapping tactic compels the buddy allocator to predominantly allocate PMD pages, including the page associated with the MWP. There are two options for using huge pages: Transparent Huge Pages (THP) or hugetlbfs. Albeit hugetlbfs offers more stable exploitation results, these huge pages must be pre-allocated. However, due to the widely spread application of hugetlbfs [25], such as databases [39], virtualization, and even Java [18], SLUBStick utilizes hugetlbfs.

Table 7.2: Primitives used for our attack evaluation.

Generic Cache	MP	AP	MWP
kmalloc-8	add_key	signalfd_ctx	do_signalfd4
kmalloc-16	add_key	aa_revision	key_ctl_key_verfify
kmalloc-32	add_key	anon_vma_name	key_ctl_key_verfify
kmalloc-64	add_key	snd_ctl_file	key_ctl_key_verfify
kmalloc-96	add_key	vfio_container	key_ctl_key_verfify
kmalloc-128	add_key	dlm_user_proc	key_ctl_key_verfify
kmalloc-192	add_key	pp_struct	key_ctl_key_verfify
kmalloc-256	add_key	snd_compr_file	replace_user_tlv
kmalloc-512	add_key	tls_context	replace_user_tlv
kmalloc-1024	add_key	pipe_buffer	replace_user_tlv
kmalloc-2048	add_key	key.description	replace_user_tlv
kmalloc-4096	add_key	net_device	replace_user_tlv

MP: Measurement Primitive **AP**: Allocation Primitive **MWP**: Memory Write Primitive.

7. Attack Evaluation

In this section, we evaluate the effectiveness of SLUBStick. We initially exploit a synthetic DF vulnerability (see Section 7.1) to show its applicability across generic caches from `kmalloc-8` to `kmalloc-4096`. We then exploit 9 CVEs (see Section 7.2), consisting of DF, UAF, and OOB vulnerabilities.

Evaluation setup. For our experimental setup, we run the exploits in a virtual machine via QEMU 6.2.0, equipped with 4 cores and 16 GB RAM. We ran Ubuntu 22.04 LTS, with Linux kernels v5.19 and v6.2 for x86, and v6.2 for aarch64, to demonstrate architecture and version independence. These kernels were the unmodified generic ones, which presents a considerable challenge for exploitation.

7.1. Synthetic Vulnerability

To evaluate SLUBStick for all generic caches, we create a synthetic DF vulnerability in the Linux kernel, integrated into a module, i.e., read device driver `rdd`. Its simplified code is seen in Listing 7.6. For the exploitation processes, we need this driver code's allocation (`ALLOC`) and release (`FREE`) functionalities. We compile and then include this driver code during the operating system's startup for our three kernels.

With the synthetic DF vulnerability in place, we execute SLUBStick on all generic caches from `kmalloc-8` to `kmalloc-4096`. For the measurement primitive, we use `add_key` with the invalid argument `_descr` (see Section 4.1). As allocation primitive, we utilize objects retrieved via our systematic analysis (see Section 5.2). For the MWP, we accomplish the exploitation with all three types, persistent 1 and 2 as well as temporal, using `do_signalfd4`, `replace_user_tlv`, and `key_ctl_key_verify`, respectively. Table 7.2 shows all primitives used for this evaluation. Equipped with these primitives, we successfully execute SLUBStick, exploiting the DF for privilege escalation.

7.2. Real-World Vulnerabilities

In line with prior research [6, 29, 47, 50, 54], we evaluate SLUBStick with a selection of real-world vulnerabilities. We port these vulnerabilities to our kernels to demonstrate the version, architecture, and kernel binary independence of SLUBStick. Our evaluation includes 9 vulnerabilities that are systematically classified based on the capability they provide. If the root cause of a vulnerability is a race condition that allows an adversary to derive a kernel heap vulnerability, we classify it as a heap vulnerability. For instance, since both race conditions, CVE-2023-21400 and CVE-2022-29582, can be pivoted to a UAF and DF, respectively, we classify them as a UAF and DF. In total, our selection consists of 3 DF vulnerabilities that enable SLUBStick directly, as well as 3 OOB and 3 UAF that we pivot to achieve a DF state.

The feasibility of pivoting an OOB and UAF depends on the write capability provided. We demonstrate with our results that the capability to overwrite as low as two bytes is enough to pivot to a DF state. Examples include corrupting a reference counter (as seen in CVE-2022-29582 and CVE-2023-3609) or nullifying the two least significant bytes of a pointer (like in CVE-2022-27666). For pivoting, SLUBStick requires an exploitable object containing a data pointer or reference counter at the overwrite point. The purely manual identification of suitable objects is time-consuming due to the extensive nature of the Linux kernel. For this reason, we utilize CodeQL to assist in identifying suitable objects for OOB and UAF write vulnerabilities, described in Section 10.3.1.

Table 7.3: Exploitability shown on real-world vulnerabilities.

CVE	Capability	Cache
CVE-2023-21400	DF	kmalloc-32
CVE-2023-3609	UAF	kmalloc-96
CVE-2022-32250	UAF	kmalloc-64
CVE-2022-29582	UAF	files_cachep
CVE-2022-27666	OOB	kmalloc-4096
CVE-2022-2588	DF	kmalloc-192
CVE-2022-0995	OOB	kmalloc-96
CVE-2021-4157	OOB	kmalloc-64
CVE-2021-3492	DF	kmalloc-4096

Exploitability results. Table 7.3 shows the exploitability of SLUBStick, showcasing its functionality across kernel heap vulnerabilities and cache sizes. Specifically, we can free objects twice for the CVE2023-21400, CVE-2022-2588, and CVE-2021-3492 vulnerabilities, allowing SLUBStick to exploit these vulnerabilities and directly escalate privileges.

Additionally, the CVE-2022-0995 vulnerability allows overwriting out-of-bounds originating from the `watch_queue` object. We exploit this write capability to corrupt the reference counter of an `anon_vma_name` object, both located within the same generic cache, i.e., `kmalloc-96`. With the corrupted reference, we create a DF state of an `anon_vma_name` object, which enables the execution of SLUBStick. For CVE-2022-27666, the OOB allows overwriting memory adjacent to the `page_frag` object (allocated from generic cache `kmalloc-4096`). We perform a cross-cache overflow to zero the `next` pointer's two least significant bytes of a `msg_msg` object positioned on the adjacent memory chunk. This allows us to free the `next` twice so we can execute SLUBStick. As a third OOB write vulnerability, CVE-2021-4157, we exploit the overwriting capability within `kmalloc-64` to tamper with the reference counter `kref` of an `eventfd_ctx` object. As a result, we can pivot this to a DF and perform SLUBStick.

Furthermore, the CVE-2022-29582 vulnerability allows a `file` UAF, specifically using the `file` object after it has been invalidly freed. By strategically pivoting this vulnerability using cross-cache, we can reclaim the memory chunk for the generic cache `kmalloc-256`. Subsequently, we reclaim the invalidly freed `file` object to corrupt its reference counter. This allows us to free the `file` a second time, creating a DF state to run SLUBStick. Similarly, CVE-2023-3609 allows to use a `tc_u_hnode` object after it has

been freed. Reclaiming the memory within the same generic cache and overwriting the reference counter, we pivot this vulnerability to a DF state of `tc_u_hnode` to perform SLUBStick. Lastly, the CVE-2022-32250 vulnerability provides a write capability within the generic cache `kmalloc-64`, which we use to corrupt a data pointer within `fdtable` for a DF state.

8. Discussion

Impact on existing attacks. Our side-channel supported approach presented in Section 4 greatly enhances the reliability of cross-cache attacks from generic caches and makes them practical for exploitation. Thus, it amplifies the effectiveness of exploitation methods employing cross-cache attacks.

For instance, DirtyCred [29] relies on the cross-cache approach to exploit DF vulnerabilities. This exploitation entails pivoting DF vulnerabilities to trigger an invalid free on a credential object. With our enhancement of the cross-cache attack, DirtyCred's success rate in pivoting DF vulnerabilities from generic caches also significantly increases, making DirtyCred an even more significant threat to kernel security.

Moreover, SLUBStick is more versatile than DirtyCred as it does not rely on an invalid free on a credential object. Instead, an invalid free operation on any object is sufficient.

Container escape. In addition to privilege escalation, our research demonstrates that SLUBStick can directly enable container escapes such as Docker. For containers permitting FUSE, SLUBStick uses FUSE to extend the time window in persistent code pattern 2 and the temporal pattern. For (hardened) containers prohibiting FUSE, SLUBStick uses persistent code pattern 1 or the slow page fault approach [20] instead of FUSE. It then modifies kernel code accessible from userspace via a syscall, i.e., `sys_setresuid`.

Noise. With SLUBStick, we present a reliable approach for privilege escalation, leveraging a software timing side channel on the SLUB's object allocation. To maximize the reliability of SLUBStick, we minimize noise as follows:

CPU migration, the process of moving a task to another CPU, can introduce instabilities as both the slab and buddy allocator maintain per-CPU

lists. To mitigate migration and, hence, increase reliability, SLUBStick may pin all running processes used in exploitation to a single CPU. However, it is not essential as illustrated in Table 7.1.

Due to the preemptive nature of the Linux kernel, preemption may occur during the timing measurement of a measurement primitive, potentially corrupting the result. To limit this noise source, SLUBStick performs timing measurements and keeps track of the number of objects associated with the current slab. If SLUBStick observes slow timing, indicating the allocation of a new memory chunk, it validates this by checking the object number of the current slab. This validation prevents misinterpretation of slow allocations, ensuring that previous and current objects reside in the same slab.

Manual vs automated components. We use the CodeQL [15] static analyzer to generate a list of possible objects or primitives automatically. Through manual inspection, we then refine this list by selecting suitable candidates. To provide a representation of the effectiveness of our automated part, our evaluated kernel contains 66 787 structures, where our automated tool outputs 8 601 potential as allocation and memory write primitives. After manual inspection, we get 36 suitable structures for allocation primitives. For memory write primitives, the tool outputs 2 046 copy locations, where manual inspection yields 15 memory write primitives. For measurement primitives, the tool outputs 195 primitives, with 14 after inspection. Tables 7.6 to 7.9 show our findings. Further details on this process are provided in Section 10.3.

Cross version/architecture dependencies. Kernel exploitation often depends on the specific kernel version and system architecture, as various exploitation techniques are closely tied to these factors. For example, KASLR is bypassed either with a read primitive [5] or a hardware side channel [3]. Read primitives are closely tied to specific kernel versions and configurations, while hardware side channels depend on the system architecture. Additionally, numerous exploits rely on constructing ROP chains [49, 55], which involves a detailed inspection of the kernel binary, a process connected with system architecture, Linux kernel version, and configuration. SLUBStick distinguishes itself by not relying on bypassing KASLR or constructing a ROP chain, nor does it use architecture-specific data. As a result, SLUBStick is resilient to variations in kernel versions and architecture dependencies.

Kernel defenses. The last decade has seen a surge of proposals to improve kernel security. We briefly discuss the defenses, particularly considering SLUBStick.

KASLR is designed to enhance kernel security by randomizing the memory layout, making it more challenging to exploit vulnerabilities to perform Code Reuse Attacks (CRA) and data-only attacks. To further complicate CRA, researchers have introduced Control-Flow Integrity (CFI) [1], which has been adapted to kernel software [10, 14, 33, 46]. CFI helps ensure the integrity of the kernel control flow by restricting it to an approximated control-flow graph. Moreover, the Linux kernel has incorporated various hardening strategies to make the allocator more resistant to memory corruption vulnerabilities. These include slab list randomization, protection of slab meta-data, and slab quarantine [41]. However, researchers have identified bypassing attacks [26, 41, 54] for these hardening strategies. Additionally, heap separation is a defense integrated into the Linux kernel to separate caches containing security-critical data, e.g., `cred`, or objects often used for exploitation [29, 40, 56], e.g., `msg_msg`. To further enhance the separation of kernel objects, researchers [9] have proposed to increase the separation granularity. This involves randomly assigning each allocation site to one of these separated caches, making heap spraying attacks more challenging. However, SLUBStick leverages common code patterns, allowing it to circumvent the separation attempt of generic caches. Going a step further, our timing side channel on the allocator can even be utilized to determine whether two allocation sites share the same cache. In summary, the described countermeasures, while valuable, appear ineffective in mitigating SLUBStick.

AUTOSLAB [28] is a defense provided by grsecurity via paid subscription. Since AUTOSLAB separates allocator caches based on their allocation types, it restricts that different allocation types share the same allocation cache. However, this granularity is still too coarse-grained, as, e.g., the type `char *` is used by multiple allocation, measurement, and memory write primitives. As a result, the allocation cache with type `char *` is still exploitable with SLUBStick.

SLAB_VIRTUAL [43], which is currently under development, is designed to mitigate cross-cache attacks by allocating kernel objects via virtual addresses rather than direct physical memory. However, Torvalds and Molnar noted that it has drawbacks like significant overhead and incompatibility with DMA [43]. If merged, DMA-allocated memory will most

likely be excluded [43]. This leaves more than 350 allocation sites unprotected, exploitable to obtain all necessary code patterns (allocation, measurement, and memory write primitives, such as `sun8i_ce_aes_setkey`, `monwrite_new_hdr`, and `sti_hqvdp_vtg_cb`). Hence, SLUBStick can exploit a vulnerability, e.g., CVE-2023-2194, of a DMA-allocated object. Since DMA-allocated memory is mainly used in drivers, known to be vulnerable [35], SLUBStick still poses a threat.

Lastly, prior academic works [11, 34, 42, 45] proposed defenses to protect page tables to mitigate data-only attacks.

While existing defenses such as AUTOSLAB and SLAB_VIRTUAL demonstrate promise in mitigating SLUBStick, none provide comprehensive protection. This underscores the threat SLUBStick poses to kernel security.

Public exploits. Recently, multiple non-academic exploits have been proposed [2, 12, 13, 17, 19, 30, 48, 53]. Some [2, 17, 19, 48] leverage cross-cache attacks using the dedicated `file` cache. In contrast, we focused on generic caches, which are considerably more challenging to exploit. This is primarily because generic caches have numerous allocation sites, resulting in significant allocation noise. Specifically, while the dedicated `file` cache has one allocation site (`dup_f`), generic caches have from 354 (`kmalloc-4096`) to 2250 (`kmalloc-64`) on systems like our evaluated Ubuntu. Hence, we proposed our side-channel supported approach to perform cross-cache attacks on generic caches reliably.

Two exploits, bad io_uring [30] and the exploit demonstrated by Wu et al. [48], highlight the exploitation of cross-cache attacks on older kernel versions (i.e., v5.10), targeting Android devices. Bad io_uring leverages a cross-cache attack to misuse `pipe_buffer` as an arbitrary read and write. Wu et al. (concurrently to our work) demonstrated page-table manipulation by misusing `signalfd_ctx`. Both exploits rely on stabilization objects in conjunction with multiple retriggers of cross-cache reuse. Specifically, bad io_uring utilized additional `pipe_buffer` objects for stabilization, while Wu et al. relied on `seq_operations`. However, kernel advances beyond v5.14 introduced heap separation (i.e., `kmalloc-cg-*`), rendering the reuse of these stabilization objects ineffective for exploitation. Consequently, these exploitation strategies are thwarted by newer kernel versions. In contrast, SLUBStick exploits more recent systems, including v5.19 and v6.2, for a wide variety of heap vulnerabilities.

9. Conclusion

This paper presented SLUBStick, a novel kernel exploit technique that enables arbitrary memory read-and-write primitives through a practical software cross-cache attack. For our cross-cache attack, we used a software timing side channel on the SLUB allocator, significantly enhancing the success rate for frequently used generic caches to over 99 %. Moreover, using page-table manipulation, SLUBStick effectively converts a limited kernel heap vulnerability into arbitrary read-and-write capabilities. We demonstrated privilege escalation in the Linux kernel using a synthetic vulnerability and 9 real-world CVEs, showcasing its serious threat.

Acknowledgements

We thank Daniel Gruss, the anonymous reviewers, and our shepherd for their valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE and AWARE project (FFG grant numbers 888087 and 891092), the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Any opinions, findings, 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 (p. 241).

[2] Awarau and pql. CVE-2022-29582 An io_uring vulnerability. 2022. URL: https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/ (pp. 208, 216, 227, 242).

[3] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 217, 240).

[4] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In: USENIX Security. 2020 (p. 212).

[5] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In: CCS. 2020 (pp. 211, 212, 240).

[6] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In: CCS. 2019 (pp. 228, 237).

[7] Jonathan Corbet. A slab allocator (removal) update. May 2023. URL: https://lwn.net/Articles/932201/ (p. 210).

[8] Jonathan Corbet. Blocking userfaultfd() kernel-fault handling. May 2020. URL: https://lwn.net/Articles/819834/ (p. 230).

[9] Jonathan Corbet. Randomness for kmalloc(). July 2023. URL: https://lwn.net/Articles/938637/ (pp. 219, 241).

[10] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In: S&P. 2014 (p. 241).

[11] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In: NDSS. 2017 (p. 242).

[12] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel. 2022. URL: https://syst3mfailure.io/corjail/ (p. 242).

[13] ETenal. CVE-2022-27666: Exploit esp6 modules in Linux kernel. 2022. URL: https://etenal.me/archives/1825 (pp. 208, 242).

[14] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: RAID (2023) (p. 241).

[15] GitHub. CodeQL. 2021. URL: https://codeql.github.com/ (pp. 224, 231, 240, 249).

[16] Luke Gix. FUSE for Linux Exploitation 101. 2022. URL: https://exploiter.dev/blog/2022/FUSE-exploit.html (p. 230).

[17] h0mbre. Escaping the Google kCTF Container with a Data-Only Exploit. 2023. URL: https://h0mbre.github.io/kCTF_Data_Only_Exploit%5C# (pp. 208, 216, 242).

[18] IBM. Working with hugetlbfs huge-page support. 2023. URL: https://www.ibm.com/docs/en/linux-on-z?topic=hps-working-huge-pages-3 (p. 235).

[19] javierprtd. No CVE for this bug which has never been in the official kernel. 2023. URL: https://soez.github.io/posts/no-cve-for-this.-It-has-never-been-in-the-official-kernel/ (pp. 208, 216, 227, 242).

[20] Choo Yi Kai. A new method for container escape using file-based DirtyCred. 2023. URL: https://starlabs.sg/blog/2023/07-a-new-method-for-container-escape-using-file-based-dirtycred/ (pp. 230, 231, 239).

[21] The Linux Kernel. FUSE. 2023. URL: https://www.kernel.org/doc/html/next/filesystems/fuse.html (p. 230).

[22] Imran Khan. Linux SLUB Allocator Internals and Debugging. 2022. URL: https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-1 (pp. 210, 212).

[23] Kenneth C Knowlton. A fast storage allocator. In: Communications of the ACM (1965) (p. 210).

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 217).

[25] Mike Kravetz. hugetlbfs: Not just for databases anymore! 2015. URL: https://blogs.oracle.com/linux/post/hugetlbfs-not-just-for-databases-anymore/ (p. 235).

[26] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In: USENIX Security. 2023 (pp. 212, 221, 226, 228, 241).

[27] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting Kernel Races through Raising Interrupts. In: USENIX Security. 2021 (p. 228).

[28] Zhenpeng Lin. How AUTOSLAB Changes the Memory Unsafety Game. 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game (pp. 208, 216, 219, 241).

[29] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In: CCS. 2022 (pp. 208, 216, 230, 237, 239, 241).

[30] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android. 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf (pp. 208, 242).

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security. 2018 (p. 217).

[32] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (p. 205).

[33] Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In: AsiaCCS. 2024 (p. 241).

[34] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (p. 242).

[35] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In: NDSS. 2022 (p. 242).

[36] Paul McKenney. What is RCU, Fundamentally? Dec. 2007. URL: https://lwn.net/Articles/262464/ (p. 222).

[37] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In: Bluehat IL (2019) (p. 217).

[38] Joao Moreira. Kernel FineIBT Support. Apr. 2022. URL: https://lwn.net/Articles/891976/ (pp. 208, 217).

[39] MySQL. Enabling Large Page Support. 2023. URL: https://dev.mysql.com/doc/refman/8.0/en/large-page-support.html (p. 235).

[40] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html (pp. 227, 230, 241).

[41] Alexander Popov. Linux kernel heap quarantine versus use-after-free exploits. 2020. URL: https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html (p. 241).

[42] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In: S&P. 2020 (p. 242).

[43] Matteo Rizzo and Jann Horn. Prevent cross-cache attacks in the SLUB allocator. 2023. URL: `https://lore.kernel.org/linux-mm/202309151425.2BE59091@keescook/T/` (pp. 241, 242).

[44] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat USA. 2015 (p. 217).

[45] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In: NDSS. 2016 (p. 242).

[46] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In: USENIX Security. 2022 (p. 241).

[47] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects. In: USENIX Security. 2023 (p. 237).

[48] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. 2023. URL: `https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html` (pp. 208, 216, 242).

[49] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: USENIX Security. 2019 (p. 240).

[50] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In: USENIX Security. 2018 (p. 237).

[51] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (pp. 208, 216, 219, 226, 228).

[52] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (p. 217).

[53] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features. 2018. URL: https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf (pp. 227, 242).

[54] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In: USENIX Security. 2022 (pp. 212, 237, 241).

[55] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In: CCS. 2023 (p. 240).

[56] Xiaochen Zou and Zhiyun Qian. Exploit esp6 modules in Linux kernel. 2022. URL: https://etenal.me/archives/1825 (pp. 230, 241).

10. Appendix

10.1. Generic Cache Information

To trigger the recycling process reliably, we provide insights (see Table 7.4) into how a generic cache manages memory chunks and stores available objects. Generic caches from `kmalloc-8` to `kmalloc-256` use one page per slab, with the slab storing 512 to 16 objects per memory chunk. Caches larger than `kmalloc-256` use multiple pages per slab, storing between 16 and 8 objects per chunk. The **Node Partial Slab List Capacity** column indicates when the SLUB allocator releases the chunks of the slabs, e.g., for `kmalloc-256`, if this cache reaches 6 free slabs, the SLUB allocator prompts the buddy allocator to discard and recycle the slab's chunks.

Table 7.4: Detailed information of each generic cache, where ✶ denotes the number of slabs until the node partial slab list is full, prompting to discard the slabs' memory chunks.

Generic Cache	Memory Chunk	Number of Pages	Number of Objects	Node Partial Slab List Capacity✶
kmalloc-8	4 096	1	512	6
kmalloc-16	4 096	1	256	6
kmalloc-32	4 096	1	128	6
kmalloc-64	4 096	1	64	8
kmalloc-96	4 096	1	42	12
kmalloc-128	4 096	1	32	8
kmalloc-192	4 096	1	21	12
kmalloc-256	4 096	1	16	7
kmalloc-512	8 192	2	16	6
kmalloc-1024	16 384	4	16	6
kmalloc-2048	32 768	8	16	6
kmalloc-4096	32 768	8	8	6

10.2. Timings of Measurement Primitives

We perform experiments to verify that the slow allocation time ⑤ is higher than the fast allocation time ① (see Figure 7.2). We use `add_key` as the measurement primitive with an invalid `_descr` argument to fulfill the constraints described in Section 4.1, while others (e.g., `mount` with an invalid `dev_name` address) yield similar results. Our experimental environment is Ubuntu 22.04 LTS with a Linux kernel v6.2, running on a machine with Intel i7-1260P and 48 GB RAM. We allocate 16384 objects as a warm-up to ensure that the measured timing of the subsequent allocations is either from the CPU free list ① or from a new memory chunk using the buddy allocator ⑤. We then perform 4096 allocations and distinguish them between **Fast** and **Slow Allocation**. As shown in Table 7.5, the results demonstrate a notable timing difference between these allocation paths, with an average fast allocation from 1086 to 1925 timestamps and a minimum slow allocation from 4401 to 3758.

10.3. Systematic Analysis Detailed

In this section, we present our systematic analysis in more detail. Our principal approach is first to identify a list of possible objects or primitives using the CodeQL [15] static analyzer. We then use these results to manually identify suitable results or discard those results that violate the constraints of the respective primitive. For each primitive, our systematic

Table 7.5: Measured timing in timestamps for fast ① and slow ⑤ allocation (see Figure 7.2) using the measurement primitive `add_key` with an invalid `_descr` argument.

Generic Cache	Fast Allocation ①		Slow Allocation ⑤
`kmalloc-8`	1 086	±53	> 4401
`kmalloc-16`	1 114	±69	> 3224
`kmalloc-32`	1 133	±58	> 2510
`kmalloc-64`	1 130	±69	> 2468
`kmalloc-96`	1 117	±44	> 2169
`kmalloc-128`	1 250	±126	> 2470
`kmalloc-192`	1 197	±86	> 2121
`kmalloc-256`	1 359	±34	> 2439
`kmalloc-512`	1 685	±84	> 3759
`kmalloc-1024`	1 601	±30	> 3589
`kmalloc-2048`	1 561	±23	> 3415
`kmalloc-4096`	1 925	±33	> 3758

analysis results in a comprehensive list of suitable objects and functions, demonstrating the effectiveness of our approach.

CodeQL aims to find code patterns that cause vulnerabilities in software. At its core, it creates a database where essential meta-information about the examined software is stored. With this database, CodeQL uses queries as input to analyze the software and interpret the query results. We retrofit CodeQL to find allocation and measurement primitives (see Section 4.1). We also use CodeQL to find suitable victim objects for UAF and OOB write vulnerabilities (see Section 5.1) and suitable code patterns to exploit for an MWP (see Section 5.2).

10.3.1. Finding Suitable Objects to Pivot UAF and OOB Writes

To find suitable objects for pivoting UAF and OOB vulnerabilities with an overwriting capability, we need a victim object with a pointer to a dynamically allocated object or a reference counter at the location of the overwriting location. In the example of CVE-2022-32250, the UAF write vulnerability provides overwriting at offset 0x18 for objects allocated from the generic cache `kmalloc-64`. With these constraints, we use a CodeQL query to help find dynamically allocated objects with either a pointer or

```
typedef struct { size_t uaddr, size; } msg_t;
void *obj;
long rd_ioctl(struct file *_, unsigned num, size_t param) {
  msg_t msg;
  copy_from_user(&msg, param, sizeof(msg_t));
  switch (num) {
    case ALLOC:
      obj = kmalloc(msg->size);
      return 0;
    case FREE:
      kfree(obj);
      return 0;
    case READ:
      copy_to_user(msg->uaddr, obj, msg->size);
      return 0;
    default:
      return -1;
  }
}
```

Listing 7.6: Read device driver `rdd`, supporting an allocation (`ALLOC`), deallocation (`FREE`), and read (`READ`).

reference counter at an offset of 0x18. We find the `fdtable` object that satisfies these constraints with the pointer `open_fds` at an offset 0x18.

Our query takes the overwrite offset and object size as input and returns all matching objects available in the Linux kernel with default kernel configurations. From these possible objects, we manually identify a suitable object accessible from userspace as an unprivileged user, e.g., the `fdtable` for the CVE-2022-32250 vulnerability.

10.3.2. Finding Allocation Primitives

For allocation primitives, our CodeQL query identifies every persistent object allocation code location grouped by object size. From this list, we manually filter out those inaccessible from userspace as unprivileged users. The result is a list of fixed and variable-size objects, shown in Tables 7.6 and 7.7.

Table 7.6: Allocation primitives allocating a single fixed-size object during the syscall, with ✶ new objects identified.

Generic Cache	Object	Constraints
kmalloc-8	pci_filp_private✶ signalfd_ctx	
kmalloc-16	afs_file✶ aa_revision✶	
kmalloc-32	vmci_host_dev✶ seq_operations coda_file_info✶ shm_file_data	cg cache
kmalloc-64	snd_info_private_data✶ snd_ctl_file✶	
kmalloc-96	subprocess_info watch_queue vfio_container✶	
kmalloc-128	dlm_user_proc✶	
kmalloc-192	loopback_pcm✶ snd_timer_user✶ pp_struct✶	
kmalloc-256	vhci_data✶ snd_compr_file✶ msg_queue	 cg cache
kmalloc-512	tls_context mousedev_client✶	 input group
kmalloc-1024	pipe_buffer tty_struct sock xfrm_policy nouveau_cli✶	
kmalloc-2048	super_block perf_event✶	 SELinux disabled
kmalloc-4096	net_device	

10.3.3. Finding Measurement Primitives

For measurement primitives, any code snippet that copies user data into a dynamically allocated buffer and then performs validation checks can be leveraged. This includes kernel functions that use `strndup_user`, `memdup_-`

Table 7.7: Allocation primitives allocating a single variable-size object during the syscall, with ✶ new objects identified.

Elastic Object	Generic Caches	Constraints
user_key_payload	kmalloc-[32,32767)	only 200 allocation
anon_vma_name✶	kmalloc-[8,96)	
msg_msg	kmalloc-[64,4096)	cg cache
msg_msgseg	kmalloc-[8,4096)	cg cache
drm_property_blob	kmalloc-[96,INT_MAX)	
key.description	kmalloc-[8,4096)	

user and memdup_user_nul. Therefore, we use a CodeQl query to obtain all the code locations where these functions are called. Next, we manually validate, e.g., by running test programs to execute the syscall that executes the function found, that these code locations are accessible from userspace as an unprivileged user. In the example of the add_key syscall, we execute this syscall with a _desc where the first byte is a '.'. All other conditions with additional information to execute the measuring primitives can be found in Table 7.8.

10.3.4. Finding Memory Write Primitives

For memory write primitives, our queries obtained all code locations that execute copy_from_user. With all these locations, we search for our three distinct code patterns, i.e., persistent code patterns 1 and 2, and temporal code patterns. We then filter out those that violate constraints, e.g., timed_alloc shown in Listing 7.2. As a result, we obtain functions that can be used as MWP, shown in Table 7.9.

Table 7.8: Measurement primitives, where ✶ denotes allocation via the separated `kmalloc-cg-*` caches and ☆ denotes depending allocation size whether `msg_msg`/`msg_msgseg` is allocated.

Syscall	File	Argument	Allocation Size	Condition
add_key	security/keys/keyctl.c	_descr	[1,4096)	*(char *)_descr = '.'
request_key	security/keys/keyctl.c	_descr	[1,4096)	_callout_info bad address
keyctl$KEYCTL_JOIN_SESSION_KEYRING	security/keys/keyctl.c	arg2	[1,4096)	*(char *)arg2 = '.'
keyctl$KEYCTL_SEARCH	security/keys/keyctl.c	arg4	[1,4096)	arg2 invalid keyid
keyctl$KEYCTL_PKEY_QUERY	security/keys/keyctl.c	arg5	[1,4096)	arg2 invalid keyid
mount	fs/namespace.c	type	[1,4096)	dev_name bad address
fsopen	fs/fsopen.c	_fs_name	[1,4096)	_fs_name not existing
fsl_hv_ioctl$FSL_HV_IOCTL_SETPROP	drivers/virt/fsl_hypervisor.c	*(size_t *)arg	[1,4096)	*((size_t *)arg+1) bad address
perf_ioctl$PERF_EVENT_IOC_SET_FILTER	kernel/events/core.c	arg	[1,4096)	!has_addr_filter(event)
joydev_ioctl_common$JSIOCSAXMAP	drivers/input/joydev.c	argp	[64,UINT64_MAX)	*(char *)argp > 0x3f
fsconfig$FSCONFIG_SET_FD	fs/fsopen.c	_key	[1,256)	fget(aux) not existing
prctlPR_SET_VMAPR_SET_VMA_ANON_NAME	kernel/sys.c	arg5	[1,80)	*(char *)arg5 = 1
io_uring_register$IORING_REGISTER_RESTRICTIONS	fs/io_uring.c	arg	[16,UINT32_MAX)	*(short *)arg = 5
msgsnd$alloc_msg✶	ipc/msgutil.c	len	[64/8☆,4096)	mtext bad address

Table 7.9: Code patterns allowing for an MWP, where ✶ denotes the same function for allocation and triggering the MWP of the persistent object, and ☆ and ✢ denote distinct function for allocation and triggering the MWP.

Function	Type	Size	Constraints
ipmi_open☆/ipmi_ioctl✢	P1	8	
do_signalfd4✶	P1	8	
joydev_ioctl✶	P1	5680	
replace_user_tlv	P2	[1,131072)	
atmel_ioctl	P2	[1,32)	CAP_NET_ADMIN
netlink_sendmsg	P2	[1,INT_MAX)	
tun_sendmsg	P2	[1,INT_MAX)	
tap_sendmsg	P2	[1,INT_MAX)	
mount	T	[1,4096)	
key_ctl_key_verfify	T	[1,256)	
mtdchar_writeoob	T	[1,4096)	
mmc_blk_ioctl_copy_from_user	T	96	
ptp_ioctl	T	1216	
__cld_pipe_inprogress_downcall	T	[1,65536)	
__bpf_copy_key	T	[1,512)	bpf as unprivileged

P1/2: Persistent code pattern 1/2 T: Temporal code pattern.

8

Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels

Publication Data

Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024

Contributions

The author of this thesis is the main author of this work. The author's contributions are *one-day exploitation insights*, *defense insights*, *defense inclusion and effectiveness analysis*, and *novel findings*, as well as most of the written text.

Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels

Lukas Maar[1] Florian Draschbacher[1,2] Lukas Lamster[1]
Stefan Mangard[1]

[1] Graz University of Technology [2] A-SIT Austria

Abstract

With the mobile phone market exceeding one billion units sold in 2023, ensuring the security of these devices is critical. However, recent research has revealed worrying delays in the deployment of security-critical kernel patches, leaving devices vulnerable to publicly known one-day exploits. While the mainline Android kernel has seen an increase in defense mechanisms, their integration and effectiveness in vendor-supplied kernels are unknown at a large scale.

In this paper, we systematically analyze publicly available one-day exploits targeting the Android kernel over the past three years. We identify multiple exploitation flows representing vulnerability-agnostic strategies to gain high privileges. We then demonstrate that integrating defense-in-depth mechanisms from the mainline Android kernel could mitigate 84.6 % of these exploitation flows. In a subsequent analysis of 994 devices, we reveal a widespread absence of effective defenses across vendors. Depending on the vendor, only 28.8 % to 54.6 % of exploitation flows are mitigated, indicating a 4.62 to 2.95[1] times worse scenario than the mainline kernel.

Further delving into defense mechanisms, we reveal weaknesses in vendor-specific defenses and advanced exploitation techniques bypassing defense implementations. As these developments pose additional threats, we discuss potential solutions. Lastly, we discuss factors contributing to the absence of effective defenses and offer improvement recommendations. We envision that our findings will guide the inclusion of effective defenses, ultimately enhancing Android security.

[1]Factors of $\frac{1-0.288}{1-0.846}$ and $\frac{1-0.546}{1-0.846}$, respectively.

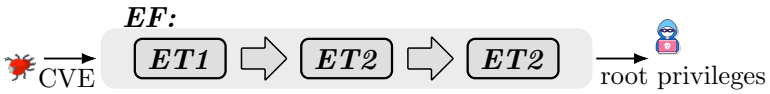


Figure 8.1: The exploitation flow ***EF*** is a vulnerability-agnostic chain of exploitation techniques ***ET***, with one ***ET*** elevating a primitive to a more powerful form [7]. ***EF*** leverages the capabilities of a vulnerability to gain root privileges ultimately.

1. Introduction

Over the past decade, the mobile phone market has reached an all-time high, with more than one billion units sold in 2023. Given our daily reliance on mobile phones for communication, financial transactions, and personal data storage, this surge in device adoption underscores the critical need for robust security measures protecting these devices.

Despite the importance of mobile security, recent studies [13, 27, 44, 62, 68] have revealed that Android's security-critical kernel patches often lag significantly behind the mainstream Linux kernel. In over 20 % of cases, delays exceeding one year occur [62], mainly due to the downstream approach of most Android vendors. This delayed deployment of security-critical patches creates opportunities for malicious actors to attack the Android Linux kernel. While these attacks would be classified as one-day exploits due to the known nature of their vulnerabilities, they effectively function as zero-day exploits during the extensive unpatched period. The severity of this situation is underscored by findings from Google Project Zero [9, 47] and Threat Analysis Group [51], which highlight a prevalence of exploits in the wild targeting these unpatched vulnerabilities in the Android kernel.

On the defensive side, the mainline Android kernel has seen an increase in vulnerability-agnostic defenses preventing one-day exploits. While these defense-in-depth mechanisms may be readily available, *their integration and effectiveness in vendor-supplied kernels are unknown.* Consider, for example, the case of the Pegasus spyware. Using BadBinder, an exploit [48] known since 2019, malicious actors can infect target devices with their payload. While an effective defense has been available for over 10 years [49], its rollout status in vendor-provided kernels is entirely opaque. The questionable deployment or absence of such defenses leaves devices vulnerable to one-day exploitation flows (see Figure 8.1), thus creating a significant

security gap in the Android ecosystem. Malicious actors can exploit this and mount attacks against insufficiently protected devices based on public exploits.

In this paper, we address the inadequate protection of Android devices against one-day exploitation flows through comprehensive analysis. We systematically analyze all publicly available one-day exploits targeting memory safety vulnerabilities in the Android Linux kernel over the past three years, comprising 26 exploits. In doing so, we unveil the diversity of these one-day exploits and classify 10 distinct exploitation techniques. In a subsequent analysis, we examine 8 defense-in-depth mechanisms present in the mainline Android kernel and find that they effectively prevent 84.6 % of the previously identified one-day exploitation flows. This percentage serves as the ground truth for how secure mobile devices could be if their kernels were up to date with the defenses enabled. Given the maximum achievable security, we can *quantify the level of security that is actually reached in Android devices.*

For this, we conduct the first large-scale analysis on kernel-level defense-in-depth mechanisms for Android devices via a mostly automated approach. We demonstrate a widespread absence across vendors and uncover flaws in vendor-specific defenses. In our analysis, we cover Android devices from all top 7 vendors (e.g., Samsung, Xiaomi, and Huawei), along with three recognized vendors (i.e., Google, OnePlus, and Fairphone), covering more than 84 % of the global Android device share [6]. We analyze devices from 2018 to 2023 using Android versions 9 to 14 and kernels ranging from v3.10 to v6.1. In total, we analyze 994 device firmwares and 1533 Android kernel source codes. Our results suggest that *the level of security that is actually reached is severely lacking* compared to the mainline Android kernel.

Our work presents four novel findings. First, we provide in-depth insights into the absence of effective defenses in vendor-provided kernels. On average, only 41.5 % of our analyzed one-day exploitation flows can be mitigated. This varies across vendors, from 28.8 % for the least (i.e., Fairphone) to 54.6 % for the most secure (i.e., Google) vendor, indicating a 4.62 to 2.95 times worse scenario than the ground truth.

Second, we unveil advancements in two exploitation techniques, enabling malicious actors to bypass the defense intended against the base technique. These advancements are applicable in all one-day exploitation flows that

use the base technique. While these advancements pose additional threats to Android devices, we discuss potential mitigations.

Third, we uncover 4 and 2 distinct weaknesses in Samsung's and Huawei's vendor-specific defenses, respectively. These issues impact Samsung devices ranging from Galaxy A04/A14 to Galaxy S23 5G/Ultra, and, thus, the entire range of low-end to high-end devices, as well as the entire range of Huawei devices. We demonstrate that these defenses do not fully prevent the targeted exploitation technique, or we demonstrate modified exploits that bypass the defense.

Fourth, we discuss factors that may contribute to the lack of effective defenses. While we observe a correlation between older kernel versions and higher one-day susceptibility, we reveal that susceptibility extends beyond mere version correlation. We present factors such as a lack of importance of security features and vulnerable configurations, as well as performance costs (confirmed by Google, Samsung, and Huawei), which are particularly relevant for low-end devices. We also make recommendations to Google and downstream vendors to improve Android security.

We open source[2] our tools that detect the widespread lack of included and effective defenses.

Contributions. The main contributions of our work are

(1) One-Day Exploitation Insights: We analyze 26 one-day exploits and classify 10 different exploitation techniques.
(2) Defense Insights: Based on these insights, we demonstrate defenses for the identified techniques.
(3) Defense Inclusion and Effectiveness Analysis: We unveil a significant gap between the maximum available security and that reached by vendor-supplied kernels.
(4) Novel Findings: We present in-depth insights into the absence of effective defenses in vendor-supplied kernels, exploitation advancements, weaknesses, and factors likely contributing to the absence of defense.

Disclosure. We disclosed our findings to all 10 vendors. While some did not respond (e.g., Oppo and Xiaomi), others (i.e., Google, Fairphone, Motorola, Huawei, and Samsung) acknowledged our findings (fully or partially), and some of these patched unsecured phones to enhance Android security.

[2] `https://github.com/IAIK/DefectsInDepth`.

Outline. Section 2 provides background. Section 3 shows the high-level workflow. Section 4 presents the one-day analysis and defense identification. Section 5 presents the large-scale defense analysis. Sections 6 and 7 discuss potential solutions and related work. Section 8 concludes our work.

2. Background

2.1. Android Ecosystem and Android Kernels

Android is primarily designed for mobile devices and undergoes active development led by Google. The Android kernel is based on the Linux kernel. For major platform releases, Google specifies compatible launch kernels for new devices and upgrades kernels for existing device updates.

Historically, vendors maintained separate Linux kernel trees for each product model, hindering upstream bug fixes due to vendor-specific code and hardware drivers. Despite the introduction of monthly Android Security Bulletins in 2015, prior research [23, 68] indicates continued delay in patch integration. In response, Google introduced the Generic Kernel Image (GKI) project in Android 11 on kernel versions above or equal to v5.4, aiming to overcome slow patch adoption. This initiative separates the Android kernel into a hardware-agnostic core maintained by Google and vendor-specific modules loaded dynamically. Moreover, it restricts the Android kernel to some constraints, such as ABI compatibility.

2.2. Kernel Exploitation

Fundamental Kernel Defenses. The Linux kernel employs defense-in-depth mechanisms to make vulnerability exploitation more difficult. These are included via the configuration file `.config`. One fundamental defense is the W^X policy, which dictates that sections may never be writeable and executable. Consequently, an attacker cannot simply inject instructions for privilege escalation. Kernel Address Space Layout Randomization (KASLR) randomizes the location of binary sections at boot time. Thus, an exploit typically breaks KASLR through a read primitive or a side channel [20]. Lastly, Privilege Access Never (PAN) prevents access to user-accessible memory while in kernel space, mitigating the control-flow redirection to userspace.

Kernel Exploitation on Android. The exploitation flow (see Figure 8.1) of most Android kernel exploits consists of three stages: First, an adversary *breaks KASLR* to identify the locations of critical structures. Second, the adversary obtains an *arbitrary read-and-write primitive* that allows them to perform the third step, which is gaining *full root privileges*.

To *break KASLR*, an adversary typically triggers a memory safety vulnerability, e.g., Use-After-Free (UAF) or Out-Of-Bounds (OOB) access, to leak a kernel address. By knowing the Android kernel binary under attack, the adversary then computes the kernel base address. Depending on how powerful the underlying vulnerability is, the adversary either continues or re-triggers this (or another) vulnerability to obtain an *arbitrary read-and-write primitive*. They then typically manipulate credentials to elevate their privileges. Furthermore, they tamper with kernel memory to disable SELinux's Mandatory Access Control (MAC), obtaining *full root privileges*.

Kernel Heap Attacks. Since most memory-safety vulnerabilities concern heap-allocated memory [66], dynamically allocated during runtime, it is a popular attack target.

Use-After-Free. UAF vulnerabilities occur when a resource that is still referenced is freed. A typical UAF exploit works as follows: First, an adversary causes the memory slot of a *vulnerable object* that is still in use to be freed. Freeing the memory slot causes the allocator to reuse the slot for future allocations. Second, they allocate a *reallocated object* such that the vulnerable and reallocated objects simultaneously use the previously freed slot. Third, they use either the vulnerable or the reallocated object to obtain a read or write primitive for the slot. Exploiting a Double-Free (DF) or Invalid-Free (IF) vulnerability (which are special cases of a UAF, where the slot is either freed twice or with an offset) works similarly.

In practice, several challenges render such attacks more difficult to execute. Most vulnerabilities grant only weak write capabilities, such as zeroing out memory at a particular offset. Additionally, to successfully exploit a UAF, the adversary requires knowledge of how the kernel's allocator (i.e., slab allocator) reuses memory slots.

There are generally two ways to exploit this reuse: With *in-cache reuse*, the adversary reuses the freed memory slot for another object that lives in the same slab cache. This only works in caches that contain the vulnerable and reallocated objects, e.g., `kmalloc-*` caches. Hence, the adversary

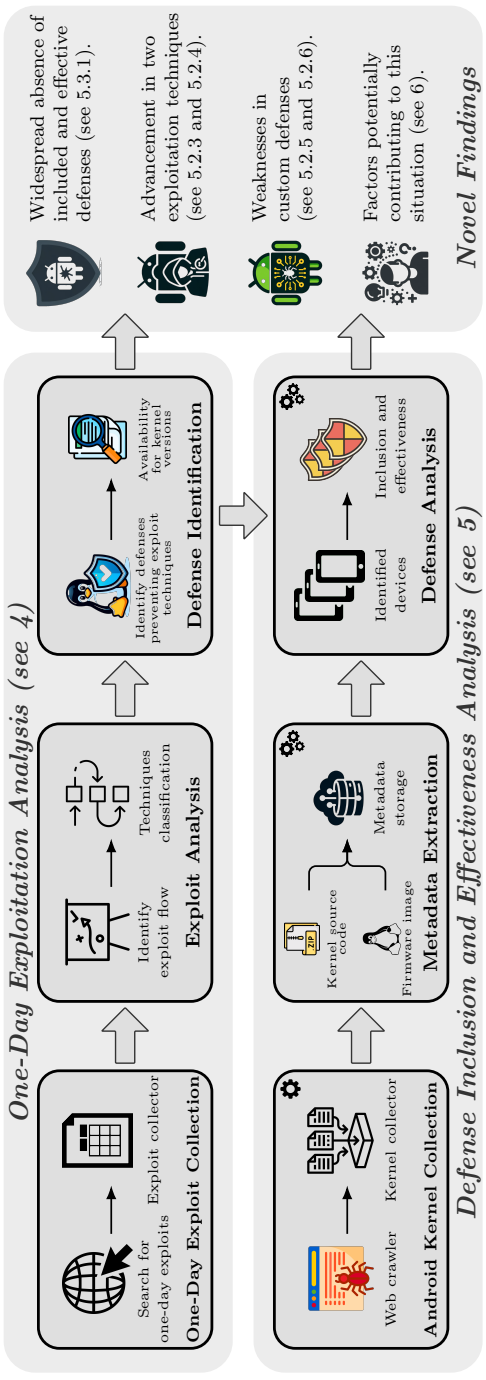


Figure 8.2: The high-level workflow of our study where ✿ indicates fully automated and ✿ indicates mostly automated.

is limited to objects that have the same (or similar) size and the same allocation properties as the vulnerable object.

The other way is to use a *cross-cache reuse* [33, 39, 61] attack. Here, the adversary frees all slots of a slab page, prompting the slab allocator to return the slab page that contains the vulnerable object to the page allocator. The page is then allocated either as a different type of page or to another slab cache. This allows them to reuse a memory slot between slab caches of different types, allocation sizes, and allocation properties.

Out-Of-Bounds. Exploiting an OOB vulnerability [12, 66] with write capabilities follows a similar process. An adversary triggers the OOB write, often in the form of a linear overflow, to manipulate sensitive data in an adjacent memory slot (i.e., victim object). This sensitive data typically references a vulnerable object, e.g., through a reference counter or data pointer [39, 43]. The adversary then forces the memory slot of the vulnerable object into a state where it is referenced twice. This upgrades the OOB write to be exploited analogously to the UAF three-stage exploitation flow typically.

3. High-Level Workflow

This section presents the high-level workflow of our study, depicted in Figure 8.2. It consists of three main components: the *One-Day Exploitation Analysis* and *Defense Inclusion and Effectiveness Analysis*, both of which yield *Novel Findings*.

In the *One-Day Exploitation Analysis* (see Section 4), we manually analyze all publicly available one-day exploits targeting memory safety vulnerabilities in the Android Linux kernel from the last three years. Our goal is to identify the exploitation flows employed in these exploits. In this context, we refer to an exploitation flow (see Figure 8.1) as a vulnerability-agnostic chain of exploitation techniques that exploit a vulnerability to gain full root privileges. An exploitation technique is a reusable and reasonably generic strategy for transforming an exploit primitive into a more powerful one [7]. In our study, we analyze 26 one-day exploits and uncover a diverse range of exploitation flows, with 10 used exploitation techniques. In a subsequent analysis, we identify 8 defense-in-depth mechanisms present in the mainline Android kernel v6.1, mitigating most exploitation techniques

and, hence, 84.6 % of exploitation flows. This percentage serves as the ground truth for the maximum achievable security of mobile devices.

In the *Defense Inclusion and Effectiveness Analysis* (see Section 5), we collect Android kernels released by all top mobile phone vendors (i.e., Samsung, Xiaomi, Oppo, Vivo, Realme, Huawei, Motorola, Google, OnePlus, and Fairphone) between 2018 and 2023. Our goal is to determine the inclusion and effectiveness of defense mechanisms in protecting these mobile devices. For this, we collect 994 device firmwares and 1533 kernel source codes. Our analysis reveals that a significant portion of the analyzed device firmwares lacks multiple defenses, and some of the defenses are flawed, leaving devices vulnerable to multiple of the one-day exploitation flows analyzed in our one-day exploitation analysis.

Our analysis reveals four *Novel Findings*. First, we reveal the widespread absence (see Section 5.3.1) of included and effective defenses against one-day exploitation flows across vendors. Second, we demonstrate advancements in two exploitation techniques (see Sections 5.2.3 and 5.2.4). While these advancements enable bypassing the defense intended against the base techniques, we discuss potential solutions. Third, we uncover 4 and 2 weaknesses (see Sections 5.2.5 and 5.2.6) in Samsung's and Huawei's custom defense, respectively. Lastly, we discuss (see Section 6) factors potentially contributing to the absence of effective defenses and offer improvement recommendations.

4. One-Day Exploitation Analysis

In this section, we elaborate on our systematic analysis of all publicly available one-day exploits targeting memory safety vulnerabilities in the Android Linux kernel over the past three years. We identify and examine 26 exploits, demonstrating that their exploitation flow uses one or more of the 10 exploitation techniques outlined in Section 4.1. These techniques follow a generic strategy for transforming an exploit primitive into a more powerful one. In Section 4.2, we demonstrate that defense-in-depth mechanisms present in the Android kernel v6.1 can mitigate 22 (i.e., 84.6 %) one-day exploitation flows. Lastly, Section 4.3 demonstrates that the remaining 4 one-day exploits either exploit substantially powerful vulnerabilities or can be mitigated by a defense currently in development [46].

Table 8.1: Exploitation flow used by publicly available one-day exploits, where defense-in-depth mechanisms present in the Android Linux kernel v6.1 can ✓ or cannot ✗ prevent the exploitation flow. The exploitation flow is preventable depending on ✶ Samsung's RKP [15] variant. Two one-day exploits ☆ exploit the same CVE with different exploitation flows.

CVE	Vulnerabilities	Exploitation flow	Goal Primitive	Preventable
CVE-2019-2215	UAF	in-cache reuse → unlink operation → KASLR leak, in-cache reuse → unlink operation → `addr_limit` overwrite	arbitrary r/w	✓
CVE-2019-2025 ☆	UAF	in-cache reuse → unlink operation → KASLR leak, in-cache reuse → unlink operation → `file` overwrite	arbitrary r/w	✓
CVE-2020-0030	UAF	in-cache reuse → unlink operation → KASLR leak, in-cache reuse → unlink operation → `addr_limit` overwrite	arbitrary r/w	✓
CVE-2021-1968,-1969,-1940	UAF	leak attacker-controlled data location, KASLR leak, in-cache reuse → CFH → ret2bpf	arbitrary r/w	✓
CVE-2021-0920	UAF	in-cache reuse → unlink operation → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2021-1905	UAF	cross-cache reuse → tamper allocator meta-data → KASLR leak → CFH → ret2bpf	arbitrary r/w	✓
CVE-2022-22265	DF	in-cache reuse → KASLR leak, cross-cache reuse → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2021-25369,-25370	Leak, UAF	KASLR leak, in-cache reuse → `file` overwrite → CFH → `addr_limit` overwrite	arbitrary r/w	✓
CVE-2016-3809,-2021-0399	Leak, UAF	KASLR leak, in-cache reuse → `seq_file` overwrite → CFH → ret2bpf	arbitrary r/w	✓
CVE-2022-20409	UAF	in-cache reuse → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2023-21400	DF	cross-cache reuse → Dirty PageTable	arbitrary r/w	✓
CVE-2022-28350	UAF	cross-cache reuse → Dirty PageTable	arbitrary r/w	✗/✶
CVE-2020-29661	UAF	cross-cache reuse → Dirty PageTable	arbitrary r/w	✗/✶
CVE-2021-22600	DF	in-cache reuse → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2020-0423	UAF	in-cache reuse → KASLR leak → unlink operation → KSMA	code modification	✓
CVE-2022-22057	UAF	in-cache reuse → KASLR leak → slab freelist corruption → KSMA	code modification	✓
CVE-2023-26083,-0266	Leak, UAF	KASLR leak, in-cache reuse → `ctl_file` overwrite → CFH	arbitrary r/w	✗
CVE-2020-0041	UAF	in-cache reuse → KASLR leak, in-cache reuse → unlink operation → tamper `sysctl`	arbitrary r/w	✓
CVE-2019-2205	UAF	in-cache reuse → KASLR leak, in-cache reuse → unlink operation → tamper `binder_proc`	arbitrary r/w	✓
CVE-2019-2025 ☆	UAF	in-cache reuse → KASLR leak, in-cache reuse → unlink operation → KSMA	code modification	✓
CVE-2020-3680	UAF	in-cache reuse → KASLR leak → unlink operation → KSMA	code modification	✓
CVE-2022-20421	UAF	cross-cache overflow → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓
CVE-2022-0847	Uninit Variable	uninitialized `pipe` → DirtyPipe → overwrite the cached file page	arbitrary r/w	✓
CVE-2021-4154	UAF	in-cache reuse → DirtyCred → overwrite shared library	arbitrary r/w	✗
CVE-2021-38001	OOB R/W	OOB write → stack manipulation → KASLR leak → OOB write → stack manipulation → CFH → ret2bpf	arbitrary r/w	✓
NO_NUMBER (∼2021)	OOB W	OOB write → slab freelist corruption → `pipe_buffer` DF → KASLR leak → `pipe_buffer` overwrite	arbitrary r/w	✓

22/26

```
struct list_head {
  struct list_head *next;
  struct list_head *prev;
};
/* Unlinks element e */
void list_del(struct
 -> list_head *e) {
  e->next->prev = e->prev;
  e->prev->next = e->next;
}
```

Listing 8.1: Unlinking operation.

```
struct binder_thread {
  struct list_head wait;
  struct task_struct *task;
};
void remove_wait_queue(struct
 -> binder_thread *bt) {
  /* Trigger unlinking */
  list_del(&bt->wait);
}
```

Listing 8.2: Trigger unlinking.

One-Day Exploits. We obtained 26 one-day exploits (see Table 8.1) from public sources, e.g., Google Project Zero [47], Blackhat [35], Github [42], or other websites [59]. Our selection consists of one-days exploiting memory safety vulnerabilities, as the Android kernel has established defenses to prevent their exploitation. By including other vulnerabilities, such as logical (e.g., CVE-2022-22706) and GPU (e.g., CVE-2023-33107) flaws, we expect that the susceptibility to one-day exploits increases as the mainline Android kernel does not yet effectively mitigate them. Our study spans the last 3 years, from 2020 to November 2023. This aligns with Google Project Zero's efforts to track zero-day exploits targeting Android devices. Earlier public exploits are less documented, so we focus on this more recent timeframe [50].

4.1. Identified Exploitation Techniques

We observe that most one-day exploits have distinct exploitation flows to convert one or more memory safety vulnerabilities into either an arbitrary read-and-write primitive or code modification (see Table 8.1). By examining these exploitation flows, we identify 10 exploitation techniques.

We refer to an exploitation technique as a strategy for turning one exploitation primitive into a more powerful one, with examples of primitives being n-byte OOB write, UAF write, program counter control, or arbitrary read and write. We classify exploitation techniques based on strategies that recur over multiple one-days and are reasonably generic [7]. An example of a technique is control-flow hijacking, which turns program counter control into code execution and is used by multiple one-days. Another

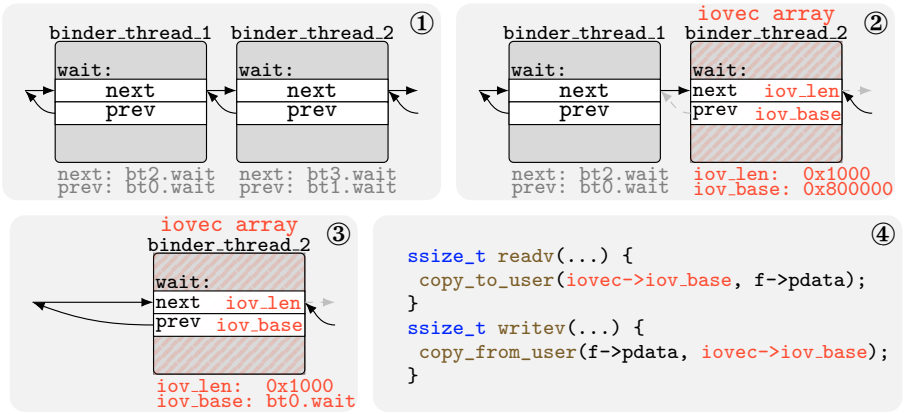


Figure 8.3: Exploitation example of the unlink operation.

example is the unlink operation, which may turn an OOB or UAF write of a double-linked list into a once-triggerable write or read primitive.

ET1: Unlink Operation. By exploiting a vulnerability, an adversary ensures that a victim object resides in the same memory slot as a double-linked list, i.e., `list_head` with `next` and `prev` (see Listing 8.1). The adversary then initiates the unlinking via `list_del`, resulting in a write to the victim object. The one-days CVE-2019-2215, CVE-2019-2025, and CVE-2020-0030, for example, leverage this unlink operation to first leak `binder_thread->task`, whose layout is shown in Listing 8.2, and then overwrite `task->addr_limit` (**ET2**).

Figure 8.3 illustrates an exploitation example [48], where initially, an adversary prepares a double-linked list ①. They then exploit a vulnerability to ensure that the second `wait` entry (`binder_thread_2.wait` or short `bt2.wait`) resides in the same memory slot ② as an `iovec` object. This `iovec` stores a user buffer, with `iov_base/len` being the user buffer's pointer/size, commonly used for file reading or writing. Executing `remove_wait_queue` on the first `wait` entry (`bt1.wait`) overwrites the `iov_base` of the second `wait` entry with `bt1.wait->prev` ③ (at Line 7 of `list_del`). Consequently, the buffer `iovec` now points to `binder_thread_0.wait` (short `bt0.wait`). These exploits then use the `iovec` read/write functionality ④ (e.g., `readv` or `writev`) to write to or read from the `iovec->iov_base` and, hence, `bt0.wait`. This approach is used to leak `binder_thread->task` and overwrite `task->addr_limit`. While this example shows the usage of `iovec` (which has been fixed for v4.13 [2]),

```
1 u64 access_ok(const void __user *addr, u64 size) {
2   return (u64)((u65)addr + (u65)size <= (u65)current->addr_limit +
  -> 1);
3 }
4 u64 copy_from_user(void *to, const void __user *from, u64 n) {
5   u64 res = n;
6   if (access_ok(from, n))
7     res = raw_copy_from_user(to, from, n);
8   return res;
9 }
```

Listing 8.3: Userspace data copy function validates with `access_ok` whether `addr` refers to userspace memory.

other security-critical objects can also be misused, e.g., `msg_msg` [43] or `pipe_buffer` in CVE-2021-0920.

ET2: `addr_limit` Overwrite. This technique turns a `task->addr_limit` overwrite into an arbitrary read and write. AArch64 kernels below v5.11 include `addr_limit` in `task`, which holds the highest address accessible within user-data copy functions, e.g., `copy_*_user`. These functions call `access_ok` to validate that the user address is lower than `addr_limit` (see Line 6 of Listing 8.3), aiming to ensure user address access. However, by overwriting `addr_limit` with `KERNEL_DS` (i.e., -1), an adversary can deceive the kernel into legally accessing kernel memory within these copy functions. Hence, syscalls (e.g., `read` and `write`) using these copy functions can be misused as an arbitrary read-and-write primitive.

ET3: `pipe_buffer` Overwrite. Overwriting the `pipe_buffer` yields an arbitrary read and write as follows. Initially, an adversary requires an arbitrarily triggerable overwrite capability for a `pipe_buffer` object that is still in use. One approach is to exploit a UAF vulnerability so that a `pipe_buffer` and a specific object (e.g., `eventfd_ctx` or `signalfd_ctx`) reside in the same memory slot. Since this specific object is writable from userspace, it enables manipulating `pipe_buffer` (e.g., `eventfd_ctx` for CVE-2021-22600). Another approach enforces the coexistence of a `pipe_buffer` and the backed physical page of another `pipe_buffer` in the same slot (cf. CVE-2022-22265).

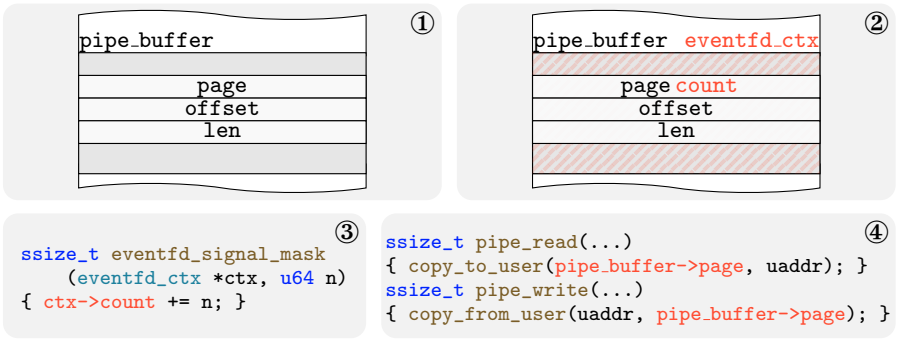


Figure 8.4: Exploiting `pipe_buffer` to obtain an arbitrary r/w.

Figure 8.4 illustrates the exploitation of CVE-2021-22600. In ①, the memory layout of a `pipe_buffer` is shown with its members `page`, `offset`, and `len`. Step ② exploits the vulnerability where afterward `pipe_buffer` and `eventfd_ctx` reside in the same memory slot, and `page` and `count` coexist on the same address. Calling `eventfd_signal_mask` ③ allows to change `count` and, hence, `pipe_buffer->page`. Consequently, `pipe_read/write` ④ read from or write to this controlled address, granting an arbitrary read and write.

ET4: Control-Flow Hijacking. Various one-day exploits perform a Control-Flow Hijacking (CFH) attack, leveraging an overwrite capability of either a function pointer or a pointer to a function pointer. Compared to x86_64 exploitation, they do not resort to Return-Oriented Programming (ROP) [10]. Instead, they identify an execution path resulting in an arbitrary read-and-write primitive. For instance, CVE-2023-0266 overwrites the `f_ops` pointer (referencing a table of function pointers for file interactions) of `ctl_file`. As a result, the syscalls `read` and `write` confuse the `void *pdata` member of `ctl_file`, leading to a misuse of `copy_*_user` and yielding an arbitrary read-and-write primitive.

ET5: Ret2bpf. Ret2bpf serves as an alternative to ROP, offering a similarly potent capability [8, 28]. Its prerequisites [28] involve hijacking the control flow (**ET4**), partial control of the first argument register, control over the second argument register, and a controllable data region. In ret2bpf, a data region is crafted to contain valid eBPF [8] instructions, performing, for instance, an arbitrary read and write. With the CFH primitive and the crafted eBPF instructions, ret2bpf performs a CFH attack to execute `___bpf_prog_run(regs, inst)`. This function interprets the crafted eBPF instructions as if the eBPF verifier had validated them.

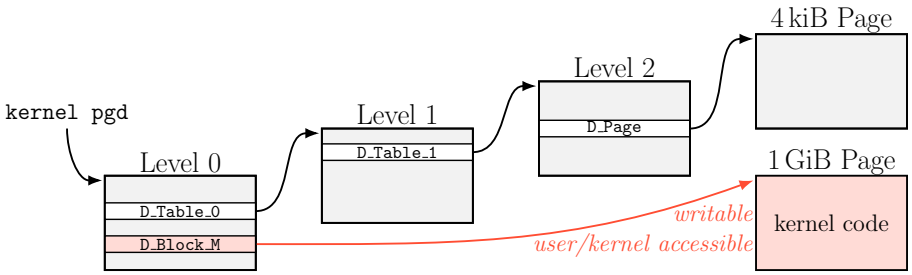


Figure 8.5: **KSMA:** Due to a write capability to page table level 0, an adversary maliciously overwrites the `D_Block_M` entry to refer to kernel code as writable and user accessible.

Here, `inst` is the data region holding the crafted eBPF instructions, and `regs` represents a writable section used for registers.

ET6: Slab Freelist Corruption. This exploitation technique turns a once-only OOB or UAF write into a memory slot overwrite capability. It requires a memory slot that is currently in the freed state. By exploiting a write capability on this free slot (e.g., zeroing memory due to a UAF or OOB write), an adversary manipulates a freelist pointer stored within the free slot. Then, by allocating an object, the adversary illegally reclaims the memory slot referenced by the corrupted freelist pointer. This allocated object typically grants overwrite capabilities for the reclaimed memory slot.

ET7: KSMA. Yong et al. [64] introduced the Kernel-Space Mirroring Attack (KSMA), which transforms a once-triggerable write primitive into a kernel code manipulation capability. This transformation is done by manipulating a page table level 0, called Page Global Directory (PGD) (e.g., `swapper_pg_dir`), representing the kernel address space.

Specifically, KSMA forges an entry within the kernel's page table level 0, designating its address range as accessible from user and kernel space. This forged entry is marked as a 1 GB huge page and references kernel code. Consequently, the entire kernel code (including kernel data) is readable and writable from userspace. The page-table layout after performing KSMA is shown in Figure 8.5 with a 3-level page-table translation (i.e., 39 bit Virtual Address Size (`VA_SIZE`) and 4 kiB page size, but it works similarly for other configurations). This kernel code modification is then utilized to disable SELinux and manipulate a syscall to elevate the privileges.

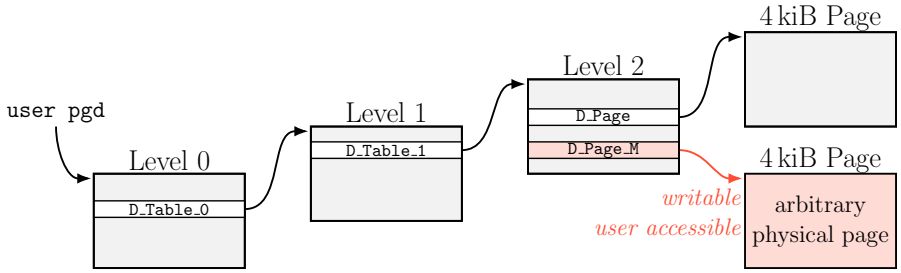


Figure 8.6: **Dirty PageTable:** With an arbitrary page table level 2 write capability, an attacker tampers the `D_Page_M` entry to refer to an arbitrary page as writable and user accessible.

ET8: Dirty PageTable. Dirty PageTable [59] shows how page-table tampering results in an arbitrary read and write on Android (where Maar et al. [39] show generic page-table manipulation). It exploits a UAF (cf. CVE-2022-28350 and CVE-2020-29661) or DF (cf. CVE-2023-21400) for a cross-cache attack [61]. This causes an object with arbitrary overwrite capabilities (e.g., `signalfd_ctx` for CVE-2023-21400) to reside in the same memory slot as a page table used for user address translation. Figure 8.6 shows this, where an adversary has an arbitrary overwrite to the page-table entry `D_Page_M` due to the cross-cache attack. By triggering the overwrite, they gain control over the page frame number of this entry. Reading or writing to the user address using this page-table entry gives them arbitrary physical memory access.

ET9: DirtyPipe. The DirtyPipe attack [30] exploits an uninitialized variable to escalate privileges. The CVE-2022-0847 vulnerability allows to use the `pipe_buffer.flags` variable uninitialized. Consequently, this vulnerability allows overwriting of any file contents in the page cache, also in the case of read-only files, which results in privilege escalation.

ET10: DirtyCred. The DirtyCred exploit [34] allows an attacker to escalate privileges. It exploits a file UAF to free a writable file currently in use. Prior to this invalid free, it performs a write operation to the file and stalls this write operation. After the free, it reclaims the file object for a read-only high-privilege file. Continuing the stalled write operation now writes to the read-only high-privilege file. With this file manipulation, DirtyCred can, e.g., overwrite a kernel module with malicious code to construct an arbitrary read and write.

4.2. Defenses to Prevent Exploitation Flows

We identify 10 defense-in-depth mechanisms present in the Android kernel v6.1 or provided by vendors. They prevent exploitation techniques and, hence, exploitation flows, with the findings shown in Table 8.1 and detailed in Table 8.2.

DM1: `CONFIG_DEBUG_LIST`. This defense includes checks in `del_list` whether `e->next->prev == e` and `e->prev->next == e`. If these checks fail, the entry will not be unlinked. Thus, it mitigates the unlink operation (**ET1**). In Figure 8.3, for instance, overwriting from step ② to ③ is prevented as `iovec->iov_base` is not equal to `bt1.wait`.

DM2: `CONFIG_ARM64_UAO`. User-Access Override (UAO) [9] is a hardware-enforced defense that aims to mitigate `addr_limit` overwrite (**ET2**). It introduces new unprivileged load and store instructions that behave like privileged ones when the UAO bit is set. This restricts user-data copy functions, e.g., `copy_*_user`, from being misused to read from or write to kernel addresses directly.

DM3: `kmalloc-cg-*`. Linux kernels above v5.13 support heap segregation at the allocator cache level. It separates caches to provide a designated cache for security-critical data marked as accounted, such as for `msg_msg`, `pipe_buffer`, `file`, and `task_struct`. For generic caches, a cache for non-security-critical data (i.e., `kmalloc-*`) and a cache for security-critical data (i.e., `kmalloc-cg-*`) are created. Free and available cached objects will never share the same memory slots within these caches. Hence, this mitigates the `pipe_buffer` overwrite (**ET3**) and unlink operation (**ET1** with security-critical objects), as these techniques rely on security-critical and non-security-critical data sharing the same memory slot. While adversaries might consider cross-cache attacks, three challenges arise with this approach, making the transition infeasible. First, for generic caches, the success rate significantly decreases to 40 % [61], with failure scenarios potentially resulting in a crash. The small success rate makes this approach impractical since the cross-cache reuse only pertains to a small part of the exploit and may need multiple repetitions. Second, exploits that engage in cross-cache attacks typically rely on prior in-cache reuse attacks to stabilize the exploit. For instance, CVE-2022-22265 stabilizes by in-cache reallocating the double-freed slot of the `pipe_buffer` as an `iovec` multiple times. Separating the `pipe_buffer` from objects intended for stabilizing, such as `iovec`, makes the exploit unstable, mostly resulting in a crash, successfully preventing exploitation. A similar applies

to CVE-2023-21400, where `seq_operations` (accounted) are prevented from being in-cache reallocated as `signalfd_ctx` (not accounted). Third, various UAF exploits (e.g., CVE-2021-0399) offer a tight time window in which an in-cache attack is exploitable. In contrast, cross-cache attacks require more time due to the recycling/reclaiming of the slab page to/from the page allocator [61], making small windows not exploitable.

DM4: `CONFIG_CFI_CLANG`. Control-Flow Integrity (CFI) [1, 3] restricts the control flow to an approximate Control-Flow Graph (CFG), limiting the targets for CFH attacks (**ET4**). The Android kernel uses Clang's implementation [3], providing function-signature-grained CFI. It prevents CFH attacks that overwrite function pointers with arbitrary functions, e.g., CVE-2021-1905 and CVE-2021-0399.

DM5: `CONFIG_BPF_JIT_ALWAYS_ON`. To mitigate ret2bpf (**ET5**), this defense mechanism forces BPF to always use the JIT engine instead of the interpreter. Consequently, the `___bpf_prog_run` function used by ret2bpf is not compiled and, therefore, cannot be called, preventing ret2bpf.

DM6: `CONFIG_SLAB_FREELIST_HARDENED`. To mitigate the manipulation of slab allocator metadata, this defense hardens the slab allocator by adding sanity checks. This includes XORing the freelist pointer with a pseudo-random number, preventing slab freelist corruption (**ET6**).

DM7: `CONFIG_INIT_ON_ALLOC_DEFAULT_ON`. It zeroes out the memory slot for both allocations by the page and slab allocator. Consequently, it greatly minimizes the exploitability of uninitialized values. It prevents the exploitation of DirtyPipe (**ET7**) as the uninitialized `pipe_buffer.flags` cannot be misused to overwrite file content in the page cache.

DM8: KSMA Protection. In response to KSMA (**ET8**), researchers proposed to move all kernel level 0 global page tables (e.g., `swapper_pg_dir` and `tramp_pg_dir`) to a read-only section [63]. As a result, these page tables cannot be manipulated for a huge kernel memory mapping (e.g., 1 GiB) that is writable from userspace, thus preventing KSMA.

DM9: Samsung RKP. Samsung's Real-time Kernel Protection (RKP) [15] employs hypervisor-based protection designed to mitigate code modification, data modification, and control-flow hijacking in the kernel. To address kernel code modification, RKP ensures the integrity of page tables (**ET8** and **ET9**) and code by mapping them as read-only, protected by the hypervisor. Hypervisor calls permit legitimate writes to these protected pages. RKP also limits CFH attacks (**ET4**) by including checks before

indirect branches that restrict control-flow transfers to a function-grained CFG.

DM10: Huawei HKIP. Huawei Kernel Integrity Protection (HKIP) [24] employs hypervisor-based protection that protects kernel code and critical kernel data. It also limits privilege escalation and protects additional control-flow-related data. To achieve this, HKIP ensures the integrity of certain page tables (**ET8** and **ET9**), `addr_limit` (**ET2**), CFI metadata, and eBPF interpreted code by protecting them via the hypervisor. Hypervisor calls or exceptions to the hypervisor permit legitimate writes to these protected pages. Protecting CFI metadata only provides additional protection for modules and not against the CFH technique. Similarly, protecting the eBPF-interpreted code does not prevent against ret2bpf, as it only safeguards the already interpreted instructions.

Further defenses. The ongoing research in improving kernel security yielded results with various kernel defenses. For instance, `CONFIG_HARDENED_USERCOPY` restricts `copy_*_user` from reading and writing out of bounds [60]. Other examples include `CONFIG_INIT_STACK_ALL_ZERO` mitigating uninitialized stack variable exploitation [37] and `CONFIG_STACKPROTECTOR_STRONG` providing stack protection [25]. While these defenses cover a broad range of vulnerability mitigation, our focus is specifically on defenses against one-day exploitation flows on the Android kernel (**DM1-10**).

4.3. Unpreventable Exploitation Flows

We identify 4 one-day exploitation flows targeting the Android kernel that remains unpreventable by mainline defenses. Among these, DirtyCred (cf. CVE-2021-4154) presents a powerful technique that falls beyond the defense prevention scope. A similar applies to the CFH one-day (cf. CVE-2023-26083,-0266), redirecting the control flow to perform an arbitrary read and write without violating signature-based CFI.

While our identified defenses do not effectively prevent two other one-days (Dirty PageTable, cf. CVE-2022-28350 and CVE-2020-29661), researchers are actively developing a new defense mechanism specifically designed to counter cross-cache reuse attacks [46]. This mitigation strategy involves switching the allocation of memory slots cached by the slab allocator from physical to virtual pages, thereby preventing the reuse of slab pages returned by the page allocator.

5. Defense Inclusion & Effectiveness Analysis

In this section, we outline our systematic analysis demonstrating a *widespread deficiency of included defense mechanisms across vendors* as well as *shortcomings of certain defenses*. Our approach (see Figure 8.2) consists of three mostly automated stages: Initially, we collect kernel source codes and firmwares (see Section 5.1) for Android devices from 10 vendors. We then analyze kernel codes (see Section 5.2) to assess the effectiveness of defenses provided by the mainline kernel or vendors. Lastly, we analyze firmwares (see Section 5.3) to detect included effective defenses in devices.

Android Devices. For our analysis (done in November 2023), we cover Android devices from vendors that constitute more than 84 % [6] of the global market. These include the top 7 vendors [6], i.e., Samsung, Xiaomi, Oppo, Vivo, Huawei, Realme, and Motorola, along with Google, OnePlus, and Fairphone. We assess devices released between 2018 and 2023, utilizing Android versions 9 through 14 and kernels ranging from v3.10 to v6.1. These Android versions account for a share of more than 86 % [5], with specific percentages for Android 13, 12, 11, 10 and 9, being 25.7 %, 21.3 %, 17.3 %, 13.9 %, and 8.7 %, respectively. Android 14, while at a negligible market share at the moment, is also considered.

We decided to start with phones released in November 2018 (5 years from the start of this work), as the lifespan of Android phones is 4-6 years (4y for Huawei, 5y for Google, and 6y for Samsung) [16, 55]. Hence, our selection ensures a comprehensive overview of the current device landscape.

5.1. Collection of Firmwares and Kernel Codes

This step automatically collects firmwares (not protected by captchas) and kernel code. To achieve this, we implement a Python script using Selenium that crawls web pages to collect firmwares and kernel source code from our 10 vendors. We manually collected firmware protected by captchas or other automation detections (i.e., $\approx 45.3\,\%$).

Firmwares. Our 10 vendors produced 1698 devices between November 2018 and November 2023 (see Table 8.4). For 1109 of them, firmwares were provided, where we only considered the most recent release either officially (e.g., Google) or via an intermediate supplier (e.g., Oppo).

Kernel Codes. We collected 1533 kernel codes (see Table 8.4) with different releases for the same device (e.g., Samsung and Huawei) where available. Other vendors (e.g., Google and Vivo) use the same kernel code for multiple devices, resulting in less collected code than firmwares.

5.2. Analysis of Kernel Source Codes

We examine kernel source codes for efficacy against exploitation techniques. Initially, we provide evidence that our identified defenses (see Sections 5.2.1 and 5.2.2) reflect the real world of mitigating exploitation techniques. However, we also identify shortcomings in these defenses. We show that they can be bypassed, indicating that their efficacy is less than intended due to advanced techniques (see Sections 5.2.3 and 5.2.4) or weaknesses (see Sections 5.2.5 and 5.2.6).

5.2.1. Mainline Defenses in Downstreamed Kernels

7 of the 8 mainline defense mechanisms are intrinsically tied to the core functionality of the Android kernel:

- Associating specific defenses with versions, i.e., **DM3**.
- Not compiling dangerous functions, i.e., **DM5**.
- Replacing non-hardened with hardened functions, i.e., **DM1/4/6/7/8**.

For example, `CONFIG_DEBUG_LIST` (**DM1**) uses the hardened function `__list_add_valid` to validate metadata in double-linked lists. Another example is `kmalloc-cg-*` (**DM3**), which utilizes a segregated set of allocator caches for kernel versions 5.13 and above. The exception is `CONFIG_ARM64_UAO` (**DM2**), which is hardware-dependent. Although our analysis might identify this defense as present, this defense can be bypassed, as we demonstrate in Section 5.2.4. Hence, regardless of whether it is enabled, it does not protect `addr_limit` overwrite (**ET2**).

5.2.2. Identified Downstream Defenses

Some vendors provide custom defenses to improve kernel security. We semi-automatically analyze the 1533 collected kernels and provide evidence of 3 vendor-specific downstream defenses against the identified exploitation techniques. The analysis works as follows: First, we automatically collect

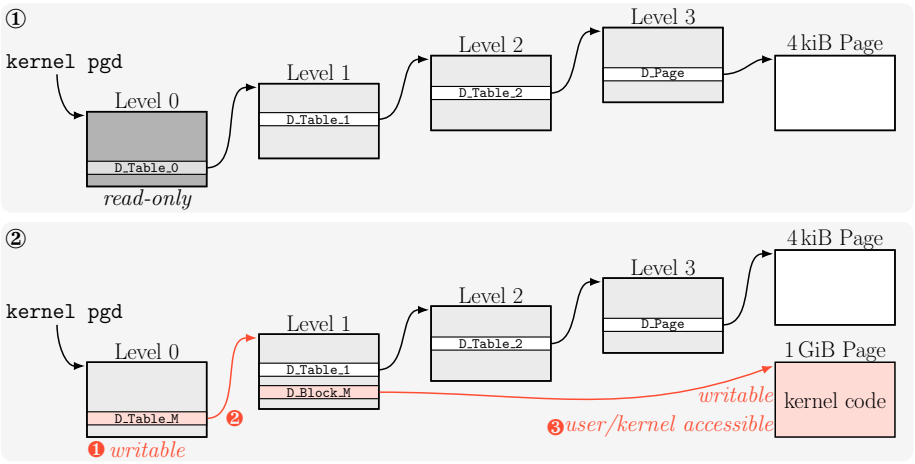


Figure 8.7: **Advanced KSMA:** ① Initial 4-level page table translation with the level 0 table mapped as read-only. ② Modifies the level 0 mapping to mark it as writable ❶, overwrites `D_Table_M` ❷, and appends `D_Block_M` ❸, to have a 1 GiB mapping accessible from userspace.

the configuration flags in the `./security` subdirectory and in the files that require changes to mitigate exploitation techniques, e.g., `vmlinux.ld.S` and `mmu.c` to prevent KSMA (**ET8**). Second, we manually analyze those flags collected from downstream kernels that are not present in the mainline kernel (i.e., Google). Our analysis results in Samsung RKP (**DM9**), Huawei HKIP (**DM10**), and `CONFIG_PG_DIR_RO` from Vivo (which we consider in the firmware analysis as **DM8**). We also received confirmation from Fairphone and Motorola that they do not include vendor-specific defenses.

5.2.3. Advanced Kernel-Space Mirroring Attack

Despite the KSMA mitigation patch, we present an advancement in reenabling KSMA. Its prerequisite is the same constraint write capability as the base KSMA (**ET8**), but it uses it twice: First, it changes the mapping of the level 0 PGD to writable, and second, it maliciously inserts the page-table entry into the PGD. For a 48 bit `VA_SIZE` system with 4-level translation, our technique triggers the write three times, as depicted before ① and after ② the attack in Figure 8.7. The first write changes the PGD mapping to writable ❶, while the second changes the PGD entry to user accessible ❷. The third write inserts then the entry ❸ in the page-table level 1.

For this technique, the locations of three page-table pages (*level 0*, *level 1*, and *level 3'* corresponding to the mapping *level 0* as read-only) are crucial. Depending on whether `CONFIG_UNMAP_KERNEL_AT_EL0` is active, `swapper_pg_dir` or `tramp_pg_dir` is used as a *level 0* page. A KASLR code leak, combined with knowledge of the kernel binary under attack, is sufficient to obtain these locations since both locations are mapped to a fixed offset to the kernel base address. This step is also needed for the base KSMA.

Both *level 1* and *level 3'* are allocated via the page allocator during the early initialization stages and accessed via the Direct-Physical Mapping (DPM) [40], which is a virtual memory mapping to the entire physical memory. Since the page allocator returns the same physical page during different boots, their locations can be determined. The DPM may be randomized on newer Android kernels (e.g., v6.1). To overcome randomization, a heap address leak is typically sufficient to derandomize the DPM, as the kernel heap uses the DPM directly. Typically, leaking a heap address requires no additional effort beyond leaking the kernel base address.

Armed with the three page-table locations, our advanced KSMA uses the write capability three times to obtain kernel code modification, which no mainline defense can prevent. The level of difficulty of our advancement is similar to the base KSMA, but the write capability is triggered three times, and for recent Android versions, a heap leak is required.

Experiments. Our setup involves a buildroot filesystem with an Android kernel (aarch64 with a 48 bit `VA_SIZE`), specifically v5.15 and v6.1. We run it inside a virtual machine with 4 cores and 4 GB RAM via QEMU 6.2.0. We introduce a write primitive and run our exploits as an unprivileged user, giving them the same capabilities as the base KSMA version. As a result, we successfully execute our advanced KSMA and obtained an arbitrary code modification primitive.

Mitigation. Our advanced KSMA requires the locations of all mapping page tables that are not randomized during the early initialization process. Hence, an adversary can still deduce their location by knowing the kernel binary under attack (and a heap leak for v6.1). To counteract the advanced KSMA, we propose randomizing the locations of the page tables during this early initialization stage, ensuring that adversaries cannot obtain information about the page's location.

5.2.4. Shortcoming of User Access Override

The UAO feature is believed to prevent the `addr_limit` overwrite (**ET2**) [36] effectively. This technique manipulates `addr_limit` with `KERNEL_DS` to facilitate, for example, pipes for arbitrary kernel reads and writes. Specifically, it first writes the pointer to a userspace buffer to one pipe end. It then performs a `read` syscall with a kernel address as an argument, prompting the userspace copy function to write the userspace buffer's content to this kernel address. With UAO enabled, setting `addr_limit` to `KERNEL_DS` prevents the first write operation. Moreover, setting `addr_limit` to `USER_DS` prevents the second write to kernel memory.

However, since `addr_limit` operates at thread granularity, we spawn two threads, *T1* and *T2*, where we only illegally overwrite the `addr_limit` of *T2* with `KERNEL_DS`. We leverage *T1* to perform the first write and *T2* for the second write. As a result, we can bypass UAO without further restrictions. Prior work [9] has presented similar bypasses.

Mitigation. A mitigation would be to remove the `addr_limit` functionality or use kernels above v5.11, which do not support `addr_limit` anymore.

5.2.5. Samsung RKP Weaknesses

We inspect Samsung RKP [15], designed to prevent page-table manipulation and limit CFH attacks. However, we demonstrate that various RKP variants only protect certain page tables and, thus, do not mitigate page-table manipulation. They also provide less CFH protection than the mainline defense.

Analysis. For each of these findings, we provide statistical data on their occurrence, collected using the following approach. We first perform automated source code analysis, followed by manual verification. We then conduct experiments to demonstrate the severity of these identified problems.

Findings. First, some kernels have RKP disabled and do not map `tramp/swapper_pg_dir` or `tramp_pg_dir` as read-only. Compared to the mainline defense, this results in less security as an adversary can directly perform KSMA. We found this weakness mostly in low-end devices such as Galaxy A04/A14 (i.e., released 2022/2023), missing both pages and Galaxy

M10 (i.e., released 2019), missing `tramp_pg_dir`, representing 25.4 % and 1.7 % of kernels, respectively.

Second, while some variants protect page tables used for userspace address translation, we observe a strong tendency to exclude this protection towards new high-end devices such as Galaxy S23 5G. Specifically, we observe that less than 53 % of devices include this protection, indicating that more than 47 % are vulnerable to Dirty PageTable [59].

Third, `CONFIG_FASTUH_RKP` is a performance-optimized RKP variant included in over 60 % of all v5.4 kernels, providing a maximum number of read-only pages protected by the hypervisor. If the system demands more, RKP resorts to allocating unprotected pages. An adversary can exhaust these read-only protected pages and, subsequently, perform Dirty PageTable. This performance-optimized RKP variant is available for lower-end devices, e.g., Samsung Galaxy J6, and for high-end devices, e.g., Samsung Galaxy S20 FE and S21+ 5G. Similarly, `CONFIG_TIMA_RKP` provides similar weak protection for page tables, mainly used by older devices.

Fourth, `CONFIG_RKP_CFP_JOPP/_ROPP` aim to mitigate CFH attacks [15] by providing function-granular CFI (`JOPP`) and return address protection (`ROPP`). However, our analysis of exploitation flows reveals that all 6 CFH attacks redirect the control flow with at least function granularity. Hence, both defenses are ineffective in mitigating any of the CFH attacks.

Experiments. For the first weakness, we use the same setup as for our advanced KSMA technique and overwrite unprotected page-table pages to perform KSMA. We then implement POCs for the other weaknesses on a Samsung Galaxy S20 FE, where we modify the kernel code to obtain the corresponding primitive. For the second weakness, we use the introduced write primitive for a page table used for a userspace address translation to successfully perform Dirty PageTable. For the third weakness, we demonstrate that we can drain the protected pages with memory exhaustion. We then prompt the kernel to allocate a page that should be protected, but this is not due to memory exhaustion. For the fourth weakness, we demonstrate that RKP does not mitigate control-flow hijacking to arbitrary functions. As a result, control-flow protection does not prevent the 6 CFH attacks we analyzed.

5.2.6. Huawei HKIP Weaknesses

We examine Huawei's HKIP [24], particularly regarding the protection against KSMA and Dirty PageTable.

Analysis. We observe that HKIP is only included in certain devices and enabled in about 62 %. In the following, we analyze HKIP and experimentally demonstrate the absence of protection for crucial page-table pages.

Findings. First, HKIP protects page-table pages that are allocated for kernel address translations (e.g., via `pte_alloc_one`) in a specific virtual address range. As a result, HKIP does not protect page tables for userspace address translations, leaving devices vulnerable to Dirty PageTable.

Second, while HKIP protects the ttbr (hardware register that stores the current PGD for address translation) switch, it may not be compatible with frequent ttbr switching defenses, i.e., software PAN (`CONFIG_ARM64_SW_TTBR0_PAN`) switches the ttbr for each `copy_*_user` and Meltdown protection (`CONFIG_UNMAP_KERNEL_AT_EL0`) for each user kernel switch. No device has HKIP with either one of these two enabled, leaving these devices vulnerable to KSMA.

Experiments. We could not run experiments with the Huawei kernel source codes as they either had compilation errors, no `defconfig` (e.g., `ranchu64`) viable for virtual environments or failed to boot in QEMU. Therefore, we adapted a Google kernel v4.14 to tag pages that HKIP would have protected. For our page-table manipulation attacks, we experimentally observed that HKIP does not protect page-table pages that KSMA and Dirty PageTable manipulate.

5.3. Analysis of Firmwares

This work refers to the firmware as the stock ROM, the original software loaded onto the device by the vendor. It consists of multiple images [4], such as the system and boot image. Figure 8.8 shows the automated workflow of our implemented Python script, extracting the necessary metadata for defense detection. It first extracts the boot image ① using open-source tools, which requires different tools [21, 31, 32, 52, 56] as vendors encode the boot image differently. It then extracts the kernel binary ② using `unpack_bootimg` [35]. Lastly, it uses `kallsyms_finder` and

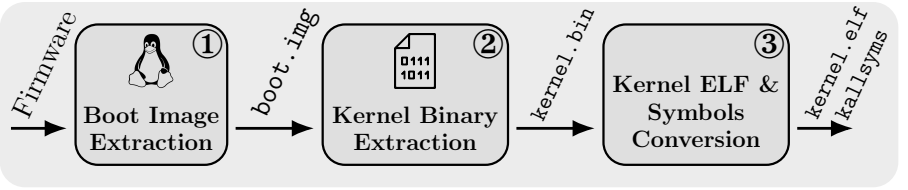


Figure 8.8: Workflow of extracting `kernel.elf` and `kallsyms` from the firmware, required for the defense detection.

`vmlinux_to_elf` to reconstruct the symbols (i.e., `kallsyms`) and convert the kernel binary to an analyzable ELF (i.e., `kernel.elf`) ③ [41].

The `kallsyms` and `kernel.elf` components form the basis of defense detection. Our Python script uses `kallsyms` to identify global functions within the kernel binary, allowing us to deduce the active defense mechanisms. The presence of `__list_add_valid` in `kallsyms`, for instance, indicates the status of `CONFIG_DEBUG_LIST` (**DM1**). Our script does similar assessments for other defenses (see Table 8.3). It uses the `kernel.elf` to determine the status of KSMA protection (**DM8**) and `CONFIG_SLAB_FREELIST_HARDENED` (**DM6**). For KSMA protection, all PGDs (e.g., `swapper_pg_dir`) must be mapped in a read-only section. The presence of calling `get_random_long` within `__kmem_cache_create` indicates the status of `CONFIG_SLAB_FREELIST_HARDENED`.

Our evaluation also includes five features for system security; KASLR (`CONFIG_RANDOMIZE_BASE`), code write protection (`CONFIG_STRICT_KERNEL_RWX`), freelist randomization (`CONFIG_SLAB_FREELIST_RANDOM`), restricting user access in kernel (`CONFIG_ARM64_(SW_TTBR0_)PAN`), and Meltdown protection (`CONFIG_UNMAP_KERNEL_AT_EL0`).

Evaluated Firmwares. Out of the 1698 released and 1109 collected devices, our analysis extracted 994 firmwares, resulting in a collection rate of 58.5 %, which aligns with prior work on reverse engineering firmwares [11, 14, 65].

Due to the unavailability of certain firmwares, our analysis could not cover all released devices. However, we observed that the missing firmwares are distributed either normally regarding device age, such as those from Huawei and Vivo, or tailored to older devices, as seen with Xiaomi and Realme. Given our finding that older devices tend to include fewer defenses, our analysis provides conservative results. Thus, we anticipate the real-world scenario to be even more concerning.

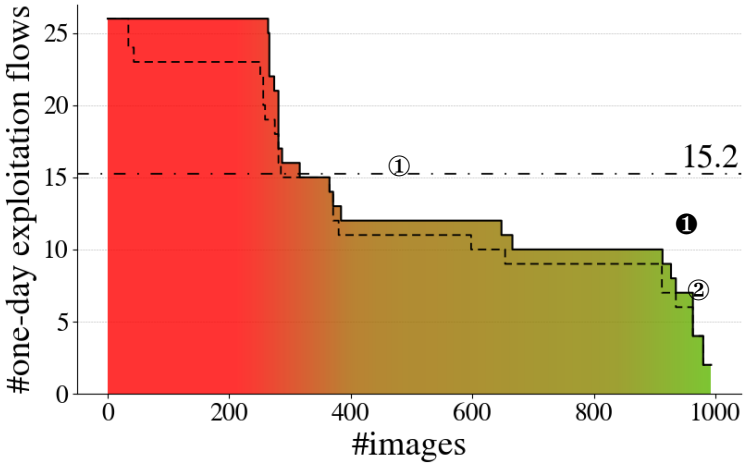
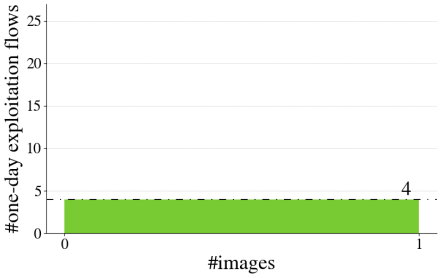


Figure 8.9: Susceptible one-day exploitation flows of all device images. While ① indicates that 281 images are susceptible to 21 or more one-day exploitation flows and ② indicates 913 images to 10 or more, the ❶ line represents the average susceptibility of 15.2 exploitation flows of all 994 images.

5.3.1. Analysis Results

We fully automate the detection of included defenses. Table 8.5 presents the defenses included for each vendor's firmwares. Our results indicate a lack of basic defenses (e.g., PAN and KASLR) and a significant lack of defenses against one-day exploit flows. In particular, significant portions of the firmwares do not include defenses such as `CONFIG_DEBUG_LIST`, which is critical to mitigate BadBinder [48].

Susceptibility. Using data from Section 5.2 and Table 8.2, we evaluate the effectiveness and assess the susceptibility of firmwares to one-day exploitation flows. We consider a firmware to be susceptible to a one-day exploitation flow if it does not include a defense that can prevent the vulnerability-agnostic exploitation flow. Figure 8.9 illustrates the overall susceptible one-day exploitation flows per firmware with two curves. The dashed line depicts the impact of the widespread defense lack, while the outer line incorporates both the lack and efficacy shortcomings (see Section 5.2), providing a more comprehensive view. Without these shortcomings, on average, nearly two one-day exploitation flows could have been prevented. Both findings highlight the worrying situation and lack of effective defenses to prevent exploitation flows.

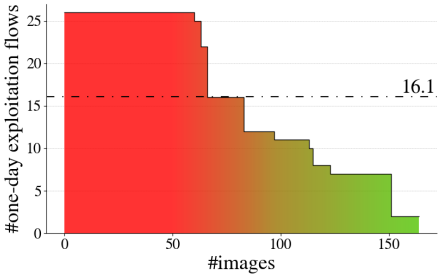


(a) Ground truth

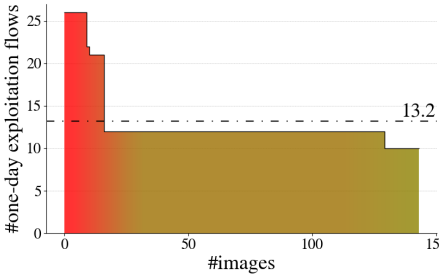
#	Vendor
1	Google
2	Realme
3	OnePlus
4	Xiaomi
5	Vivo

#	Vendor
6	Samsung
7	Motorola
8	Huawei
9	Oppo
10	Fairphone

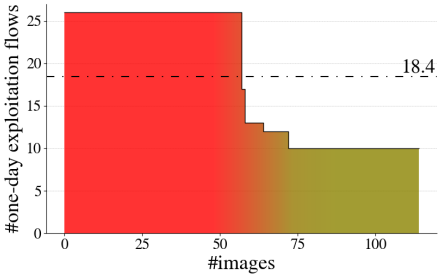
(b) Ranking



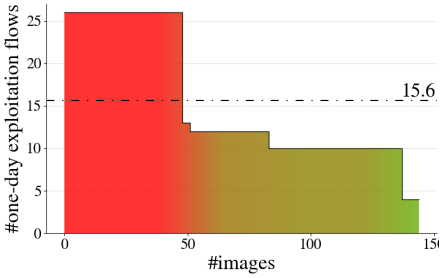
(c) Samsung



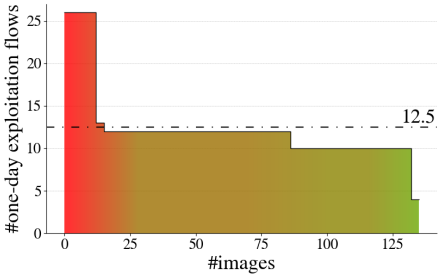
(d) Xiaomi



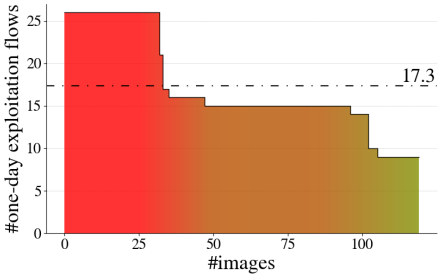
(e) Oppo



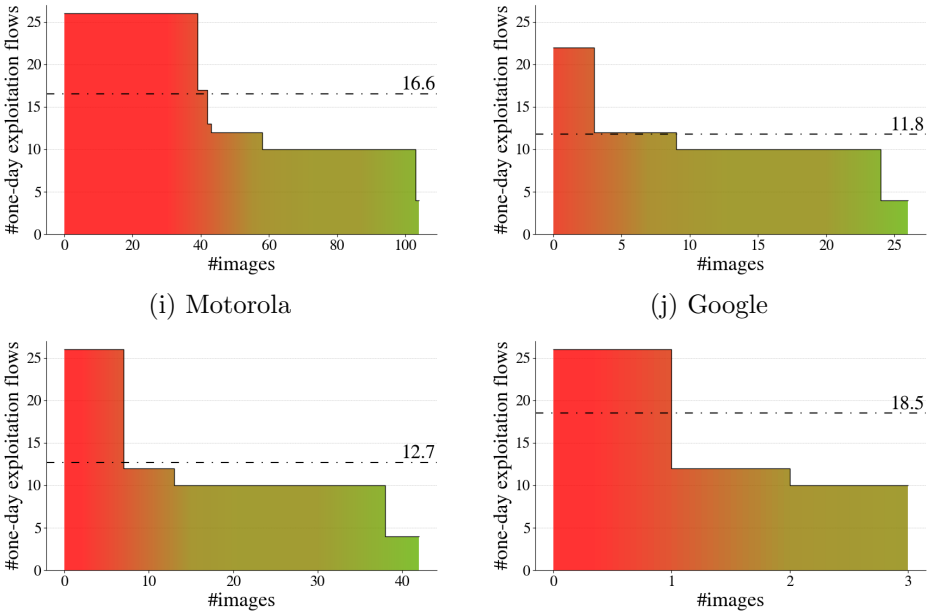
(f) Vivo



(g) Realme



(h) Huawei



(i) Motorola (j) Google

(k) OnePlus (l) Fairphone

Figure 8.10: Analysis results per devices for each vendors.

Takeaway 8.1

Even though effective defenses (see Table 8.2) for a large share of the one-day exploitation flows are available, they are rarely activated in vendor-provided kernels.

Susceptibility per Vendor. We further organize the results by vendor, presenting each in Figure 8.10. Figure 8.10a depicts the ground truth, showcasing the maximum achievable security with all available mainline defenses. Figure 8.10c-8.10l show each vendor's susceptible one-day exploitation flows, including the lack of defenses and efficacy shortcomings. We specifically highlight Google, Fairphone, and Samsung, representing the most and least secure, and with the highest market share. Their susceptibility is 11.8, 18.5, and 16.1, respectively, while the ground truth has 4. We compute the factor by which they are worse than the ground truth, resulting in 2.95 ($\approxeq \frac{11.8}{4}$), 4.62, and 4. Figure 8.10b presents the ranking of vendors according to this deterioration factor.

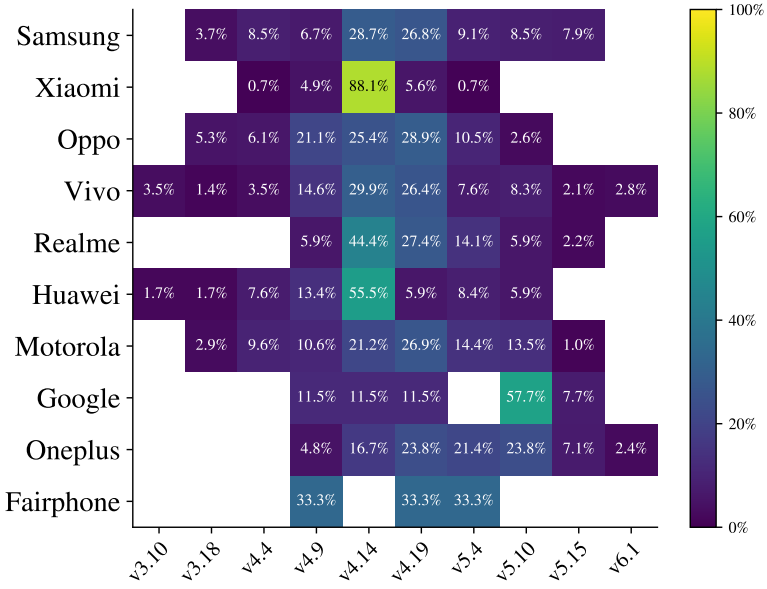


Figure 8.11: Applied Android kernel versions for each vendor.

Takeaway 8.2

Protection against one-day exploitation flows is highly vendor dependent, varying between a 4.62 to 2.95 worse scenario than applying all available mainline defenses.

Susceptibility per Kernel Version. To illustrate a version dependency, we initially obtain the used kernel versions. Figure 8.11 shows the results covering v3.10 to v6.1. For context, v4.19 was released in 2018, while v3.10 was released in 2014. We then analyze the susceptibility to one-day exploitation flows, organized by kernel version and vendor (see Figure 8.12). The figure includes a ground truth, representing how many exploitation flows remain susceptible for a given kernel version with all available defenses integrated (see Table 8.6).

Three findings emerge from this analysis: First, almost no device kernel prior to v4.14 includes any defenses. Since ret2bpf (**ET5**) is not exploitable on v3.10, it may be less susceptible than v3.18. Second, newer kernels tend to have more active protection against exploitation flows, observed across almost all vendors. This is particularly true for those obeying the GKI constraints ($\geq$v5.4 for GKI-1.0 or $\geq$v5.10 for GKI-2.0). Third, although

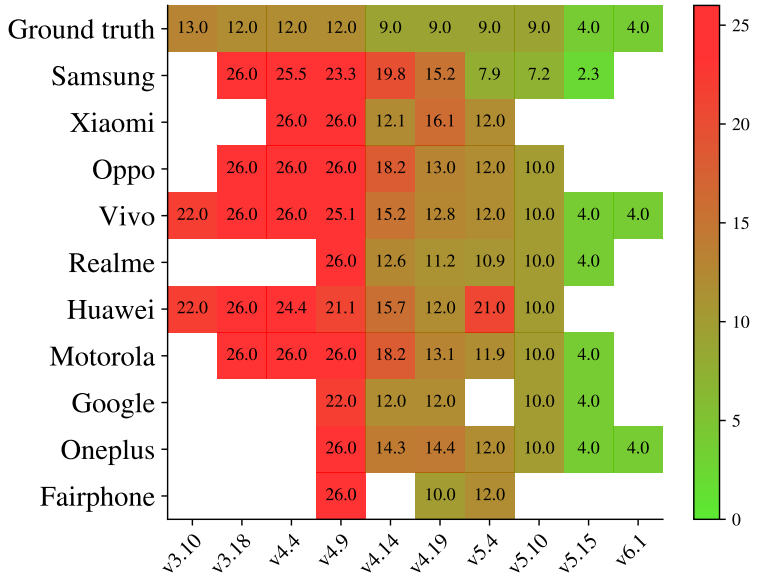


Figure 8.12: Susceptible exploitation flows per version/vendor.

newer kernels provide more defenses, a v3.10 kernel with all available defenses enabled would protect more flows than 38.1 % of our analyzed kernels.

Takeaway 8.3

While newer kernels provide more defenses, a v3.10 kernel with all available defenses enabled would mitigate more exploitation flows than 38.1 % of vendor-supplied kernels.

Susceptibility per Low/High-End Device. We differentiate the susceptibility according to whether it is a low-end or a high-end device: We initially compute the average one-day susceptibility of the latest low-end and high-end devices from vendors offering both classes, i.e., all except Google and Fairphone (see Table 8.7). We then compute the susceptibility reduction of high-end compared to low-end devices. For instance, with a susceptibility score of 4.5 and 5.5 for high-end and low-end Samsung devices, respectively, the reduction is 18.2 %. Overall, the reduction is between 0 % to 63.6 %, with an average value of 23.8 %, which indicates a significant reduction of high-end to low-end devices.

Takeaway 8.4

There is a significant gap of 23.8 % between the one-day susceptibility of high-end and low-end devices.

6. Discussion

Factors Potentially Contributing to the Absence of Effective Defenses. Our analysis, highlighted in Takeaway 8.1, reveals a concerning reality: vendors lack the inclusion and effectiveness of defenses against one-day exploitation flows. Here we discuss potential factors contributing to this situation.

First, as indicated by Takeaway 8.2, there is variability in susceptibility to one-day exploitation flows across vendors. While Google and OnePlus demonstrate lower susceptibility, others like Huawei show higher ones. As these vendors utilize different kernel versions, we observe a correlation between higher susceptibility and the use of older versions. Hence, a potential contributing factor is the *use of older kernel versions*.

Second, as emphasized in Takeaway 8.3, susceptibility extends beyond mere kernel version correlation. Even the deprecated kernel v3.10 (released about ten years ago) would mitigate more one-day exploitation flows, if properly configured, than 38.1 % of vendor firmwares. Huawei underscores this statement with their v5.4.86 kernels, nearly twice as bad as the properly configured v3.10. This lack of proper configuration appears prevalent across multiple vendors. Hence, the second potential contributor is *a lack of importance regarding security-relevant features for the Android kernel*.

Third, as shown in Takeaway 8.4, we observe that low-end are more susceptible to one-day exploitation flows than high-end devices, as observed by most vendors. On the one hand, low-end devices tend to be less powerful than high-end devices, and on the other hand, enabling defenses increases the performance overhead. To compensate for this performance cost, vendors may deliberately not enable defenses for performance gains. Therefore, the third potential factor is *performance cost, especially for less powerful low-end devices*.

Recommendation to Improve Android Security. With these insights, we propose that Google updates the Android Compatibility Definition Document (CDD), which outlines the requirements for devices to be compatible with Android. While for Android 14 some fundamental defenses are recommended (e.g., `CONFIG_CFI_CLANG`) or required (e.g., `CONFIG_STRICT_KERNEL_RWX`), other critical ones are absent (e.g., `CONFIG_DEBUG_LIST`). By including our findings, we anticipate a substantial improvement in Android security.

Responses. Google responded that they are aware of this problem and are gradually enforcing kernel defenses that will be integrated. However, as defenses can come at a performance cost, enforcing them across all vendors is difficult, especially for low-end devices. They pointed out that `CONFIG_DEBUG_LIST` has been enforced in the past, but vendors complained about the performance hit. This resulted in critical defenses not being integrated. Samsung and Huawei responded similarly, as integration comes at a performance cost, i.e., Samsung for not activating RKP on all (especially low-end) devices and Huawei for not protecting all page tables. These responses highlight our third potential contribution factor. Fairphone and Motorola acknowledged our findings and integrated defenses, while the others did not respond.

Automation and Standardization. Fully automating the analysis process would enhance the demonstration of the effectiveness of defenses. We have already automated several steps, such as parts of the firmware and kernel code acquisition, metadata extraction, and defense analysis, all of which are scalable. Challenges remain in the acquisition and analysis of zero-days and the acquisition of all firmware. Our work addresses these challenges manually and encourages standardization, drastically reducing manual effort. Therefore, our work addresses current technical challenges and encourages progress for future identification of effectively integrated defenses, ultimately improving Android security.

False Negatives/Positives. A false negative occurs when we interpret a device as being susceptible to an exploitation flow when it is not. This could have happened if we have overlooked defenses. To ensure we identified all mainline defenses, we executed each exploitation technique (**ET1-10**) with security measures enabled, resulting in the defenses (**DM1-8**) preventing these exploits. To ensure that we have identified all downstream defenses, we performed a semi-automated analysis of the 1533 downstream kernels in Section 5.2.2, which yielded 3 vendor-specific defenses. While misinterpreted firmware analysis could also lead to false negatives, most defenses

are intrinsically tied to the kernel's core functionalities. As described in Section 5.2.1, those defenses that are not intrinsically tied can only lead to false positives, i.e., we interpret a device as mitigating an exploitation flow when it does not. This means our results can be interpreted as conservative, and the real world may be even more worrying.

7. Related Work

Large-scale Firmware Analysis. Possemato et al. [44] investigated compliance with Android's compatibility guidelines and found customizations as security drawbacks. Subsequent work [23] has highlighted delays in adopting critical patches. Other studies scanned ROMs for insecure access policies [22, 45] or privacy-intruding apps [19, 23, 53]. For embedded systems, researchers have uncovered vulnerabilities at a large scale [14, 18] and revealed a reluctance to activate attack mitigations in Linux-based IoT devices [65].

Android Security Patch Ecosystem. Prior works studied the deployment of security updates to Android systems. Wu et al. [58] noted that most Android Security Bulletin (ASB) issues stem from native code. Farhang et al. [17] found that CVEs in the kernel took the longest to propagate to vendor ASBs, while other researchers [29, 68] reported weeks to months of delay in deploying Android security updates.

Patch Detection. Researchers proposed strategies to detect patches in kernel binaries. Zhang et al. [67] presented a detection approach by deriving a signature from the mainstream version, which is then compared with target kernels. PDiff [27] statically extracts the semantics of source-level patches and uses a similarity-based measure to detect patches in compiled kernels. Dynamic approaches [26, 69] automatically adapt existing PoC exploits to different kernel variants.

Vulnerability Patching. Researchers have proposed solutions to address the long delays in kernel patch deployment. Wang et al. [57] prevented bugs discovered by a sanitizer from being triggered and, hence, exploited till a patch is available. Talebi et al. [54] instrumented vulnerable syscall implementations to undo harmful side-effects. Other researchers focused on downstream Android kernels. Chen et al. [13] proposed hot-patching

with Lua code to filter vulnerable function arguments. Xu et al. [62] extended this by suggesting automated binary hot patches from source-level upstream fixes.

Zero-Day Analysis. Google Project Zero [9, 25, 47] and Threat Analysis Group [51] hunt for zero-days in the wild. They release public findings covering various entities, e.g., Android phones, significantly enhancing system security.

8. Conclusion

This work conducted a one-day analysis of Android devices, combined with an analysis of defense inclusion and effectiveness. Our findings unveiled a significant gap between the current state of Android security and its maximum potential. We discussed potential contributing factors and offered recommendations for improvement, enhancing Android security.

Acknowledgements

We thank Mathias Oberhuber, Andreas Kogler, the anonymous reviewers, and our shepherd for their valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Any opinions, findings, 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 (p. 276).

[2] Al Viro. iov_iter: saner checks on copyin/copyout. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=09fc68dc66f7597bdc8898c991609a48f061bed5 (p. 270).

[3] Android. Kernel Control Flow Integrity. 2022. URL: https://source.android.com/docs/security/test/kcfi (p. 276).

[4] Android. Overview. 2024. URL: https://source.android.com/docs/core/architecture/partitions (p. 284).

[5] AppBrain. Top Android OS versions. accessed: 28.11.2023. 2023. URL: https://web.archive.org/web/20231128122419/https://www.appbrain.com/stats/top-android-sdk-versions (p. 278).

[6] AppBrain. Top manufacturers. accessed: 14.09.2023. 2023. URL: https://web.archive.org/web/20230915054021/https://www.appbrain.com/stats/top-manufacturers (pp. 261, 278).

[7] Brandon Azad. A survey of recent iOS kernel exploits. 2020. URL: https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html (pp. 260, 266, 269).

[8] Brandon Azad. An iOS hacker tries Android. 2020. URL: https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html (p. 272).

[9] Ian Beer. Mind the Gap. 2022. URL: https://googleprojectzero.blogspot.com/2022/11/ (pp. 260, 275, 282, 294).

[10] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: ACM Conference on Computer and Communications Security (CCS). 2008 (p. 272).

[11] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS. 2016 (pp. 285, 301).

[12] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In: USENIX Security. 2020 (p. 266).

[13] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android Kernel Live Patching. In: USENIX Security. 2017 (pp. 260, 293).

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In: USENIX Security. 2014 (pp. 285, 293, 301).

[15] Samsung Knox Documentation. Real-time Kernel Protection (RKP). 2023. URL: https://docs.samsungknox.com/admin/fundamentals/whitepaper/core-platform-security/real-time-kernel-protection/ (pp. 268, 276, 282, 283).

[16] Everphone. What is the average smartphone lifespan? 2023. URL: https://web.archive.org/web/20231123081219/https://everphone.com/en/blog/smartphone-lifespan/ (p. 278).

[17] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An Empirical Study of Android Security Bulletins in Different Vendors. In: WWW. 2020 (p. 293).

[18] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In: CCS. 2016 (p. 293).

[19] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An Analysis of Pre-installed Android Software. In: S&P. 2020 (p. 293).

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (p. 263).

[21] Hemanth. Extractor of SpreadTrum firmware files with extension pac. 2023. URL: https://github.com/HemanthJabalpuri/pacextractor (p. 284).

[22] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R. B. Butler. BIGMAC: fine-grained policy analysis of android firmware. In: USENIX Security. 2020 (p. 293).

[23] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. Large-scale security measurements on the android firmware ecosystem. In: International Conference on Software Engineering (ICSE). 2022 (pp. 263, 293).

[24] Huawei. EMUI 11.0 Security Technical White Paper. 2020. URL: https://consumer.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/emui_11.0_security_technical_white_paper_v1.0.pdf (pp. 277, 284).

[25] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack. 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html (pp. 277, 294).

[26] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In: S&P. 2023 (p. 293).

[27] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In: CCS. 2020 (pp. 260, 293).

[28] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel. 2021. URL: https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf (p. 272).

[29] Kailani R. Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G. Bardas. Deploying Android Security Updates: an Extensive Study Involving Manufacturers, Carriers, and End Users. In: CCS. 2020 (p. 293).

[30] Max Kellermann. The Dirty Pipe Vulnerability. 2022. URL: https://dirtypipe.cm4all.com/ (p. 274).

[31] Bjoern Kerler. oppo_decrypt. 2023. URL: https://github.com/bkerler/oppo_decrypt (p. 284).

[32] Bjoern Kerler. oppo_decrypt_ozip. 2022. URL: https://github.com/bkerler/oppo_ozip_decrypt (p. 284).

[33] Zhenpeng Lin. How AUTOSLAB Changes the Memory Unsafety Game. 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game (p. 266).

[34] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In: ACM. 2022 (p. 274).

[35] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android. 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf (pp. 269, 284).

[36] Linux Kernel Driver DataBase. CONFIG_ARM64_UAO: Enable support for User Access Override (UAO). 2024. URL: https://cateee.net/lkddb/web-lkddb/ARM64_UAO.html (p. 282).

[37] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In: NDSS. 2017 (p. 277).

[38] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024 (p. 257).

[39] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 266, 274).

[40] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (p. 281).

[41] Marin. vmlinux-to-elf. 2023. URL: https://github.com/marin-m/vmlinux-to-elf (p. 285).

[42] Man Yue Mo. One day short of a full chain: Part 1 - Android Kernel arbitrary code execution. 2021. URL: https://securitylab.github.com/research/one_day_short_of_a_fullchain_android/ (p. 269).

[43] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html (pp. 266, 271).

[44] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In: S&P. 2021 (pp. 260, 293).

[45] Zeinab El-Rewini and Yousra Aafer. Dissecting Residual APIs in Custom Android ROMs. In: CCS. 2021 (p. 293).

[46] Matteo Rizzo and Jann Horn. Prevent cross-cache attacks in the SLUB allocator. 2023. URL: https://lore.kernel.org/linux-mm/202309151425.2BE59091@keescook/T/ (pp. 267, 277).

[47] Maddie Stone. 2022 0-day In-the-Wild Exploitation...so far. 2023. URL: https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html (pp. 260, 269, 294).

[48] Maddie Stone. Bad Binder: Android In-The-Wild Exploit. 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html (pp. 260, 270, 286).

[49] Maddie Stone. CONFIG_DEBUG_LIST=y. 2020. URL: https://twitter.com/maddiestone/status/1245834936629616640?lang=de (p. 260).

[50] Maddie Stone. Detection Deficit: A Year in Review of 0-days Used In-The-Wild in 2019. 2020. URL: https://googleprojectzero.blogspot.com/2020/07/detection-deficit-year-in-review-of-0.html (p. 269).

[51] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022. 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html (pp. 260, 294).

[52] Superr. splituapp. 2019. URL: https://github.com/superr/splituapp (p. 284).

[53] Thomas Sutter and Bernhard Tellenbach. FirmwareDroid: Towards Automated Static Analysis of Pre-Installed Android Apps. In: MOBILESoft. 2023 (p. 293).

[54] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo Workarounds for Kernel Bugs. In: USENIX Security. 2021 (p. 293).

[55] USA Today. How long before a phone is outdated? Here's how to find your smartphone's expiration date. 2023. URL: https://web.archive.org/web/20231022153016/https://eu.usatoday.com/story/tech/columnist/komando/2023/10/22/how-to-find-smartphone-expiration-date/71255625007/ (p. 278).

[56] Vasya. payload dumper. 2023. URL: https://github.com/vm03/payload_dumper (p. 284).

[57] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel. In: USENIX Security. 2023 (p. 293).

[58] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In: AsiaCCS. 2019 (p. 293).

[59] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html (pp. 269, 274, 283).

[60] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: USENIX Security. 2019 (p. 277).

[61] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (pp. 266, 274–276).

[62] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic Hot Patch Generation for Android Kernels. In: USENIX Security. 2020 (pp. 260, 294).

[63] Jun Yao. arm64/mm: move {idmap_pg_dir,tramp_pg_dir,swapper_pg_dir} to .rodata section. 2018. URL: https://patchwork.kernel.org/project/linux-hardening/patch/20180620085755.20045-2-yaojun8558363@gmail.com/ (p. 276).

[64] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features. 2018. URL: https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf (p. 273).

[65] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building Embedded Systems Like It's 1996. In: NDSS. 2022 (pp. 285, 293, 301).

[66] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In: USENIX Security. 2022 (pp. 264, 266).

[67] Hang Zhang and Zhiyun Qian. Precise and Accurate Patch Presence Test for Binaries. In: USENIX Security. 2018 (p. 293).

[68] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An Investigation of the Android Kernel Patch Ecosystem. In: USENIX Security. 2021 (pp. 260, 263, 293).

[69] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem. In: NDSS. 2024 (p. 293).

9. Appendix

9.1. Detailed Statistics

9.1.1. Detailed Defense Detection of Kernels

Table 8.3 illustrates the comprehensive list of how we assess the state of our identified defense mechanisms. We follow the procedure to identify symbols of globally reachable functions within the `kallsyms` file. This file contains all globally reachable functions and variable symbols used in the kernel binary, e.g., marked with `EXPORT_SYMBOL`. For instance, the presence of `__list_add_valid` in `kallsyms` indicates the status of the `CONFIG_DEBUG_LIST`. As another example, the symbol `cache_random_seq_create` indicates the presence of `CONFIG_SLAB_FREELIST_RANDOM`. A similar assessment stands true for detecting the other defenses. Additionally, to identifying symbols `kallsyms`, our approach also detects defense mechanisms which do not contain globally reachable symbols. For instance, `CONFIG_SLAB_FREELIST_HARDENED` only includes inline functions and member variables. To detect the presence of this defense, our approach analyzes the kernel binary, more specifically, the function where these inline calls are executed, e.g., `get_random_long` within function `kmem_cache_open`. Executing the call indicates the presence of this defense. To detect the presence of the KSMA protection, `swapper_pg_dir` and `tramp_pg_dir` must also be mapped read-only. For instance, these pages might be mapped between `__start_rodata` and `__init_begin`.

9.1.2. Statistical Results of Firmware Extraction

Table 8.4 illustrates the extractable firmwares. Our success rate of 58.5 % (with a collection and extraction rate of 65.3 % and 89.6 %) from produced devices to extractable firmwares aligns with prior work [11, 14, 65]. The two main reasons for extraction failure were that our approach did not recognize the correct format or that part of the firmware was corrupted.

Table 8.2: Mitigation of one-day exploits, using ✓ to indicate defenses that prevent used exploitation techniques. Conversely, ✗ indicates ineffective defenses (see Sections 5.2.3, 5.2.4 and 5.2.6). Samsungs's defenses ✗/✶ are either ineffective or only effective in certain variants (see Section 5.2.5).

CVE	DM1	DM2	DM3	DM4	DM5	DM6	DM7	DM8	DM9	DM10
CVE-2019-2215	✓	✗	✓							✓
CVE-2019-2025	✓		✓							
CVE-2020-0030	✓	✗	✓							✓
CVE-2021-1968,-1969,-1940				✓	✓				✗	
CVE-2021-0920	✓		✓							
CVE-2021-1905				✓	✓				✗	
CVE-2022-22265			✓							
CVE-2021-25369,-25370		✗	✓	✗					✗	✓
CVE-2016-3809,-2021-0399			✓	✓	✓				✗	
CVE-2022-20409			✓							
CVE-2023-21400			✓						✶	✗
CVE-2022-28350									✶	✗
CVE-2020-29661									✶	✗
CVE-2021-22600			✓							
CVE-2020-0423	✓							✗	✓	✓
CVE-2022-22057						✓		✗	✓	✓
CVE-2023-26083,-0266				✗					✗	
CVE-2020-0041	✓		✓							
CVE-2019-2205	✓		✓							
CVE-2019-2025	✓		✓					✗	✓	✓
CVE-2020-3680	✓		✓					✗	✓	✓
CVE-2022-20421			✓							
CVE-2022-0847							✓			
CVE-2021-4154										
CVE-2021-38001				✓	✓				✗	
NO_NUMBER (~2021)			✓			✓				

DM1: CONFIG_DEBUG_LIST **DM2:** CONFIG_ARM64_UAO **DM3:** kmalloc-cg-* **DM4:** CONFIG_CFI_CLANG **DM5:** CONFIG_BPF_JIT_ALWAYS_ON **DM6:** CONFIG_SLAB_FREELIST_HARDENED **DM7:** CONFIG_INIT_ON_ALLOC_DEFAULT_ON **DM8:** KSMA protection **DM9:** Samsung RKP **DM10:** Huawei HKIP

Table 8.3: Symbols and used additional information for our defense detection approach. The defense feature ☆ is enabled if this symbol (e.g., globally visible function or variable) is present within the `kallsyms` containing all kernel symbols.

Defense Feature	Kernel Executable	Present within `kallsyms`☆	Information
CONFIG_DEBUG_LIST		__list_add_valid, __list_del_entry_valid __list_add, __list_del_entry	≥v3.18 <v3.18
CONFIG_CFI_CLANG		cfi_module_add, cfi_module_remove	
CONFIG_BPF_JIT_ALWAYS_ON		___bpf_prog_run __bpf_prog_run	≥v4.14 <v4.14
kmalloc-cg-*			available for ≥v5.13
CONFIG_INIT_ON_ALLOC_DEFAULT_ON		init_on_alloc	
CONFIG_ARM64_UAO		uao_thread_switch, cpu_enable_uao	no addr_limit for ≥v5.11
CONFIG_SLAB_FREELIST_HARDENED	get_random_long		called within __kmem_cache_create
KSMA Protection	swapper_pg_dir, tramp_pg_dir	swapper_pgdir_lock, swapper_pg_dir, tramp_pg_dir	*_pg_dir mapped as read-only
Samsung RKP		rkp_init	only for Samsung devices
CONFIG_RANDOMIZE_BASE		module_alloc_base, kaslr_early_init	
CONFIG_STRICT_KERNEL_RWX		set_debug_rodata, mark_readonly, mark_rodata_ro	named as CONFIG_DEBUG_RODATA for <v4.14; on v3.10 only for 32 bit systems
CONFIG_ARM64_PAN		cpu_enable_uao	
CONFIG_ARM64_SW_TTBR0_PAN	"emulated: Privileged Access Never (PAN) \ using TTBR0_EL1 switching"	reserved_ttbr0	<v5.4; available for ≥v4.10 ≥v4.19
CONFIG_SLAB_FREELIST_RANDOM		cache_random_seq_create, cache_random_seq_destroy	
CONFIG_UNMAP_KERNEL_AT_EL0		tramp_pg_dir	

Table 8.4: Statistical results of firmware extraction and kernel code collection.

Vendors	Firmware Extraction			Kernel Code
	#devices	#available	#extracted	#collected
Samsung	197	190	164	654
Xiaomi	278	151	143	188
Oppo	229	145	114	29
Vivo	307	178	144	30
Realme	307	137	135	135
Huawei	182	121	119	218
Motorola	115	112	104	246
Google	26	26	26	9
OnePlus	54	46	42	21
Fairphone	3	3	3	3
Total	1698	1109	994	1533

Table 8.5: Included defenses averaged over all firmwares for each vendor. ✶ inidcates that it is ineffective while ☆ indicates that it is ineffective for kernels <v5.11.

Vendor	CONFIG_DEBUG_LIST	CONFIG_BPF_JIT_ALWAYS_ON	CONFIG_CFI_CLANG	kmalloc-cg-*	CONFIG_INIT_ON_ALLOC_DEFAULT_ON	CONFIG_ARM64_UAO☆	CONFIG_SLAB_FREELIST_HARDENED	KSMA protection✶	CONFIG_RANDOMIZE_BASE	CONFIG_STRICT_KERNEL_RWX	CONFIG_ARM64_(SW_TTBRO_)PAN	CONFIG_SLAB_FREELIST_RANDOM	CONFIG_UNMAP_KERNEL_AT_EL0	Samsung RKP	Huawei HKIP
Samsung	60	49	26	8	63	96	25	5	84	100	91	27	16	39	
Xiaomi	89	94	74	0	93	98	10	1	97	100	97	10	83		
Oppo	50	49	19	0	44	95	37	13	91	100	95	49	14		
Vivo	69	65	27	5	67	96	44	22	95	98	88	73	22		
Realme	91	91	34	2	89	100	36	22	100	100	99	47	44		
Huawei	15	18	67	0	20	92	12	13	97	100	79	87	14		62
Motorola	62	58	34	1	59	90	44	29	89	100	79	58	31		
Google	88	88	100	8	88	100	65	65	100	100	100	77	65		
Oneplus	83	83	52	10	83	100	69	55	100	100	100	90	55		
Fairphone	67	67	33	0	67	100	33	33	100	100	100	67	33		

Table 8.6: ✓ indicates defenses available for mainline Android kernel from v3.10 to v6.1, while ✗ indicates that the defense is not required for the specific version.

Kernel	CONFIG_DEBUG_LIST	CONFIG_BPF_JIT_ALWAYS_ON	CONFIG_CFI_CLANG	kmalloc-cg-*	CONFIG_INIT_ON_ALLOC_DEFAULT_ON	CONFIG_ARM64_UAO	CONFIG_SLAB_FREELIST_HARDENED	KSMA protection	CONFIG_RANDOMIZE_BASE	CONFIG_STRICT_KERNEL_RWX	CONFIG_ARM64_(SW_TTBR0_)PAN	CONFIG_SLAB_FREELIST_RANDOM	CONFIG_UNMAP_KERNEL_AT_EL0
v3.10	✓	✗								✓			
v3.18	✓	✓				✓			✓	✓	✓		
v4.4	✓	✓				✓			✓	✓	✓		✓
v4.9	✓	✓	✓			✓			✓	✓	✓	✓	✓
v4.14	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓
v4.19	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓
v5.4	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
v5.10	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
v5.15	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
v6.1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

Table 8.7: The susceptibility reduction (i.e., **Susc Reduc**) against one-days of high-end to low-end devices.

Vendor	Low-End		High-End		Susc Reduc
	Devices	Susc	Devices	Susc	in %
Samsung	Galaxy A(1,2,3,5)4	5.5	Galaxy S23.*	4.5	18.2
Xiaomi	Redmi 12.*	12.0	13T.*	12.0	0.0
Oppo	A(3,9)8	12.0	Find X2.*	10.0	16.7
Vivo	Y(100,27)	11.0	X100.*	4.0	63.6
Realme	C(33,53,55)	10.7	Neo 5.*	10.0	6.2
Huawei	Nova 11.*	15.5	P60.*	10.0	35.5
Motorola	G(1,5,8)4.*	10.7	Edge 40.*	8.5	20.3
OnePlus	Nord 3.*	10.0	11.*	7.0	30.0
Mean					23.8

9

KernelSnitch: Side-Channel Attacks on Kernel Data Structures

Publication Data

Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025

Contributions

The author of this thesis is the main author of this work. The author's contributions are the proposal of *identification of critical timing side channels in the kernel* and *leakage amplification and evaluation*. From *side-channel attacks*, the author's contributions are the covert channel and the kernel heap pointer leak. Finally, the author's contributions are also most of the written text.

KernelSnitch: Side-Channel Attacks on Kernel Data Structures

Lukas Maar Jonas Juffinger Thomas Steinbauer Daniel Gruss
Stefan Mangard

Graz University of Technology

Abstract

The sharing of hardware elements, such as caches, is known to introduce microarchitectural side-channel leakage. One approach to eliminate this leakage is to not share hardware elements across security domains. However, even under the assumption of leakage-free hardware, it is unclear whether other critical system components, like the operating system, introduce software-caused side-channel leakage.

In this paper, we present a novel generic software side-channel attack, KernelSnitch, targeting kernel data structures such as hash tables and trees. These structures are commonly used to store both kernel and user information, e.g., metadata for user-space locks. KernelSnitch exploits that these data structures are variable in size, ranging from an empty state to a theoretically arbitrary amount of elements. Accessing these structures requires a variable amount of time depending on the number of elements, i.e., the occupancy level. This variance constitutes a timing side channel, observable from user space by an unprivileged, isolated attacker. While the timing differences are very low compared to the syscall runtime, we demonstrate and evaluate methods to amplify these timing differences reliably. In three case studies, we show that KernelSnitch allows unprivileged and isolated attackers to leak sensitive information from the kernel and activities in other processes. First, we demonstrate covert channels with transmission rates up to 580 $\frac{\text{kbit}}{\text{s}}$. Second, we perform a kernel heap pointer leak in less than 65 s by exploiting the specific indexing that Linux is using in hash tables. Third, we demonstrate a website fingerprinting attack, achieving an F_1 score of more than 89 %, showing that activity in other user programs can be observed using KernelSnitch. Finally, we discuss mitigations for our hardware-agnostic attacks.

1. Introduction

The performance of modern computer systems crucially depends on the efficiency of hardware and software. On the hardware level, numerous optimizations, such as caching, contribute significantly to hardware performance. Instead of always taking the slow path to the main memory, caches offer a shortcut by providing a local copy of the data. Inherently, this introduces a timing difference. Side-channel attacks exploit specifically such timing differences [37], allowing an attacker to infer secret information and, e.g., covertly transmit data [46], break Address Space Layout Randomization (ASLR) [28], leak cryptographic keys [69], or spy on user input [24]. Besides caches, numerous other optimizations have been discovered to leak information through timing, e.g., contention of execution ports [2] or execution unit schedulers [18, 19]. An intuitive approach to eliminate all microarchitectural side-channel attacks, commonly considered a last resort, is to not share hardware elements across security domains anymore.

However, even under the assumption of leakage-free hardware, the software can also introduce timing side channels for the same reason: improving efficiency. The exploitation of timing differences has been studied on algorithms designed for security contexts, e.g., in weak cryptographic implementations [49], as well as on algorithms that were not primarily designed for security contexts but used in them [6, 21, 33, 54, 56]. Timing differences can also be introduced generically at the system level: For instance, the software also has different types of caches that leak information [16, 22, 64], in-kernel allocators or synchronization primitives [41, 43, 57], interrupts [15, 60] or variation in the instruction or memory access sequence [63, 69]. Although the concept of constant-time code [37] is well-understood, its general-purpose application is impractical [55]. In particular, leakage introduced on the operating system level [22, 41, 43, 57, 60] is critical, as this leakage is not visible in the user-level source code and is not mitigated by constant-time code in the user application. Hence, it is unclear whether, despite leakage-free hardware and hardened security-critical algorithms, other critical system components, like the operating system, introduce generic software-observable side-channel leakage.

Recent research highlights leakage from system components: Gruss et al. [22] showed that the operating system page cache can be exploited like hardware caches. Patel et al. [50] demonstrated a performance-degrading attack exploiting intra-kernel resource contention. Lee et al. [41]

and Maar et al. [43] presented timing side channels in the Linux slab allocator to infer when a new slab is created. Shen et al. [57] presented a covert channel based on mutual exclusion primitives. These works motivate further research on side channels introduced by the operating system architecture to understand whether more security- and privacy-critical attacks are possible.

In this paper, we present a novel generic side-channel attack, KernelSnitch, that targets data container structures inside the kernel. We present attacks on four types of data structures in the kernel: fixed-size hash tables, dynamically resizable hash tables, radix trees, and red-black trees, all commonly used data structures in the Linux kernel. KernelSnitch exploits that these data structures have a variable size, ranging from an empty state to a theoretically arbitrary amount of elements. Accessing these data structures requires operations that depend on the amount of elements in the data structure, i.e., the occupancy level. This variance in the operations, which depends on the occupancy level, constitutes a timing side channel an unprivileged user can observe to leak information about other processes or privileged information from the kernel.

One challenge for KernelSnitch is amplification: Timing differences between occupancy levels of the victim workload can be very low, e.g., 8 additional instructions. Compared to the syscall overhead, distinguishing the timing of these extra instructions is challenging. We demonstrate two approaches to address this challenge by increasing the timing difference and, thereby, amplifying the leakage: First, we degrade the performance of the system with memory thrashing. Second, we manipulate the data structures with additional elements. We evaluate the KernelSnitch's leakage to distinguish occupancy levels with and without amplification and with different noise floors. We mainly evaluate on an Intel i7-1260P, with other processors and another architecture yielding similar results for the same evaluation programs, demonstrating hardware independence. We show that we can distinguish the occupancy level with an accuracy of better than 99.9 % on an idle system and better than 98.5 % on a noisy system up to ¾ full load.

We evaluate KernelSnitch in three case studies, showing that KernelSnitch can leak sensitive information from the kernel and activity in other user processes. Our evaluation works from unprivileged, isolated processes on the same system, e.g., within a sandbox. In our first case study, we measure the capacity of the side channel in a covert-channel scenario, achieving a

true capacity of 580 $\frac{\text{kbit}}{\text{s}}$ at an error rate of 2.8 %. In our second case study, we exploit the specific indexing of Linux for hash tables. To randomize the indices, the kernel generates the index by combining user-controlled information and kernel-controlled secret information. Using KernelSnitch, an attacker can infer the secret information that the kernel uses as input. This allows us to leak the locations of targeted kernel objects (i.e., `mm_struct` and `msg_msg`) in less than 65 s. We refer to this as a kernel heap pointer leak. While prior side-channel research [9, 23, 28, 29, 36] leaked Kernel ASLR (KASLR), e.g., the start of the text section or physical mapping, we are the first to perform a kernel heap pointer leak using a side channel[1]. In our third case study, we show the activity leakage of other user programs, e.g., Firefox. In particular, we perform a website fingerprinting attack on the Ahrefs Top 100 [1], achieving an F_1 score of 89.5 %.

Finally, we discuss defenses against the hardware-agnostic KernelSnitch attack. We identify challenges for efficient mitigation, such as the theoretically unbounded worst-case execution time and eliminating constant time as a viable solution.

Contributions. The main contributions of our work are:

1. **Identification of Critical Timing Side Channels in the Kernel:** We analyze the security properties of kernel data container structures, presenting a new side channel, KernelSnitch, exploiting the kernel's internal architecture to leak sensitive information to unprivileged users.
2. **Leakage Amplification and Evaluation:** We demonstrate information leakage amplification approaches and evaluate the leakage for multiple data container structures.
3. **Side-Channel Attacks:** We demonstrate three attack case studies: A covert channel with up to 580 $\frac{\text{kbit}}{\text{s}}$, a kernel heap pointer leak in less than 65 s, and a website fingerprinting attack with an F_1 score of more than 89 %.

Disclosure. We have disclosed our KernelSnitch attack to the Linux kernel security team.

[1] As Linus Torvalds and Kees Cook noted during our disclosure, KASLR is broken against local attackers, but leaking kernel heap pointers is not.

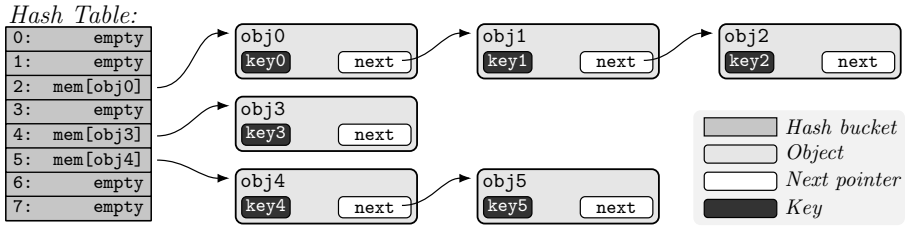


Figure 9.1: Visual representation of a Linux kernel's hash table.

Opensource. We provide open-source implementations of our timing side-channel attack, which leaks the occupancy level of data structures discussed in the paper. The code is available at `https://doi.org/10.5281/zenodo.14249716`.

Outline. Section 2 provides background information. Section 3 presents an overview of KernelSnitch. Section 4 presents a root cause analysis. Section 5 details how we amplify the leakage. Section 6 presents three attack case studies. Section 7 discusses related work, and Section 8 discusses mitigations. Section 9 concludes our work.

2. Background

In this section, we provide background on Linux kernel data container structures, including hash tables and trees.

2.1. Container Structures in the Linux Kernel

The C language used in Linux has no container structures like C++ (e.g., `vector`). Hence, several containers are explicitly written for Linux and optimized for their use in the kernel.

Double-Linked Lists. Linux provides a generic double-linked list, i.e., `list_head`, which is extensively used throughout the kernel. The `list_head` struct consists of `next` and `prev` pointers and is commonly used to organize lists, e.g., for the `mm_struct` or `task_struct` lists. Insertion and deletion work by re-linking pointers correspondingly, with a constant runtime. In the worst case, an object lookup requires iterating through the entire list, with a runtime linear in length.

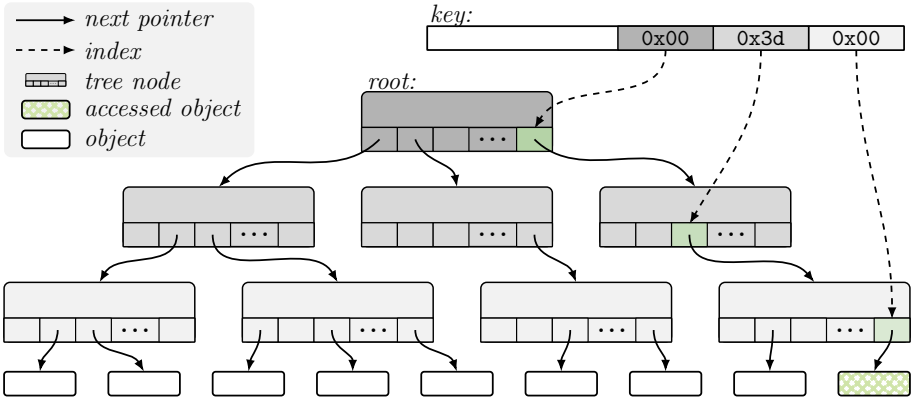


Figure 9.2: Visual representation of a three-level radix tree, with *key* referencing this tree to index the hatched object.

Hash Tables. In the Linux kernel, hash tables use a hash function to compute the index in an array of buckets. In each bucket, objects are typically stored in a linked list (see Figure 9.1). To access the object with key `key1`, the kernel computes its hash value to determine the hash bucket, resulting in bucket `2`. It then iterates through the linked list within bucket `2`, comparing the keys of stored objects until it finds a match with the key from `obj1`. While all hash tables in the Linux kernel use this approach to access objects by their keys, there are variations between *fixed-size* and *dynamically resizable* hash tables: Fixed-size hash tables have an array with a predetermined number of buckets, e.g., the `plist_head` object for the buckets, using `list_head` internally. Dynamically resizable hash tables, e.g., `rhashtable`, adjust their bucket sizes based on the occupancy level of the buckets.

Trees. Linux supports multiple trees, including two widely-used ones, i.e., *radix tree* [11] and *red-black tree* [12].

The *radix tree* associates a pointer value with an integer key, offering efficient memory usage and quick lookups [11]. Figure 9.2 illustrates a three-level radix tree example. In general, each tree node contains multiple slots (typically 6 bit), each pointing to `NULL`, a child tree node, or a stored object. These slots are indexed by parts of the integer key. During a key lookup, the kernel uses the most significant bit block to find the corresponding slot in the root node, followed by subsequent bit blocks for lower-level tree nodes. In the example of Figure 9.2, with the key having three bit blocks, a single tree lookup uses the most significant bit block

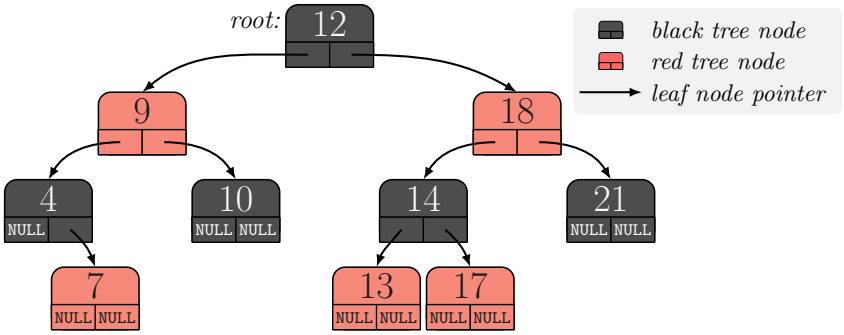


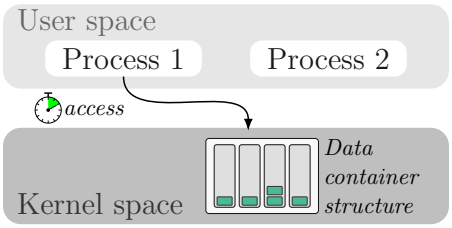
Figure 9.3: Visual representation of a red-black tree, which stores items with an increasing order.

(i.e., `0x00`) for the third level (i.e., root node), the middle bit block (i.e., `0x3d`) for the second level, and the least significant bit block (i.e., `0x00`) for the first level, referencing the accessed object pointer.

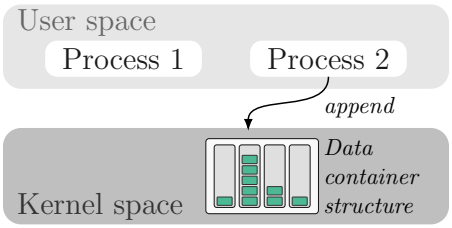
The *red-black tree* is a variant of a semi-balanced binary tree. Each tree node contains a value and up to two references to child nodes. All child nodes on the left branch are smaller, while all child nodes on the right branch are greater than the current tree node. Hence, nodes are ordered from lowest to highest (see Figure 9.3 with values 4 to 21). Each tree node is colored either red or black, with the root node being black. Insertion and deletion operations involve re-coloring tree nodes to ensure an approximation of the tree balance [12]. Consequently, due to this re-balancing, the red-black tree has a logarithmic worst-case execution time for lookups [12].

3. High-level Overview

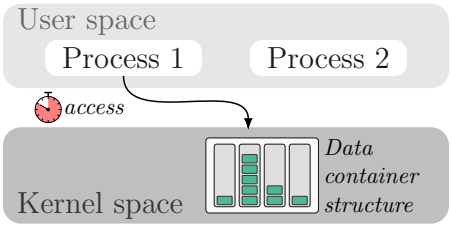
This section provides an overview of KernelSnitch, a timing attack on kernel container structures to leak sensitive information. At its core, KernelSnitch observes the occupancy level of data container structures in the kernel and deduces sensitive information from it through the following process, as shown in Figure 9.4: KernelSnitch measures the timing of accesses to data container structures shared between processes (see Figure 9.4a), by timing the syscall that accesses the kernel structure. Depending on the occupancy level, the timing of this access syscall varies. KernelSnitch then deduces the occupancy level from the obtained timing. The measured timing of



(a) Process 1 accesses (fast) a kernel *data container structure* via the syscall interface.



(b) Process 2 appends data to the *data container structure* via the syscall interface.



(c) Processes 1 re-accesses the *data container structure* which is now slower.

Figure 9.4: High-level of exploiting a kernel data container structure for a side channel.

accessing the data structure by process 1 indicates a fast operation, letting KernelSnitch infer a low occupancy (see Figure 9.4a). The occupancy level increases when additional data is appended to the structure (see Figure 9.4b). For example, appending data to a list increases its size, requiring additional iterations to access all elements. When KernelSnitch re-accesses the structure via a syscall, e.g., iterates through the entire list, it observes a slower syscall timing than on the initial access (see Figure 9.4c).

We perform three case study attacks by setting and observing the data structure occupancy level. First, we perform a *covert channel* attack by

controlling two processes, such as processes 1 and 2 from Figure 9.4. One process sets the occupancy level of a data structure to either low or high, while the other process measures its occupancy level. Second, using hash tables as data structures that use kernel heap addresses as part of the keys, KernelSnitch deduces hash collisions from setting and observing the occupancy level of hash buckets. This allows us to reconstruct kernel heap addresses from user space, i.e., *leaking kernel heap pointers*. Third, if an attacker controls process 1 while process 2 is a victim, KernelSnitch can deduce the activity of the victim, e.g., Firefox, from the occupancy level of data container structures. This activity deduction allows us to perform a *website fingerprinting* attack.

Several technical challenges have to be addressed to perform these three attacks. The following briefly describes these challenges, while subsequent sections discuss our solutions.

Occupancy Level Leakage of Data Structures. We measure the timing of syscalls that access kernel data container structures. While this timing depends on the occupancy level of these structures, we need to study this dependency in close detail, taking the numerous operations performed as part of a syscall into account. In Section 4, we address this and successfully determine the occupancy level for various data structures via timing measurements. We demonstrate the side channel in particular on fixed-size hash tables, dynamically resizable hash tables, radix trees, and red-black trees.

Amplification of the Information Leakage. Distinguishing lower and higher-level occupancy from user space is challenging. In some cases, the difference is only a few additional executed instructions, which we require to distinguish from user space. To overcome this, we demonstrate information leakage amplification methods in Section 5. These methods are classified as structure-agnostic and hardware-agnostic. We demonstrate that with these methods, we can reliably distinguish the occupancy level of data structures in idle and noisy systems. We also demonstrate that occupancy leakage and amplification are independent of the structures' allocation addresses and are consistent between reboots.

Attack Specifics. We perform three case studies of KernelSnitch attacks: covert channel, kernel heap pointer leak, and website fingerprinting, each of which has its sub-challenges. For instance, the covert channel relies on identifying a channel of the structure for communication. Consider a hash table, this involves identifying a shared bucket known to both processes.

In Section 6, we detail solutions for overcoming these sub-challenges and show how we leverage occupancy-level leakage to execute each attack. We demonstrate a covert channel with up to 580 $\frac{\text{kbit}}{\text{s}}$, a kernel heap pointer leak in less than 65 s, and website fingerprinting with an F_1 of score more than 89 %.

4. Root Cause Analysis

We analyze the security properties of data container structures, revealing a novel timing side channel, KernelSnitch, in the kernel that leaks sensitive information to unprivileged and isolated users. We showcase leakage from fixed-size hash tables (i.e., `hlist_head[]` and `plist_head[]`), a dynamically resizable hash table (i.e., `rhashtable`), a radix tree (i.e., `radix_tree_root`), and a red-black tree (i.e., `rb_root`). KernelSnitch deduces the occupancy level by measuring the timing of syscalls that access these structures from user space.

4.1. Leaking Occupancy Levels of Hash Tables

Hash tables in Linux consist of an array of key-indexed buckets, each containing a linked list of objects (see Section 2.1). When performing a hash table lookup, the kernel computes the bucket index by applying a hash function to the key. It then iterates through the linked list to find the corresponding object. As this iteration through the linked list takes time, it leaks the occupancy level of the iterated hash bucket through the required access timing. While this approach is generically applicable, we describe the occupancy leakage using the `futex_hash_table` (or `__futex_data.queues`) as an illustrative example (see Figure 9.5).

Futex Hash Table. Linux supports futexes [34] as fast user-space locking mechanisms, which mainly operates in user space and invokes syscalls for sleeping and waking otherwise. We exploit `sys_futex_wait/wake` as primitives to probe and alter the occupancy level of hash buckets within `futex_hash_table`. The wait operation (i.e., `futex` syscall with `FUTEX_WAIT_PRIVATE`), or `sys_futex_wait` in Figure 9.5, is a syscall, during which a local futex queue object (i.e., `futex_q`) is created and placed in the futex hash table (i.e., `futex_hash_table`). Storing in the hash table involves computing the hash using `futex_hash` with the current `mm_struct`'s kernel

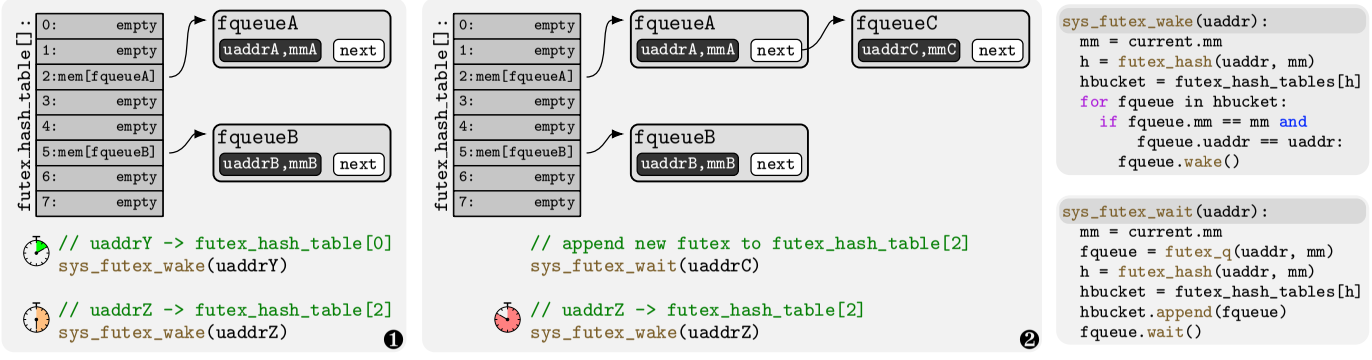


Figure 9.5: Representation of `futex_hash_table` as a fixed-size array of hash bucket, each containing a linked list of futex queues (i.e., `fqueue`). Initially ❶, buckets 2/5 store queues `fqueueA/B`. Accessing bucket 0 with `uaddrY` is fast, while accessing bucket 2 with `uaddrZ` is moderate. After adding `fqueueC` ❷ to bucket 2, accessing this bucket with `uaddrZ` becomes slow.

address and user-space address `uaddr`, which holds the user-space address of its futex structure.

We use the wait operation to increase the occupancy level of specific hash buckets in the futex hash table. In particular, we create a thread that subsequently executes `sys_futex_wait`, increasing the occupancy level. There are two ways to fill hash buckets. First, using the same `uaddr` within the same process (same `mm_struct`), KernelSnitch increases the occupancy of the same hash bucket. Second, using a different `uaddr` or a different process, KernelSnitch increases the occupancy of (most likely) different hash buckets. We use the wake operation with a mismatched identifier to probe the occupancy level of hash buckets. This syscall, simplified as `sys_futex_wake` in Figure 9.5, first computes the hash of the current `mm_struct` and the input `uaddr`. It then iterates through all futex queues linked to the corresponding hash bucket. Since we provide an `uaddr` that does not match any futex queue, the kernel iterates through the entire list, leaking the occupancy level of the hash bucket with the hash index `futex_hash(uaddr, mm)` through the wake syscall's execution time. To remove a futex queue from the `futex_hash_table`, we perform the wake operation on the sleeping thread.

The manipulation and observation of occupancy levels are detailed in Figure 9.5. The initial `futex_hash_table` ❶ contains `fqueueA/B` for hash buckets 2 and 5, respectively. We perform `sys_futex_wake` with a mismatched address (i.e., `uaddrY`), which, in combination with the current `mm_struct`, maps to bucket 0. Thus, KernelSnitch observes a fast time measurement corresponding to a low occupancy level. Similarly, repeating this process with the mismatched `uaddrZ` address associated with bucket 2 reveals a moderate occupancy level, indicating the presence of a single queue element. Furthermore, the execution of `sys_futex_wait` ❷ with the address `uaddrC` associated with bucket 2 allows KernelSnitch to increase the occupancy level of bucket 2. Consequently, performing `sys_futex_wake` with the mismatched address `uaddrZ`, corresponding to bucket 2, will have an even slower access time, leaking an increase in its occupancy level.

Vulnerable Hash Tables. We extend our analysis, showing that KernelSnitch is a generic attack, also leaking from other hash table implementations. These include `hlist_head[]` (i.e., `posix_timers_hashtable`) and `rhashtable` (i.e., `ipc_ids.key_ht`), which consist of fixed-size hash buckets and dynamically resizable hash tables, respectively.

The posix_timers_hashtable serves as a hash table to store POSIX interval timers, i.e., k_itimer. Linux has various timer-related syscalls, including sys_timer_create and sys_clock_gettime (see Listings 9.3 and 9.4 in Section 10.1), designed for creating timers and retrieving timer information. We demonstrate that these syscalls can be exploited to alter and probe the occupancy level of hash buckets within the timer hash table, which is similar to the futex hash table. In this implementation, the hash value is computed using timer_hash with the current signal_struct's kernel address and the unique timer identifier id. Based on this hash value, these syscalls access the corresponding hash bucket, adding a new timer or iterating through existing timers to retrieve information about the matching timer. To remove a timer from the hash table, a close syscall can be performed.

The ipc_ids.key_ht is a dynamically resizable hash table which stores kern_ipc_perm (short ipcp) objects that contain metadata used for user-space Inter-Process Communication (IPC). Specifically, ipcp objects are inherited by objects such as the msg_queue struct, which are intended for msg communication. The same applies to other IPC mechanisms, such as shm and sem. Linux provides syscalls for interacting with these parent objects, e.g., sys_msgcreate/msgget for the msg IPC mechanism (see Listings 9.5 and 9.6). KernelSnitch similarly exploits these syscalls as with the previous instances.

4.2. Leaking Occupancy Levels of Trees

This section demonstrates that trees are also vulnerable.

Radix Tree. This tree associates a pointer value with an integer key [11]. Each tree node contains multiple slots (typically 6 bit in size) pointing to NULL, child nodes, or stored objects. These slots are indexed by bit blocks of the key. During a key lookup, the kernel uses the most significant index to locate the corresponding slot in the root node, followed by subsequent indexes for lower-level nodes. The timing of a lookup depends on the tree's level, allowing for the leakage of its internal occupancy level through timing measurements.

One instance is the ipc_ids.ipcs_idr.root_rt radix tree, storing ipcp objects identified by unique key identifiers. Linux provides various syscalls to interact with this tree, such as sys_msgcreate/msgstat, used for appending ipcp objects to the tree or obtaining information from its parent

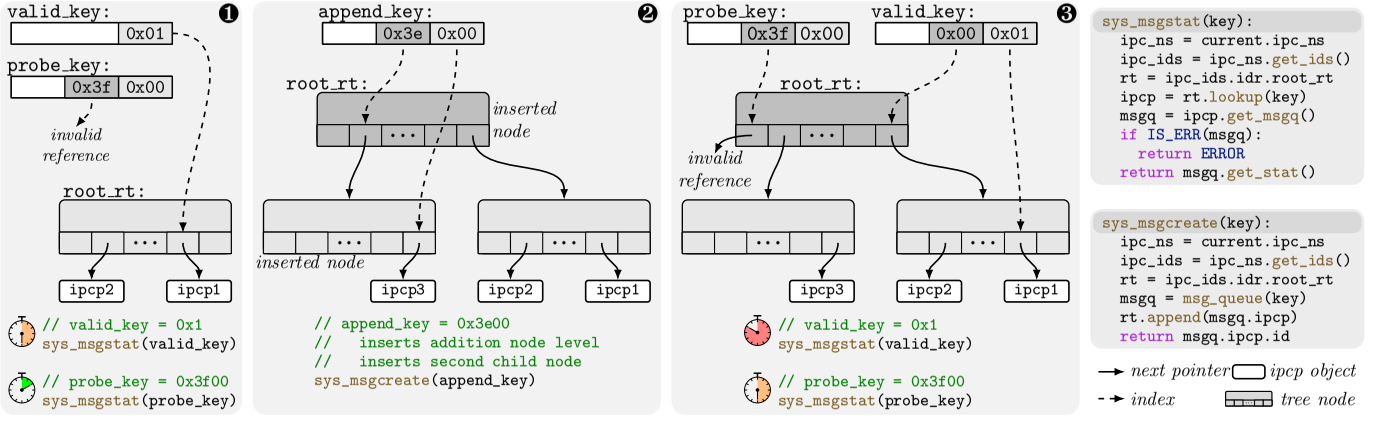


Figure 9.6: The representation of `ipc_ids.ipcs_idr.root_rt` initially consists of a one-level radix tree ❶. Probing `valid_key` with `sys_msgstat` results in a moderate access time due to a single node lookup, while probing `probe_key` results in a fast response due to no lookup. When adding `append_key` with `sys_msgcreate` ❷, the kernel inserts a second node level. Now, when probing with `probe_key` and `valid_key` ❸, an additional lookup is required, leading in increased access times.

object msg_queue. Similar to our approach with hash tables, we exploit these syscalls to alter and leak the occupancy level of the radix tree. The scenario depicted in Figure 9.6 illustrates how KernelSnitch exploits the ipc_ids.ipcs_idr.root_rt to leak its occupancy level. The radix tree initially consists of one level ❶, with the tree node having 64 slots. In the lookup of valid_key within sys_msgstat, the kernel uses the least significant 6 bits to access the 0x1 slot of the root node, retrieving the ipcp1 object. Since only one tree node is accessed during this lookup, the syscall runtime is moderate. By performing sys_msgstat(probe_key), the kernel does not access any tree node as the second 6-bit index 0x3f requires a second tree level which is not present. Thus, the access time is fast as no tree node is accessed. When adding append_key with sys_msgcreate ❷, the kernel inserts a second level and another first-level node, replacing the root node with the newly inserted second level. When re-accessing the radix tree with valid_key and probe_key using sys_msgstat ❸, their lookup and, consequently, execution time change. For valid_key, the kernel first fetches the 0x00 slot from the new root node, followed by fetching the 0x01 slot. Since a lookup for valid_key now accesses two tree nodes, the runtime is increased. For probe_key, the kernel initially fetches the 0x3f slot similarly the previous lookup. However, since this slot contains no valid next child node, the lookup yields an invalid reference. The lookup time, now accessing one node, increases compared to no node access.

Red-Black Tree. Linux uses red-black trees as key-sorted data structures [12], e.g., hrtimer_bases.clock_base.active which manages active high-resolution timers, sorting timer events by how close they are to their firing time.

Linux supports syscalls to interact with high-resolution timers, two of which are: sys_timerfd_create to create a timer and sys_timerfd_settime to activate it. Upon activation, the timer is enqueued to the hrtimer_bases.clock_base.active red-black tree. Considering the scenario in Figure 9.7, initially, the tree ❶ includes 4 timers sorted by time values ranging from 9 to 18. By calling sys_timerfd_create, the kernel creates two timers identified as fd0/1, which are not yet enqueued to the tree. Upon activating the timer identified with fd0 using sys_timerfd_settime ❷, the corresponding hrtimer is inserted at the tail of the tree since its value is 1000 larger than 18. This insertion requires two tree node accesses, resulting in a moderate enqueue time and indicating a moderate occupancy level. Activating fd1 using sys_timerfd_settime

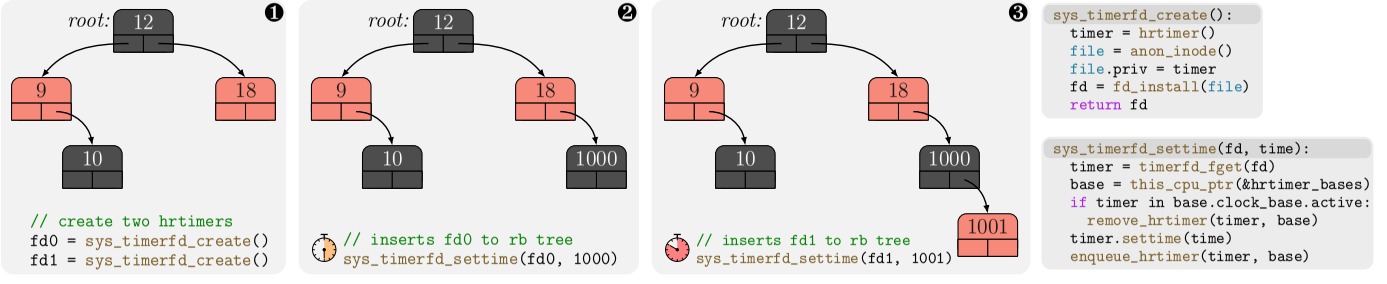


Figure 9.7: `hrtimer_bases.clock_base.active` includes four timers arranged by time values ❶, ranging from 9 to 18. When the new timer identified by `fd0` is enqueued to the tree ❷, the insertion process takes moderate time since it involves accessing two tree nodes. However, when the `fd1` timer is enqueued ❸, insertion time increases as it now requires accessing three nodes.

❸ involves three node accesses, resulting in higher execution time and suggesting a higher occupancy level compared to the previous enqueuing. Although enqueuing `fd1` triggers a tree rebalancing, this does not impact the structure's exploitability (see Section 5.2).

5. Amplification and Evaluation

KernelSnitch distinguishes lower and higher-level occupancy from user space. For instance, for the POSIX timer hash table, KernelSnitch needs to distinguish as few as 8 extra instructions executed based on time measurements from user space. We demonstrate in Section 5.1 how to amplify the information leakage in KernelSnitch attacks to a degree where these few extra instructions can be distinguished. We then evaluate the leakage without and with our amplification methods in Section 5.2, as well as with different noise floors.

5.1. Leakage Amplification

The amplification methods make the difference in occupancy of data container structures between lower and upper levels distinguishable from user space. We categorize these methods into *structure-agnostic* and *hardware-agnostic*.

Structure-Agnostic Amplification. Our structure-agnostic amplification mechanism extends the execution time of the instructions executed for each additional element. For example, the hash table `posix_timers_hashtable` iterates over the linked list of a hash bucket (see Listing 9.4 in Section 10.1). For each iteration, 8 additional instructions are executed (see Listings 9.1 and 9.2). They are two memory loads, three compares, and three jumps. Flushing targets of memory loads is a known technique [3, 24, 52], exploiting that cache hits are faster than misses [69]. As we cannot use Flush+Reload due to the lack of shared memory, we use cache eviction from user space. Since we do not assume to know the allocation addresses of the container structures nor any low-level information about the hardware caches, our eviction set consists of an array equal or larger to the Last Level Cache (LLC). Eviction is performed by accessing the entire array, essentially thrashing the LLC. This approach ensures that the targeted memory loads cause cache misses, thereby increasing the timing

difference between low and high occupancy. This amplification is agnostic to specific container structures and can be applied to any structures.

Hardware-Agnostic Amplification. We can also modify the state of the data container structure to increase the access time to a particular hash bucket by appending additional elements. For example, for the `futex_hash_table`, instead of appending one futex queue object to a specific hash bucket, we append multiple futex queues. We do this via the `sys_futex_wait` syscall, using the same user-space address `uaddr` within the same process, and, therefore, the same `mm_struct`. Since both the `uaddr` and `mm` are identical, their hash value `h0 = futex_hash(uaddr,mm)` also matches. Consequently, these futex queues are appended to the same hash bucket's linked list. Next, we invoke the probe syscall `sys_futex_wake` with a different user-space futex address `uaddr'` but the same `mm`. The syscall iterates through the futex queues within the hash bucket matching hash `h1 = futex_hash(uaddr',mm)`. If `h0` and `h1` match, the measured time becomes a function of the futex queues appended in the first stage. With more objects in the linked list of the hash bucket, the lookup time increases, significantly improving the detection of hash collisions. This generic approach works for all data containers that can contain linked data structures, i.e., other hash tables and the red-black tree.

For the radix tree, the amplification process works differently. Considering `ipc_ids.ipcs_idr.root_rt`, we aim to maximize the timing difference of the `sys_msgstat` syscall between scenario ❶ and ❸ in Figure 9.6. To achieve this, we append a specific `ipcp` object to the radix tree, introducing a new tree level. This operation is exemplified by the `sys_msgcreate(append_key)` action in scenario ❷, while we use `sys_msgstat` with `probe_key` to probe the internal state of the tree. However, to insert the new tree level, we need to occupy all slots in the first tree level beforehand. Hence, we initially append 64 `ipcp` objects. With the first level occupied, we alternate between insertion and removal of `append_key` to append or remove the second tree node, respectively. Hence, we obtain a notable timing difference between scenario ❶ and ❸, i.e., the insertion or removal of just one key.

5.2. Evaluation

We evaluate the KernelSnitch leakage with and without amplification and its noise resilience on each container structure. We then show the hardware

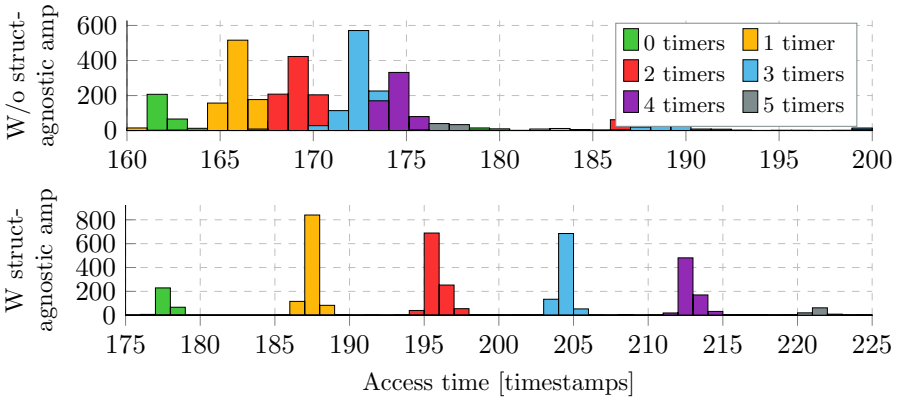


Figure 9.8: Information leakage of `posix_timers_hashtable` with and without amplification. We can see that the timings spread over a wider range with the amplification.

independence of our evaluation by running it on 4 different systems with the same source code. Crucially, the results do not depend on the allocation addresses of the structures and remain consistent between reboots.

We developed a helper kernel module to obtain the ground truth, e.g., the occupancy of a specific hash bucket. We then fill the data container with objects, modifying its occupancy level. We measure time using `rdtsc` before and after the syscall, storing the difference between these two timestamps. Using ground truth and KernelSnitch-deduced occupancies, we determine the False-Positive Rate (FPR) and False-Negative Rate (FNR). We run the evaluation with and without our amplifications for comparison (see Table 9.1). For consistent timing results, we filter outliers and focus on the ones not influenced by noise. Noise can only increase the timing and, hence, we average the lowest 8 values over 512 measurements. We evaluate on an Intel i7-1260P, with an Ubuntu 22.04.4 and kernel v6.5. We obtain similar results on 3 other processors with the same evaluation code (see Table 9.1); one even runs with kernel v5.15. We also evaluated on AArch64 (i.e., Raspberry Pi 4) with the same code except for using `clock_gettime` instead of `rdtsc`, showing similar results.

All operations performed, including syscalls and instructions, are available to unprivileged users and require no extra capabilities. Our side channel also does not rely on CPU frequency pining, as it is a privileged operation. In fact, as we show in our stress evaluation, the most dominant noise factor is the frequency fluctuation as we do not pin the frequency.

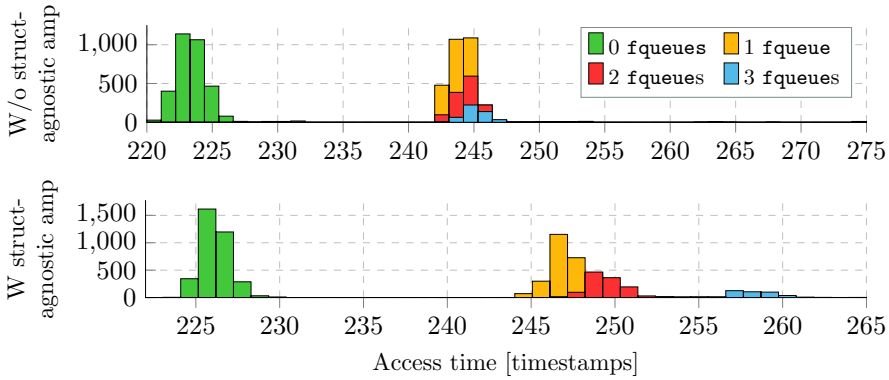


Figure 9.9: Information leakage of `futex_hash_table`.

POSIX Timer Hash Tables. We populate the `posix_timers_hash table` with 4096 timers using `sys_timer_create` to increase the occupancy of one randomly selected hash bucket out of 512. After appending each timer, we measure the time of the `sys_clock_gettime` syscall with a randomly selected, invalid `id`. We then compare the KernelSnitch-deduced occupancy of specific hash buckets with the ground truth. The measured timing is shown in Figure 9.8 without (distinguishing 1 and 0 timers, i.e., 1 to 0) and with (i.e., 3 to 0) hardware-agnostic amplification as well as without and with our structure-agnostic amplification (i.e., cache flushing). We compute the FPR and FNR using a threshold value between the medians of both histograms, e.g., 164 to distinguish 1 and 0 timers. We obtain 1.8 % (FPR) and 10.0 % (FNR) for distinguishing 1 and 0 timers. For distinguishing 3 and 0 timers, we obtain 0 % (FPR) and 9.4 % (FNR), showing the efficacy of the hardware-agnostic amplification. With structure-agnostic amplification, the FPR and FNR decrease to 0 %. The elimination of FNR and FPR yields an accuracy of 100 %.

Futex Hash Tables. We conduct a similar evaluation with `futex_hash_table`, using `sys_futex_wait` to append 8192 futex queue objects to the hash table, which consists of 4096 hash buckets on our default system[2]. The access timing was measured using `sys_futex_wake`. The evaluation results are illustrated in Figure 9.9. In contrast to the POSIX timer hash table results, the distinction between no elements within the hash buckets and one is more significant, while distinguishing multiple elements becomes less significant. One possible reason for this disparity

[2]Its size is computed as $256 \cdot$ `nr_cores`, where our system has 16 cores.

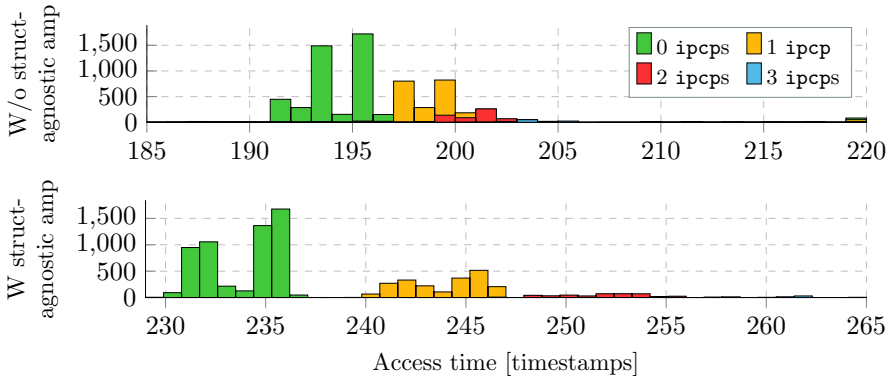


Figure 9.10: Information leakage of `ipc_ids.key_ht`.

lies in the differences in the lookup loop and the object's structure, i.e., `futex_hash_table` exhibits an early exit on lookup if no element is present. This characteristic renders the time difference between no elements and any element significant, shown in both with and without structure-agnostic amplification. When distinguishing between multiple elements, we suspect only one cache miss occurs in each iteration, where with `k_itimer` two misses occur. This results in a less significant timing difference for multiple elements.

IPC Hash Tables. For the `ipc_ids.key_ht`, we exploit `sys_timer_create` to populate data objects into pseudo-random hash buckets. We then exploit `sys_clock_gettime` to probe pseudo-randomly selected buckets. In total, we populate 4096 objects, probing the hash buckets after each insertion, and obtained the ground truth using our helper module. Figure 9.10 illustrates the results of this evaluation, with Table 9.1 summarizing the FPR and FNR, along with the improvements achieved in both leakage amplifications. Both figures show that this hash table primarily comprises hash buckets with low occupancy levels. This characteristic stems from the hash table's dynamically resizable property. If the occupancy level of multiple buckets becomes too high, the hash table automatically resizes its bucket array and restructures objects within the buckets. Importantly, as demonstrated in Section 6, although this dynamic resizing property may complicate exploitation, KernelSnitch still leaks information from these hash tables.

Radix Tree. For `ipc_ids.ipcs_idr.root_rt`, our evaluation was as follows: We begin by inserting `ipcp` objects to the radix tree using `sys_-`

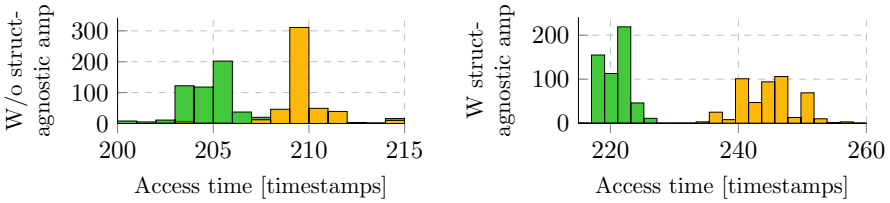


Figure 9.11: `ipc_ids.ipcs_idr.root_rt` information leakage.

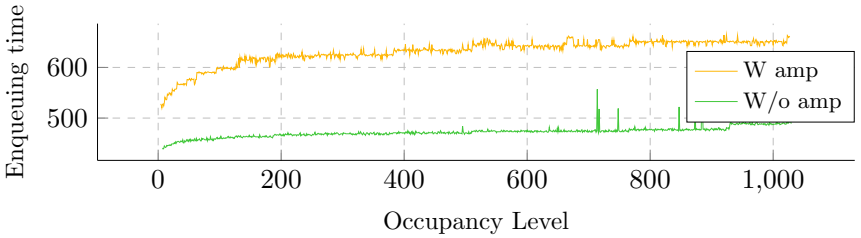


Figure 9.12: Leakage of `hrtimer_bases.clock_base.active`.

`msgcreate` until the entire first tree level is filled with valid slots (i.e., 64 slots). We then alternatively insert and remove an `ipcp`. This prompts the kernel to either append a new tree level, as depicted in scenario ❷ of Figure 9.6, or remove the just-appended level. After every insertion and deletion, we perform a radix tree lookup with the probe syscall `sys_msgstat` using an invalid key, where we do 1024 in total. Depending on whether the radix tree consists of one level ❶ or two levels ❸, the lookup timing varies. Figure 9.11 illustrates the histograms of access times with no structure-agnostic amplification, depending on whether the radix tree has one or two levels, as well as with amplification, increasing the distinguishable between these histograms. While KernelSnitch without the amplification resulted in FPR and FNR of 1.7 % and 3.9 %, the amplification eliminated all of them.

Red-Black Tree. Contrary to the prior evaluations, we conduct a slightly different one for the `hrtimer_bases.clock_base.active` red-black tree. In this evaluation, we aim to demonstrate how the enqueuing time depends on the occupancy level. As the tree maintains self-balancing, we anticipate that the enqueuing time follows a logarithmic function depending on the occupancy level. We enqueue 1024 high-resolution timers and simultaneously measure the enqueuing time and obtain the ground truth using our helper kernel module after each enqueue operation. Figure 9.12

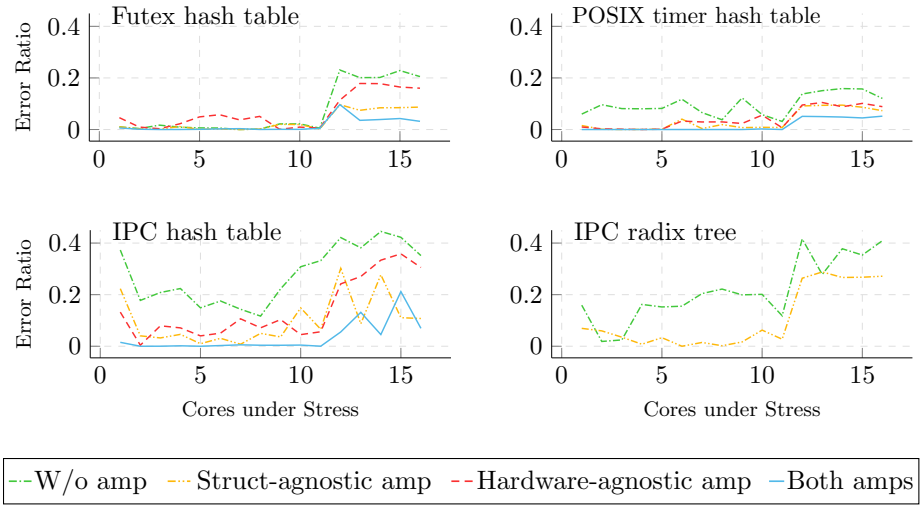


Figure 9.13: Noise evaluation results without and with amplification methods as a function of the stress cores (i.e., 1 to 16).

depicts the enqueuing time relative to the tree's occupancy level with and without structure-agnostic amplification. Both functions exhibit a logarithmic dependency on the actual occupancy level. Our amplification increases the enqueuing time by over 347 %. The occupancy level does not start at 0 timers, as the system always has default timers enqueued, e.g., `tick_sched_timer`.

External Noise. We introduce noise either through *stress evaluation* or *directly into data structures*. We show that the most dominant noise factor is the CPU frequency fluctuation and the noise resilience of our KernelSnitch side channel.

For the *stress evaluation*, we vary the number of workload threads of `stress-ng` [41, 43, 53], ranging from 1 to 16, i.e., the number of logical cores for the evaluated Intel i7-1260P. These workloads stress the CPU cores on which the workload is running. We separately evaluate structure- (i.e., distinguishing between 3 and 0 elements) and hardware-agnostic amplification (i.e., flushing CPU caches), as well as their combination. Figure 9.13 illustrates the error ratio observed in the stress evaluation, relative to the number of CPU cores experiencing stress. The error ratio represents the proportion of incorrectly deduced elements compared to the total evaluated (i.e., FPR+FNR). We note a rise in the error ratio when stress is introduced to an equal or greater number of physical cores

(i.e., 12 cores). However, with both amplifications applied, we observe a negligible error ratio below 1.5 % if stress is applied to fewer than 12 cores. This yields an accuracy of more than 98.5 % until ¾ of the full load. Concurrent measurement of the CPU frequency shows that frequency fluctuation caused by adaptive power management are the dominant noise factor for the stress evaluation. These fluctuations result in varying syscall execution times, perceived as noise for the attacker.

We also stress the Intel Xeon Gold 6530 (i.e., 64 cores) with workload threads ranging from 16 to 63. Specifically, we evaluate the POSIX timer hash table, observing no error ratio across all tests with both amplifications applied. By simultaneously measuring the CPU frequency, we observed it to be almost constant throughout the evaluation, as this is a desktop CPU with powerful cooling. This underscores the finding that the dominant noise factor is frequency fluctuation, most prevalent in laptop CPUs, e.g., Intel i7-1260P.

For introducing noise *directly into the data structure*, we found that Phoronix's Apache benchmark with 1 000 concurrent requests introduces the most noise into the futex hash table compared to other benchmarks. It introduces noise in the form of 55 000 (up to 100 000) hash bucket changes per second. We applied both amplifications and evaluated on an Intel i7-1260P, resulting in an error ratio less than 1 %. The simultaneous measurement of the CPU frequency shows that noise due to frequency fluctuations is predominant.

6. Attack Case Studies

In this section, we demonstrate the practicality of KernelSnitch attacks in three side-channel case studies: a covert channel, a kernel heap pointer leak, and a website fingerprinting attack. We run the experiments on an Intel i7-1260P with Ubuntu 22.04.4 and a Linux kernel v6.5. Our attacks have an automated calibration phase directly at the start of the exploit to determine the threshold between a low and high occupancy levels. These thresholds remain consistent for the specific data container structures and do not depend on the allocation addresses of the structures as shown in Section 5.2.

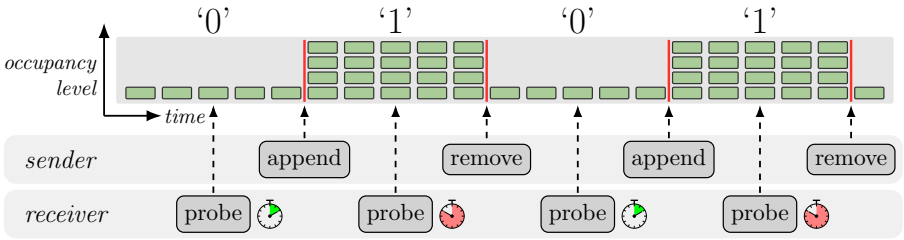


Figure 9.14: KernelSnitch covert channel's overview, where the sender alters and the receiver probes the occupancy level of a data structure in fixed time slices.

6.1. Covert Channel

We demonstrate that all container structures analyzed can be used to establish a covert channel. Our covert channel uses time slicing on the occupancy side channel to transmit data.

Threat Model. We assume that the sender and receiver run co-located on the same system. The sender has access to sensitive data but is strictly isolated and has no network access, e.g., within a sandbox. The receiver has no access to sensitive data but network-access permission, e.g., to a remote server to exfiltrate data. The sender and receiver have no shared memory or other resources shared besides the kernel itself.

Overview. We transmit data as a binary signal, e.g., in Figure 9.14 a bit sequence of '0101'. We transmit a '1' by increasing the occupancy level by appending one or more objects and a '0' by reducing the occupancy level by removing one or more objects. This results in a higher or lower probe syscall time, which is what we build our channel on. We synchronize our covert channel with a shared timer, e.g., `rdtsc` on x86_64. The transmission starts at a coarse-grained predetermined time offset (e.g., the last 38 bits wrap around at a full minute), while bits are transmitted in short predetermined time slices. For each time slice, KernelSnitch adjusts the occupancy based on the data being transmitted. The receiver process continuously probes the occupancy throughout the time slice, using the minimum probe value as a result, minimizing noise.

Design for Hash Tables. We initially identify a hash bucket for communication. The sender process uses hardware-agnostic amplification and populates one hash bucket with many elements. Subsequently, the receiver process iterates through the hash table, probing each bucket to determine

if it contains a substantial number of elements. When it identifies the bucket, the receiver knows the bucket that will serve as the shared channel. With this stage complete, the transfer starts.

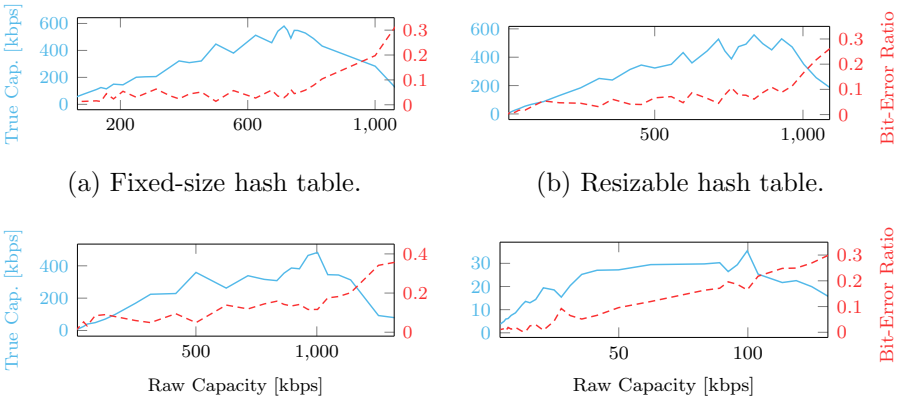
We build two covert channels. The first covert channel is based on fixed-size hash tables, leveraging `futex_hash_table`. A similar principle can be applied to other hash tables with a fixed size, e.g., `posix_timers_hashtable`. The second covert channel is based on dynamically-sized hash tables, leveraging `ipc_ids.key_ht`. For the fixed-size futex hash table, the sender initially populates the hash bucket with 64 futex queues. Subsequently, the receiver finds this hash bucket, as described above. For the transmission, a single appended futex queue is enough to transmit a '1' bit, while the absence of this futex queue transmits a '0' bit. For the dynamically-sized hash table, `ipc_ids.key_ht`, we initially populate it with 16 keys to find the shared bucket and then transmit data with occupancy differences of one key.

Design for a Radix Tree. The sender transmits '1' with the tree occupancy level 2 and a '0' with tree level 1. The tree level is manipulated by appending a specific key requiring an additional level (see `append_key` in Figure 9.6) or removing this key again. The receiver probes the radix tree occupancy, where higher probe times indicate a '1' and lower times a '0'.

Design for a Red-Black Tree. Appending 16 timers is sufficient to create a distinct probe timing difference (see Section 5). Hence, the sender appends 16 timers at the tree's tail by setting a very high initial value, causing the red-black tree to insert multiple levels. The sender and receiver encode the data in timings: a lower timing corresponds to fewer levels, i.e., a '0' bit; a higher timing indicates a '1' bit.

Evaluation. We evaluate all four data container structures (i.e., fixed-size and resizable hash tables, radix tree, and red-black tree). For each structure, we evaluate different time slice lengths and record the channel's raw capacity – the maximum potential data rate – and bit-error ratio. Shorter time slices yield a higher transmission rate but reduce the receiver's ability to probe the occupancy level reliably, e.g., the measurement may be more noisy. Consequently, while the raw capacity increases with shorter time slice lengths, the true channel capacity might decrease due to a higher bit-error ratio. To represent our channels' effectiveness, we compute the true capacity[3] based on the raw capacity and the bit-error ratio.

[3]We use Shannon's theorem: $T = C \cdot (1 + ((1-p) \cdot \mathtt{ld}(1-p) + p \cdot \mathtt{ld}(p)))$.



(a) Fixed-size hash table.

(b) Resizable hash table.

(c) Radix tree.

(d) Red-black tree.

Figure 9.15: KernelSnitch covert channel's raw capacity, bit-error ratio, and true capacity, ranging between 35 $\frac{\text{kbit}}{\text{s}}$ to 580 $\frac{\text{kbit}}{\text{s}}$.

Figure 9.15 shows the true capacity as a function of the raw capacity and bit-error ratio for all four structures. For the fixed-size `futex_hash_table` (see Figure 9.15a), the bit-error ratio is below 7 % until the raw capacity reaches 781 $\frac{\text{kbit}}{\text{s}}$. Beyond this point, the slice length becomes so short that the receiver can only execute a maximum of 3 probing syscalls to deduce the occupancy level. We observe that with at most 3 probing syscall, there is a steady increase in the bit-error ratio, reducing the actual capacity. For the futex table, the optimal true capacity of 580 $\frac{\text{kbit}}{\text{s}}$ is achieved at a raw capacity of 714 $\frac{\text{kbit}}{\text{s}}$ and a bit-error ratio of 2.8 %. We observe a similar behavior for both the dynamically-sized hash table (see Figure 9.15b) and the radix tree (see Figure 9.15c). The point of a steady increase in the bit-error ratio occurs at a raw capacity of 963 $\frac{\text{kbit}}{\text{s}}$ and 1 003 $\frac{\text{kbit}}{\text{s}}$, respectively. Their optimal true capacity is reached with 528 $\frac{\text{kbit}}{\text{s}}$ and 483 $\frac{\text{kbit}}{\text{s}}$. As both data structures are used for IPC communication, Linux isolates them within the IPC namespace. Hence, this channel is restricted to the sender and receiver sharing the same IPC namespace, prevented by sandboxes, e.g., Docker or browsers. For the red-black tree (see Figure 9.15d), we observe that the optimal true capacity is 35 $\frac{\text{kbit}}{\text{s}}$, achieved at a raw capacity of 100 $\frac{\text{kbit}}{\text{s}}$ with a 16.5 % bit-error ratio. This capacity is the lowest of the four structures, primarily due to the extended time required for appending and probing operations.

6.2. Kernel Heap Pointer Leak

Linux uses kernel heap addresses of objects such as `mm_struct` in the indices for hash table lookups. We demonstrate that we can leak these kernel heap addresses used to index hash table lookups using KernelSnitch to observe hash collisions from user space. We then demonstrate that by performing a cross-cache reuse [43, 66, 68], we can place other objects, such as the security-critical `msg_msg` [14, 30, 72], at this leaked address, thereby obtaining the location of other objects.

Threat Model. We assume an attacker has no privilege to access information about kernel addresses, and the attacker has an exploit that only works if a targeted kernel heap pointer is known, e.g., for multiple exploits [5, 14, 25, 30, 72].

Design. Our kernel heap pointer leak consists of two steps: First, we *detect hash collisions* of hash table entries with the same kernel address but different user identifiers. Second, we *enumerate all possible kernel addresses* to match the detected collisions for the user identifiers used, resulting in the kernel address used for indexing being leaked. Using the futex hash table as an example, we aim to leak the `mm_struct` address, which is used with the user identifier `uaddr` for indexing.

To *detect hash collisions*, we exploit the KernelSnitch side channel as follows: Initially, we append one futex queue to a hash bucket, such as `fqueueA` with `uaddrA` and `mmA` to bucket 2 as `futex_hash(uaddrA,mmA)` = 2. This state is depicted with ❶ of Figure 9.5, where `fqueueB` represents a queue in another bucket. We then apply structure-agnostic amplification, appending multiple queues to the hash bucket 2 using the same `uaddrA` and `mmA`. With hardware-agnostic amplification, we observe the occupancy level of the hash bucket using the same `mm_struct` (as the same user process) but with different and invalid user identifiers, i.e., `uaddr`s. A low occupancy level, such as for the user identifier `uaddrY`, indicates a different hash bucket. Conversely, a higher occupancy level, e.g., `uaddrZ`, means that the values `futex_hash(uaddrA,mmA)` and `futex_hash(uaddrZ,mmA)` match. We denote this as a hash collision of the user identifiers `uaddrA` and `uaddrZ` using the same `mm_struct`. We repeat this process until a sufficient number of hash collisions are found.

We now have a list of known user identifiers (i.e., `uaddr`s) that, combined with one unknown kernel heap address (i.e., `mm_struct`), results in the same hash value. With the hash function known (i.e., `jhash2` for the futex

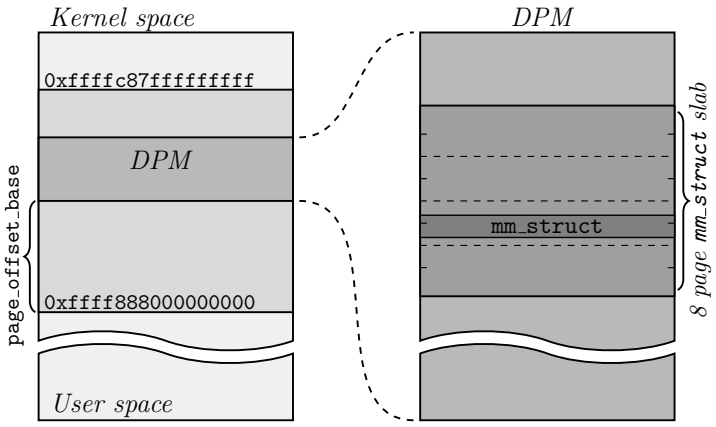


Figure 9.16: The kernel memory layout on x86_64 illustrates the kernel heap is accessible directly via the DPM, showcasing how the heap-allocated `mm_struct` object is located.

hash table), we *enumerate all possible kernel addresses* together with the known user identifiers in an offline phase to determine hash collisions. As described in the next paragraph, the search space for all possible heap addresses of a specific `mm_struct` can be reduced to $\approx 2^{35.5}$. If we find the address that, combined with all user identifiers, results in the same hash, we leak the `mm_struct` heap address. In the simplified example of Figure 9.5, we leak `mmA` as it results in the same hash value when combined with the identifiers `uaddrA` and `uaddrZ`.

Since the kernel heap is directly accessed via the Direct Physical Mapping (DPM) [45] (see Figure 9.16), the search space is the DPM offset by the randomized `page_offset_base`. The DPM serves as a virtual memory mapping of, typically, the physical memory range and spans over a significant part of the kernel address space. For instance, on x86_64, it ranges between `0xffff888000000000` and `0xffffc87fffffffff`, representing a search space of 2^{46} (when considering 8 B kernel heap alignment, it results in an entropy of 43 bit). To reduce the search space, we consider the alignment constraints of `mm_struct`, enforced by the page and slab allocator[4]. The page allocator guarantees the outer alignment, ensuring that the memory chunk (also called slab) from which `mm_struct` objects are allocated is aligned to 8 pages. The slab allocator sits on top of the page allocator and ensures that objects within these slabs are aligned to

[4]These details can be obtained from `/sys/kernel/slab/mm_struct` and remain consistent across the same kernel binary.

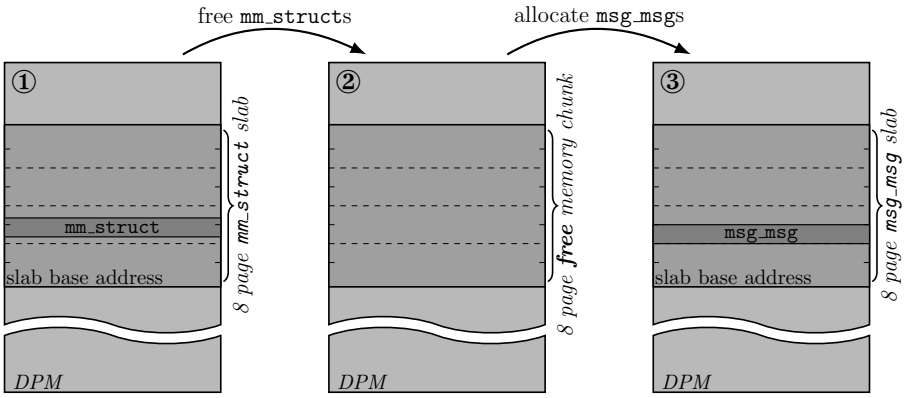


Figure 9.17: Cross-cache reuse which frees the leaked `mm_struct` (and all of its slab) and reallocate its memory chunk as `msg_msg`s, thereby obtaining the location of these `msg_msg`s.

object size (and usually also to the cache line size). Using these insights allows us to substantially reduce the search space for possible `mm_struct` addresses as follows: On our experimental system using Linux v6.5 x86_64 with the default, generic configuration, the `mm_struct` has a size of 1 360 B. Considering the alignment of the cache lines (rounded up to 1 408 B), 23 locations are possible within the 8 slab page. Hence, the search space is $2^{46-12-3} \cdot 23 \approx 2^{35.5}$, with 12 bits representing the page size and 3 bits representing the `mm_struct` slab size. Given a complexity of $\approx 2^{35.5}$, we iteratively examine all possible addresses and try to match them with previously leaked hash collisions produced with different user identifiers. Subsequently, we reconstruct the key corresponding to these hash collisions. As the kernel heap address is one of the key's inputs, we successfully obtain this address, consequently leaking heap pointers.

Cross-Cache Reuse. We perform a cross-cache reuse [43, 66, 68], which frees the leaked `mm_struct` object (including all objects of its slab) and reallocates the freed (and leaked) memory chunk for other objects. This allows us to leak the location of objects other than those directly leaked via KernelSnitch. Below, we demonstrate that by using this approach, we can leak the address of the security-critical `msg_msg`, used in several kernel exploits [14, 30, 72]. While `msg_msg` is an example, we can also leak the address of other objects.

Figure 9.17 shows the high-level overview, where Figure 9.20 provides more details. The state ① represents the leaked `mm_struct` address within

its slab. From this `mm_struct` address, we derive the base address of its slab, which is done by applying a bitmask of the 8 page memory chunk (i.e., `((1<<15)-1)`) to the address. Next, we deallocate all `mm_struct`s within this slab ②, causing the kernel to recycle the leaked 8 page free memory chunk. We then allocate multiple objects ③ of the targeted type (i.e., `msg_msg` with size 4 048) to reclaim the 8 page memory chunk. Using the alignment information of the allocator cache of `msg_msg` with size 4 048 (i.e., `kmalloc-cg-4096`), we deduce all possible object locations within this chunk. This results in 8 locations of $\texttt{n} \cdot 4\,096 + \texttt{slab\_base}$ where `n` is between 0 and 7.

We reclaim the leaked slab previously used for `mm_struct`s as `msg_msg`s with size 4 048, as both objects use the same size per-CPU page free list order of 3 (i.e., 2^3 page memory chunk). These per-CPU page free lists act as a first-level allocator cache of the page allocator. If we want to reclaim the leaked 8 page memory chunk as a different page size chunk, such as `kmalloc-cg-512` (which uses the per-CPU page free list of order 2), we must first drain the page free list of order 2. Prior work [66] has presented appropriate techniques to reliably perform this cross-page free-list reuse.

Evaluation. We implement our KernelSnitch kernel heap leak in an architecture-agnostic manner. Architecture-specific information is required to reconstruct the `mm_struct` address from hash collisions due to variations in the DPM across different architectures. We implement KernelSnitch to leak the kernel heap address for x86_64, AArch64, and RISC-V architectures. In our experiments, we successfully perform this attack natively with our x86_64 experimental setup as well as in QEMU for AArch64 and RISC-V architectures. For the native experiment, we repeat the hash-collision leaking attack 10 times, with a leak time between 1.7 s to 2.1 s. Using these collisions, we recover the correct `mm_struct` address in 2 s to 61.5 s (iterating through 1.8 % to 28.5 % of the possible kernel addresses) on a 24-core AMD EPYC 7443 processor. Consequently, KernelSnitch requires between 3.7 s to 63.6 s for a kernel heap leak. In addition to leaking the `mm_struct`, we implement the cross-cache reuse described above for our native x86_64 system. We successfully reclaimed the leaked `mm_struct` slab for the `kmalloc-cg-4096` slab cache, leaking the address of the 8 `msg_msg` objects it contains. Similar to the above, we performed 10 successful cross-cache reuses on our native system. They took between 0.847 s to 0.901 s, resulting in a total time of under 65 s for leaking a `msg_msg`.

We also exploit the POSIX timer hash table, reducing potential `k_itimer` addresses from 2^{38} to 2^9. While it only led to a partial kernel heap address

leak, it advances understanding system vulnerabilities. The POSIX timer's hash function lacks uniform output distribution, preventing determination of the linearly mapped input-to-output part (i.e., 9 bits). In contrast, the futex hash table, using `jhash`, ensures uniform distribution, enabling of the leakage the entire kernel heap address.

6.3. Website Fingerprinting

This section presents a website fingerprinting attack, showing its capability to determine when a user accesses a website from the Ahrefs top 100 [1] with an F_1 score of 89.3 %.

Threat Model. We assume the attacker executes code on the same machine as the victim but is isolated by the web browser's sandbox. Since we assume a sandboxing isolation (e.g., for the mount and network namespace), approaches such as calling `netcat` to leak browser activity cannot be used.

Design. Web browsers like Firefox rely on user-space locks, i.e., futexes, to handle transmitted and received data. During website access, the browser acquires and releases these locks and interacts with the futex hash table. This behavior creates a unique occupancy level fingerprint in the futex hash table. We leverage KernelSnitch to leak the futex hash table's occupancy level in this side-channel attack. We use a Convolutional Neural Network (CNN) to classify these occupancy level fingerprints of the Ahrefs top 100 websites [1].

We use the occupancy-level side channel of the futex hash table to obtain website traces in two stages: First, we find for each hash bucket a unique user-space address, allowing us to leak the occupancy level of each bucket with `sys_futex_wake`, as represented in Figure 9.5. To achieve this, we leverage the hardware-agnostic leakage amplification by appending a significant number of futex queues to a hash bucket. Using structure-agnostic leakage amplification, we probe all buckets with incremental user-space addresses to identify a bucket with a significant number of queues appended. Upon discovering a significant hash bucket, we validate that this user-space address does not cause a collision with a previously found user-space address. This appending and probing routine is repeated for all buckets. As a result of this initial stage, we have a set of user-space addresses indexing all hash buckets, which we refer to as our probe set. Second, we use our probe set to determine the access times of each hash

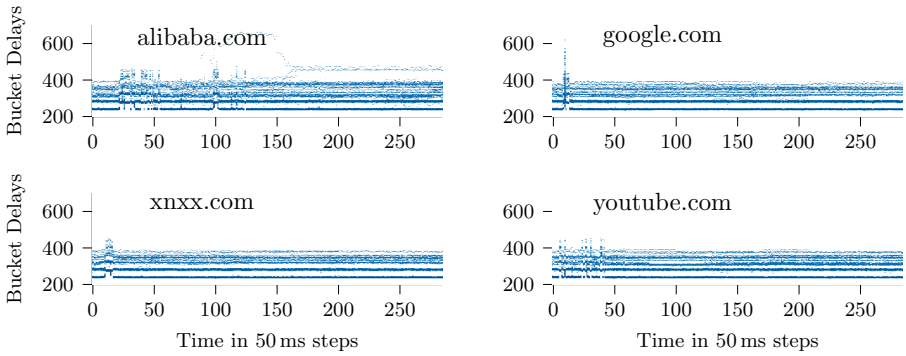


Figure 9.18: Traces of 4 famous websites, showing the delay of all bucket measurements on the y axis while website loading.

bucket with 20 samples per second. For probing, we first iterate over all hash buckets and then repeat this process for the entire sample timeframe (i.e., 50 ms). This way, we have structure-agnostic amplification without explicitly flushing the CPU caches. Subsequently, we take the minimum probe value for each hash bucket within a timeframe. This is repeated during loading a website (i.e., 15 s), creating a two-dimensional trace, consisting of the access times of all buckets at 300 timestamps. Finally, we create a histogram, rounding all bucket delays to the nearest integer and summing all up. Figure 9.18 shows four website traces. The x-axis is the time axis, and the y-axis shows the bucket delays, with the color darkness representing the number of buckets with each delay.

Our attack consists of an online and offline phase for data collection and evaluation of website traces. In the online phase, a user-space process runs KernelSnitch on the system within a sandbox, probing all hash buckets of the futex hash table, resulting in a website accessing trace. The offline phase consists of analyzing and classifying the collected traces. To classify the traces, we use a CNN with nine convolutional layers in three different sizes. The two-dimensional histograms, as shown in Figure 9.18, are the inputs to the CNN.

Evaluation. We record 100 traces for each of the 100 websites on Firefox (Chrome results in similar traces). We split the traces into 80 % training and 20 % test sets. Of the training set, we use 10 % for the validation used set while training. We perform a 5-fold cross validation by training the CNN with five randomly selected sets. Over the five validations, we

achieve an average F_1 score of 89.3 %. Figure 9.21 shows the confusion matrix with an F_1 score of 89.5 %.

7. Related Work

Numerous physical properties carry an information signal that is often tied to the specific implementation of a system or algorithm. These include power, radiation, temperature, sound, light emission, and time, with time being the most commonly used. More recent software side channels also extract information through timing differences induced by other physical properties [47, 65]. Lampson already reported in 1973 that timing differences could be exploited to transmit or extract information covertly [40]. In 1996, Kocher [37] presented the first timing side-channel attack on a cryptographic algorithm. Following the numerous timing-based attacks on cryptographic algorithms [7, 61], Osvik et al. [49] generalized the approaches into two generic techniques, Evict+Time and Prime+Probe. A decade later, Yarom et al. [69] presented Flush+Reload, monitoring cache-lines' state by removing them from the cache and timing a reload to the corresponding memory location. The timing depends on if the victim has accessed the cache line in between. Flush+Reload has virtually no false results and is frequently used by other attacks [38, 42].

Software-observable timing differences can be induced by caches and any behavioral difference on the software or instruction level, e.g., software caches [16, 22, 43, 64], page-fault interrupts [60], other interrupts [15], compression algorithms [6, 21, 33, 54, 56], differences in instruction or memory lookup sequences [63, 69], and many others. While the concept of constant-time [37] implementations has found wide adoption for cryptographic algorithms, the situation is much more difficult for general-purpose code [55]. Gao et al. [17] presented information leakage of files not fully namespaced allowing for covert channels. Gruss et al. [22] found that the operating system page cache can be exploited similarly to hardware caches. Their insights show that microarchitectural buffers and caches similarly exist in operating systems, again with caches, indicating an architectural interface is also applicable to the operating system. More recently, Patel et al. [50] presented a novel performance-degrading attack that exploits intra-kernel contention of locked kernel resources. While not a side channel, their result indicates that more architectural elements in the kernel may be exploited. Jiang et al. [31] showed that file system

sync operations affect each other's timing and can be used to build a covert channel, achieving transmissions of up to 20 $\frac{\text{kbit}}{\text{s}}$ with an error rate 0.4 %. Chen et al. [10] showed that a similar timing influence also exists with write buffers for shared files. By filling or not filling the write buffer, they can covertly transmit up to 10 $\frac{\text{kbit}}{\text{s}}$ with a 0.004 % error rate. Lee et al. [41] and Maar et al. [43] discovered timing side channels in the Linux slab allocator, inferring whether a new slab is created. This leakage increases the success rate for heap spraying [41] or cross-cache attacks [43]. Shen et al. [57] presented a covert channel based on mutual exclusion primitives [71], achieving transmission rates of up to 13.1 kbit/s with a 0.65 % error rate.

The concept of software-induced timing side channels is also related to the research problem of algorithmic complexity attacks [13]. Algorithmic complexity attacks try to provide systems with input that triggers the algorithmic worst case, e.g., a bucket with a long linked list instead of a flat hash table. The goal in these attacks is often denial of service [13, 58], deteriorating the runtime. Several works, therefore, discuss mitigations against denial-of-service algorithmic complexity attacks [4, 35]. Petsios et al. [51] presented a fuzzer to find algorithmic worst cases, including compression algorithms, for algorithmic complexity attacks. Schwarzl et al. [56] similarly built a fuzzer to find algorithmic worst cases in compression algorithms to build a new side-channel attack, showing the close relation between these two strands of research. Sun et al. [59] showed that algorithmic complexity attacks can also be used to build covert channels. Cai et al. [8] exploited algorithmic complexity attacks when exploiting race conditions on Unix file systems.

8. Mitigations

Our work extends prior research on software-induced timing side channels, showing that varying access timings of any shared kernel resource can also introduce a timing side channel, which, given the kernel interfaces, can potentially be exploitable from user space. Thus, the operating system level introduces exploitable leakage even when eliminating all hardware side channels and following all best practices for user software. Mitigating KernelSnitch has to be evaluated with performance and user experience, similar to other side channels [26, 48]. The key factors that enable KernelSnitch are the sharing of data container structures, the combination

of privileged and unprivileged information in the same shared element, the runtime variance depending on a secret state, and the possibility to measure the runtime. Eliminating any of these can fully or partially mitigate KernelSnitch.

Measuring Time. Prior work studied the removal of precise timing measurements as a defense against side-channel attacks [27, 39]. However, other works also showed how attackers can resort to alternative timing methods or mount attacks entirely without timing [52, 64, 67, 70]. In our threat model, multiple timing sources are available for legitimate reasons, and removing them would be a highly disruptive change for software developers and require hardware changes.

Combining Privileged and Unprivileged Information. The aggregation of privileged and unprivileged information in shared elements has already been identified to introduce security issues on the hardware level [23, 62]. For KernelSnitch, we also exploit that unprivileged (i.e., user identifiers) are combined with privileged information (i.e., kernel addresses). However, eliminating this only prevents the kernel heap leak.

Runtime Variance. A constant-time approach [7, 37] is not directly possible against KernelSnitch as the container structures are, in principle, only bounded by the available memory. However, KernelSnitch could be mitigated by using a watermark constant-time approach: Instead of always resorting to a (hypothetical) worst-case execution time, it may be a viable approach to maintain a watermark level of the maximum number of elements to be iterated through. Syscalls iterating through structures will consequently wait until the watermark execution time is reached, eliminating the secret-dependent runtime variance. Further security can be gained by increasing the watermark level in a coarse step size.

Sharing of Container Structures. Another mitigation is to eliminate the sharing of the kernel data container structures, e.g., isolating kernel data container structures within namespaces. However, this involves significant reworking and notable performance and memory overheads for the kernel. Future work needs to investigate the practicality of such isolation. For instance, the red-black tree that organizes global timers by their firing order poses a challenge.

9. Conclusion

In this paper, we presented KernelSnitch, a novel generic side-channel attack targeting kernel data container structures. We demonstrated and evaluated leakage amplification to make this side channel exploitable from user space. We performed three case study side-channel attacks: covert channel, kernel heap pointer leak, and website fingerprinting. Finally, we discussed potential mitigations and highlighted challenges.

Acknowledgment

We thank Mathias Oberhuber and the anonymous reviewers for their valuable feedback. This research is supported by the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087), the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided from Red Hat and Google. Any opinions, findings, 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] Ahrefs. Top Websites. 2024. URL: `https://ahrefs.com/top` (pp. 312, 340).

[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 (p. 310).

[3] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying Side Channels Through Performance Degradation. In: ACSAC. 2016 (p. 325).

[4] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. Surgeprotector: Mitigating Temporal Algorithmic Complexity Attacks Using Adversarial Scheduling. In: ACM SIGCOMM. 2022 (p. 343).

[5] Awarau and pql. CVE-2022-29582 An io_uring vulnerability. 2022. URL: https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/ (p. 336).

[6] Tal Be'ery and Amichai Shulman. A Perfect CRIME? Only TIME Will Tell. In: Black Hat Europe. 2013 (pp. 310, 342).

[7] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (pp. 342, 344).

[8] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting UNIX file-system races via algorithmic complexity attacks. In: S&P. 2009 (p. 343).

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (p. 312).

[10] Congcong Chen, Jinhua Cui, Gang Qu, and Jiliang Zhang. Write+Sync: Software Cache Write Covert Channels Exploiting Memory-disk Synchronization. In: TIFS (2024) (p. 343).

[11] Jonathan Corbet. Trees I: Radix trees. 2006. URL: https://lwn.net/Articles/175432/ (pp. 314, 321).

[12] Jonathan Corbet. Trees II: red-black trees. 2006. URL: https://lwn.net/Articles/184495/ (pp. 314, 315, 323).

[13] Scott A Crosby and Dan S Wallach. Denial of Service via Algorithmic Complexity Attacks. In: USENIX Security. 2003 (p. 343).

[14] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel. 2022. URL: https://syst3mfailure.io/corjail/ (pp. 336, 338).

[15] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In: S&P. 2016 (pp. 310, 342).

[16] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In: CCS. 2000 (pp. 310, 342).

[17] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In: DSN. 2017 (p. 342).

[18] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks. In: FC. 2024 (p. 310).

[19] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: S&P. 2023 (p. 310).

[20] Luke Gix. FUSE for Linux Exploitation 101. 2022. URL: `https://exploiter.dev/blog/2022/FUSE-exploit.html` (p. 355).

[21] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: reviving the CRIME attack. In: Unpublished manuscript (2013) (pp. 310, 342).

[22] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 310, 342).

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 312, 344).

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 310, 325).

[25] h0mbre. Escaping the Google kCTF Container with a Data-Only Exploit. 2023. URL: `https://h0mbre.github.io/kCTF_Data_Only_Exploit%5C#` (p. 336).

[26] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level. In: EuroSys. 2021 (p. 343).

[27] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In: Journal of Computer Security (1992) (p. 344).

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (pp. 310, 312).

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (p. 312).

[30] javierprtd. No CVE for this bug which has never been in the official kernel. 2023. URL: `https://soez.github.io/posts/no-cve-for-this.-It-has-never-been-in-the-official-kernel/` (pp. 336, 338).

[31] Qisheng Jiang and Chundong Wang. Sync+Sync: A Covert Channel Built on fsync with Storage. In: USENIX Security. 2024 (p. 342).

[32] Choo Yi Kai. A new method for container escape using file-based DirtyCred. 2023. URL: https://starlabs.sg/blog/2023/07-a-new-method-for-container-escape-using-file-based-dirtycred/ (p. 355).

[33] John Kelsey. Compression and Information Leakage of Plaintext. In: Fast Software Encryption. 2002 (pp. 310, 342).

[34] Michael Kerrisk. futex(2) — Linux manual page. https://man7.org/linux/man-pages/man2/futex.2.html. 2023 (p. 318).

[35] Suraiya Khan and Issa Traore. A Prevention Model for Algorithmic Complexity Attacks. In: DIMVA. 2005 (p. 343).

[36] Amit Klein and Benny Pinkas. From IP ID to Device ID and KASLR Bypass. In: USENIX Security. 2019 (p. 312).

[37] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (pp. 310, 342, 344).

[38] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 342).

[39] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In: USENIX Security. 2016 (p. 344).

[40] Butler W Lampson. A note on the confinement problem. In: Communications of the ACM 16.10 (1973), pp. 613–615 (p. 342).

[41] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In: USENIX Security. 2023 (pp. 310, 331, 343).

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security. 2018 (p. 342).

[43] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 310, 311, 331, 336, 338, 342, 343).

[44] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025 (p. 307).

[45] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (p. 337).

[46] 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 (p. 310).

[47] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In: NDSS. 2025 (p. 342).

[48] OpenSSL. Security Policy. 2024. URL: `https://www.openssl.org/policies/general/security-policy.html` (p. 343).

[49] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 310, 342).

[50] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Using Trāṭr to tame Adversarial Synchronization. In: USENIX Security. 2022 (pp. 310, 342).

[51] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In: CCS. 2017 (p. 343).

[52] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In: AsiaCCS. 2023 (pp. 325, 344).

[53] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (p. 331).

[54] Juliano Rizzo and Thai Duong. The CRIME attack. In: ekoparty security conference. Vol. 2012. 2012 (pp. 310, 342).

[55] 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 (pp. 310, 342).

[56] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical Timing Side Channel Attacks on Memory Compression. In: S&P. 2023 (pp. 310, 342, 343).

[57] Chaoqun Shen, Jiliang Zhang, and Gang Qu. MES-attacks: Software-controlled covert channels based on mutual exclusion and synchronization. In: DAC. 2023 (pp. 310, 311, 343).

[58] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks Against a NIDS. In: ACSAC. 2006 (p. 343).

[59] Xiaoshan Sun, Liang Cheng, and Yang Zhang. A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis. In: IEEE International Conference on Communications (ICC). 2011 (p. 343).

[60] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In: EuroSys. 2011 (pp. 310, 342).

[61] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In: CHES. 2003 (p. 342).

[62] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security. 2018 (p. 344).

[63] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (pp. 310, 342).

[64] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In: CCS. 2015 (pp. 310, 342, 344).

[65] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In: USENIX Security. 2022 (p. 342).

[66] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way! 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf (pp. 336, 338, 339).

[67] Haocheng Xiao and Sam Ainsworth. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. In: ASPLOS. 2023 (p. 344).

[68] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (pp. 336, 338).

[69] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 310, 325, 342).

[70] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W Fletcher. Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In: USENIX Security. 2023 (p. 344).

[71] Jiliang Zhang, Chaoqun Shen, and Gang Qu. Mex+Sync: Software Covert Channels Exploiting Mutual Exclusion and Synchronization. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023) (p. 343).

[72] Xiaochen Zou and Zhiyun Qian. Exploit esp6 modules in Linux kernel. 2022. URL: https://etenal.me/archives/1825 (pp. 336, 338).

10. Appendix

10.1. Appendix

```c
struct signal_struct;
struct list_head {
  struct list_head *next, *prev;
};
struct hlist_head {
  struct hlist_node *first;
};
struct hlist_node {
  struct hlist_node *next, **pprev;
};
struct k_itimer {
  ...
  u32 it_id;
  struct hlist_node t_hash;
  struct signal_struct *it_signal;
  ...
};
DEFINE_HASHTABLE(posix_timers_hashtable, 9);

// Calculates hash for hash table
int hash(struct signal_struct *sig, unsigned int nr) {
  return hash_32(hash32_ptr(sig) ^ nr, 9);
}

// Iterates throught the bucket's linked list to find
// k_timer matching sig and id
struct k_itimer *__posix_timers_find(
              struct hlist_head *head,
              struct signal_struct *sig,
              u32 id) {
  struct k_itimer *tim;

  hlist_for_each_entry(tim, head, t_hash) {
    if ((tim->it_signal == sig) && (tim->it_id == id))
      return tim;
  }
  return NULL;
}

// k_timer lookup with id
struct k_itimer *posix_timer_by_id(u32 id) {
  struct hlist_head *head;
  struct signal_struct *sig = current->signal;

  head = &posix_timers_hashtable[hash(sig, id)];
  return __posix_timers_find(head, sig, id);
}
```

Listing 9.1 Simplified C equivalent for a timer lookup.

```
1  posix_timer_by_id:
2    push   rbp
3    push   r13
4    push   rbx
5    // sign = current->signal
6    mov    gs:0x32880, rax
7    mov    0xbf0(rax), rdx
8    mov    rdx, rax
9    // h = hash(sign, id)
10   shr    $0x20, rax
11   xor    rdx, rax
12   xor    r13d, eax
13   imul   $0x61c88647, eax, eax
14   shr    $0x17, eax
15   // head = &posix_timers_hashtable[h]
16   mov    posix_timers_hashtable(, rax, 8), rbx
17   // node = (hlist_node *)head
18 0x50:
19   // tim = (k_itimer *)container_of(node,k_itimer,t_hash)
20   sub    $0x10, rbx
21   test   rbx, rbx
22   je     <posix_timer_by_id+0x6b>
23   // (tim->it_signal == sig)
24   mov    0x60(rbx), rax
25   cmp    rax, rdx
26   je     <posix_timer_by_id+0x86>
27 0x62:
28   // tim = (k_itimer *)tim->t_hash.next
29   mov    0x10(rbx), rbx
30   test   rbx, rbx
31   jne    <posix_timer_by_id+0x50>
32 0x6b:
33   // return NULL
34   xor    ebx, ebx
35   mov    rbx, rax
36   pop    rbx
37   pop    r13
38   pop    rbp
39   jmp    __x86_return_thunk
40 0x86:
41   // (tim->it_id == id)
42   cmp    0x34(rbx), r13d
43   jne    <posix_timer_by_id+0x62>
44   // return tim
45   mov    rbx, rax
46   pop    rbx
47   pop    r13
48   pop    rbp
49   jmp    <__x86_return_thunk>
```

Listing 9.2 ASM instructions executed for the lookup.

Figure 9.19: POSIX timer lookup from the timer's hash table using the function `posix_timer_by_id`. The instructions in blue represent the instructions executed for each element contained in the linked list of the hash bucket.

```
sys_timer_create():
  sign = current.signal
  id = sign.next_id++
  tim = k_itimer(sign, id)
  h = timer_hash(id, sign)
  hbucket =
   posix_timers_hashtable[h]
  hbucket.append(tim)
  return id
```

Listing 9.3: Alter occupancy of POSIX timer's hash table.

```
sys_clock_gettime(id):
  sign = current.signal
  h = timer_hash(id, sign)
  hbucket =
   posix_timers_hashtable[h]
  for tim in hbucket:
    if tim.sign == sign and
        tim.id == id:
      return tim.get_time()
  return ERROR
```

Listing 9.4: Probe occupancy of POSIX timer's hash table.

```
sys_msgcreate(key):
  ipc_ns = current.ipc_ns
  ipc_ids = ipc_ns.get_ids()
  msgq = msg_queue(key)
  h = ipc_ids_hash(key)
  ipc_ids[h].append(
    msgq.ipcp)
  return msgq.ipcp.id
```

Listing 9.5: Alter occupancy of IPC key's hash table.

```
sys_msgget(key):
  ipc_ns = current.ipc_ns
  ipc_ids = ipc_ns.get_ids()
  h = ipc_ids_hash(key)
  hbucket = ipc_ids[h]
  for ipcp in hbucket:
    if ipcp.key == key:
      return ipcp.id
  return ERROR
```

Listing 9.6: Probe occupancy of IPC key's hash table.

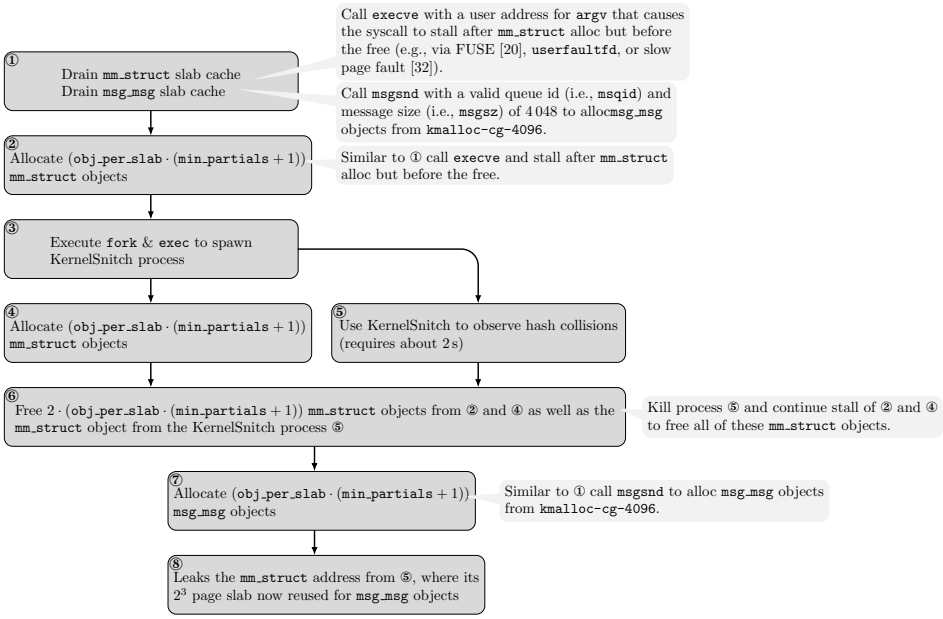


Figure 9.20: Detailed workflow of the cross-cache reuse.

Table 9.1: Evaluation results for data container structures: **X to Y** denotes the difference from **X** compared to **Y** elements. **1 to 0** signifies no hardware-agnostic amplification, while **3 to 0** indicates hardware-agnostic amplification with 2 extra elements. ✶ denotes the decrease in FPR and FNR from no amplification (i.e., **1 to 0**) to structure- and hardware-agnostic amplification (i.e., **3 to 0**). ☆ denotes Linux kernel v5.15, where for the other three we used v6.5.

	Container instance	**W/o struct-agnostic**				**W struct-agnostic**				**Reduction ✶**	
		1 to 0		**3 to 0**		**1 to 0**		**3 to 0**			
		FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	in FPR	in FNR
		%	%	%	%	%	%	%	%	%	%
Intel i7-1260P	`posix_timers_hashtable`	1.8	10.0	0.0	9.4	0.0	0.0	0.0	0.0	100	100
	`futex_hash_table`	0.0	0.6	0.0	0.6	0.0	0.0	0.0	0.0	100	100
	`ipc_ids.key_ht`	3.1	2.8	1.7	2.7	0.1	0.1	0.0	0.1	100	97
	`ipc_ids.ipcs_idr.root_rt`	1.7	3.9	-	-	0.0	0.0	-	-	100	100
Intel i7-1165G7	`posix_timers_hashtable`	0.9	9.1	0.0	7.3	0.0	0.3	0.0	0.3	100	97
	`futex_hash_table`	0.0	0.5	0.0	0.5	0.0	0.2	0.0	0.1	100	76
	`ipc_ids.key_ht`	4.5	18.6	0.0	6.9	0.1	0.8	0.0	0.3	100	98
	`ipc_ids.ipcs_idr.root_rt`	0.0	32.9	-	-	0.0	1.7	-	-	100	95
☆ Intel i7-12700	`posix_timers_hashtable`	0.0	4.2	0.0	0.3	0.0	0.0	0.0	0.0	100	100
	`futex_hash_table`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100	100
	`ipc_ids.key_ht`	1.7	0.8	0.0	0.0	0.2	0.0	0.0	0.0	100	100
	`ipc_ids.ipcs_idr.root_rt`	2.7	5.3	-	-	0.0	0.2	-	-	100	96
Intel Xeon Gold 6530	`posix_timers_hashtable`	0.2	0.3	0.0	0.0	0.0	0.0	0.0	0.0	100	100
	`futex_hash_table`	0.0	4.3	0.0	4.3	0.0	0.0	0.0	0.0	100	100
	`ipc_ids.key_ht`	0.7	0.8	0.0	0.0	0.1	0.0	0.0	0.0	100	100
	`ipc_ids.ipcs_idr.root_rt`	0.0	0.9	-	-	0.0	0.0	-	-	100	100

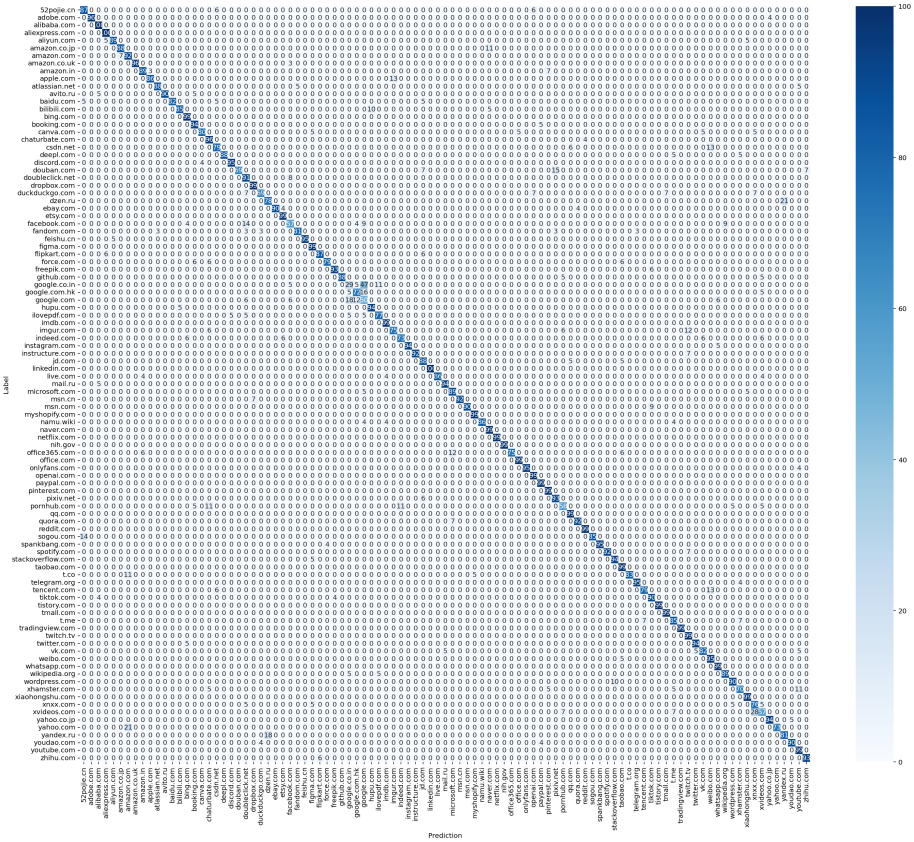


Figure 9.21: Confusion matrix of our KernelSnitch website fingerprinting attack with an F_1 score of 89.5 %.

10 When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks

Publication Data

Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025

Contributions

The author of this thesis is the main author of this work. The author's contributions are the proposal of *side-channel analysis of kernel defenses* and *relaible and stable kernel exploitation*. From *location disclosure attacks*, the author's contributions are the development and evaluation of the POCs for leaking the location of kernel heap objects, kernel stacks, and page tables. Finally, the author's contributions are also most of the written text.

When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks

Lukas Maar Lukas Giner Daniel Gruss Stefan Mangard
Graz University of Technology

Abstract

Over the past decade, the Linux kernel has seen a significant number of memory-safety vulnerabilities. However, exploiting these vulnerabilities becomes substantially harder as defenses increase. A fundamental defense of the Linux kernel is the randomization of memory locations for security-critical objects, which greatly limits or prevents exploitation.

In this paper, we show that we can exploit side-channel leakage in defenses to leak the locations of security-critical kernel objects. These location disclosure attacks enable successful exploitations on the latest Linux kernel, facilitating reliable and stable system compromise both with re-enabled and new exploit techniques. To identify side-channel leakages of defenses, we systematically analyze 127 defenses. Based on this analysis, we show that enabling any of 3 defenses – enforcing strict memory permissions or virtualizing the kernel heap or kernel stack – allows us to obtain fine-grained TLB contention patterns via an Evict+Reload TLB side-channel attack. We combine these patterns with kernel allocator massaging to present location disclosure attacks, leaking the locations of kernel objects, i.e., heap objects, page tables, and stacks. To demonstrate the practicality of these attacks, we evaluate them on recent Intel CPUs and multiple kernel versions, with a runtime of 0.3 s to 17.8 s and almost no false positives. Since these attacks work due to side-channel leakage in defenses, we argue that the virtual stack defense makes the system less secure.

1. Introduction

The security of modern systems relies on privilege levels. While user programs have lower privileges, the operating system kernel has higher

privileges and can typically access all system memory. Thus, the system's security depends directly on the security of the kernel. However, kernels – such as Linux – are also complex, which often unintentionally introduces software vulnerabilities, many of which remain unknown for years. To generically limit the impact of these vulnerabilities, kernels employ several defenses that limit what a bad actor can do after exploiting a software bug. A fundamental defense in the Linux kernel is the randomization of memory locations for security-critical objects. This strategy prevents exploitation outright or forces the bad actor to locate these randomized locations before achieving system compromise.

Side-channel attacks offer a promising approach to circumvent these randomization-based defenses and have been studied extensively. Numerous works [1, 14, 21, 31, 38, 42] have exploited a side channel in the Translation Lookaside Buffer (TLB), a CPU buffer that stores virtual-to-physical address translations. These TLB side channels have allowed partial bypasses of Kernel Address Space Layout Randomization (KASLR) [11], which randomizes parts of the kernel, e.g., the kernel code, module code, and Direct-Physical Map (DPM). Specifically, these works break code or physical KASLR, which refers to leaking the base address of randomized code sections or the DPM. Multiple kernel exploits have used these KASLR breaks [12, 26, 41], including those from Google Project Zero and Google's bug bounty program.

However, existing techniques do not reveal the locations of security-critical kernel objects, a requirement for many attacks to compromise the system. At the same time, as defenses become more integrated – particularly in the memory mapping subsystem – the risk increases that these protections, while intended to enhance security, may unintentionally expose the system to more precise leaks. As a result, it is unclear *which*, if any, of the defenses actually allow more precise leakage.

In this paper, we show that while kernel defenses are valuable in mitigating vulnerability exploitation, enabling any of 3 defenses allows for side-channel attacks to deduce the locations of security-critical kernel objects in a way that allows reliable and stable privilege escalation on modern Linux kernels. We refer to these as location disclosure attacks.

To identify kernel defenses that allow these location disclosure attacks, we perform a systematic side-channel analysis. We analyze all 127 defenses recommended by the Kernel Self-Protection Project (KSPP) [61] or used within Google's KernelCTF [12] bug bounty program. These defenses

include protection against various exploit techniques, such as cross-cache reuses [33, 44, 65, 68] – that exploit the memory reuse by a kernel allocator – and kernel code tampering attacks [10]. We classify these defenses into 5 categories. We then observe that from one category, 3 defenses – enforcing strict memory permissions or virtualizing the kernel heap or kernel stack – modify the memory mapping to create exploitable access patterns in the TLB. These defenses change the mapping so that objects are accessed with a fine-grained mapping of 4 kB instead of 2 MB, which is how most kernel memory is accessed. This results in using 4 kB TLB entries, with contention patterns observable via a side channel.

We then present location disclosure attacks that leak the locations of kernel objects. Combining strategic kernel allocator massaging with TLB contention patterns allows the leak of page-aligned object locations and consequently deduces all sub-page granular object locations, all attacker-controlled. To perform these attacks, a bad actor must first load the object's address into the TLB. However, the kernel does not provide a way to load only one target kernel address into the TLB, as even the simplest syscall accesses and loads multiple addresses. Instead, we use a so-called access primitive multiple times with different arguments, which loads numerous addresses – including the target address – into the TLB. This creates multiple TLB contention patterns, which we leak via an Evict+Reload TLB side-channel attack using 2 known and 1 novel distinguishing primitive, i.e., distinguish 2 MB from 4 kB mappings. Due to strategic prior massaging, we use these patterns to deduce the locations of most security-critical kernel objects, i.e., `pipe_buffer`, `msg_msg`, `cred`, `file`, `seq_file`, page table (all levels), and kernel stack.

To demonstrate the practicality of these disclosure attacks, we leak object locations on recent Intel CPUs ranging from the 8th to the 14th generation and generic kernels ranging from v5.15 to v6.8. We evaluate on an idle and stressed system. For an idle system, these attacks require between 0.3 s to 17.8 s with almost no false positives. For a stressed system, the false positives increase to about 7 % despite being close to full CPU load with significant TLB pressure.

Using our disclosure attacks, we can perform privilege escalation on modern kernels (e.g., v6.8) without crashes and nearly 100 % reliability. We show 3 results: First, our disclosure attacks re-enable exploit techniques that have been largely prevented. Second, disclosure attacks enable a new exploit technique that was previously not possible due to the limited capabilities

of most vulnerabilities. Third, we even argue that the virtual kernel stack defense reduces security.

Finally, we discuss the security implications of our disclosure attacks, e.g., that the exploitation becomes substantially more reliable and stable with already known security-critical kernel objects. We also discuss challenges inherent in fully mitigating location disclosure attacks, e.g., preventing the kernel from using 4 kB mappings for kernel objects.

Contributions. The main contributions of this work are:

(1) Side-Channel Analysis of Kernel Defenses: We systematically analyze all 127 defenses recommended by the KSPP or used within Google's bug bounty program for their side-channel leakage, showing that 3 leave fine-grained, exploitable TLB contention patterns.
(2) Location Disclosure Attacks: We present disclosure attacks combing allocator massaging with Evict+Reload-style TLB attacks to leak these fine-grained patterns and deduce the locations of security-critical kernel objects, i.e., heap objects, page tables, and stacks. We evaluate our attacks on recent Intel CPUs and kernel versions.
(3) Reliable and Stable Kernel Exploitation: We show that our disclosure attacks allow privilege escalation without crashes and nearly 100 % reliability on modern systems, e.g., Linux kernels v6.8. They re-enable a new exploit technique and previously prevented exploit techniques.

Outline. Section 2 provides the background. Section 3 presents the workflow. Section 4 discusses our side-channel analysis of the defenses. Section 5 combines massaging with exploitation of these side-channel leaks for disclosure attacks. Section 6 evaluates these attacks. Section 7 shows how the leakage can be used in kernel exploits. Section 8 provides a discussion. Section 9 concludes our work.

2. Background

This section provides the necessary background for this work.

Kernel Allocators. The Linux kernel provides 3 allocators (i.e., *page allocator*, *slab allocator*, and *virtual memory allocator*), where Figure 10.1 shows the memory areas used.

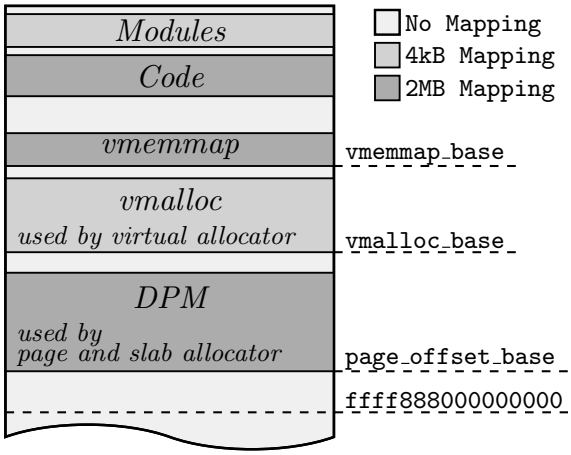


Figure 10.1: Virtual memory layout of the x86_64 Linux kernel.

First, the *page allocator* [30] divides the Direct-Physical Map (DPM) [48] – a linear virtual mapping of (typically) the entire physical memory – into page-order memory chunks. This allocator combines memory allocation with free memory coalescing. Simply put, it provides global and per-CPU page-order free lists. Both free lists allow allocation of physically contiguous page-order memory chunks, where the kernel first allocates/deallocates chunks from the per-CPU lists. If the per-CPU lists are exhausted on allocation or exceed a maximum capacity on deallocation, the kernel resorts to the global lists.

Second, the *slab allocator* allocates page-order chunks – used as slabs – from the page allocator, where the slabs cache free and available memory slots [5, 29]. This allocator provides two types of caches, both of which use slab pages for allocating and deallocating objects: dedicated and generic caches. Dedicated caches are used for frequent object allocation with `kmem_cache_alloc`. Generic caches are used for allocating less frequently used objects and have multiple allocator caches matched to different sizes. When allocating objects from a generic cache with `kmalloc`, the kernel first matches the object size to one of these caches and then returns a free and available memory slot from that cache.

Third, the *virtual memory allocator* uses `vmalloc` to provide a virtually contiguous memory with a specific mapping.

SLUBStick. Maar et al. [44] introduced a timing side channel of the slab allocator to detect new slab page allocations. It works by timing

object allocations from user space. Fast allocations indicate that the currently active slab page has returned the memory slot. Slow allocations indicate a new slab page allocation by the page allocator that returns a slot for the object. By grouping object allocations according to their timing, SLUBStick determines all objects on a slab page.

Kernel Memory Mapping. Figure 10.1 shows the virtual memory layout of the Linux kernel with 5 relevant areas for this work, all of which are randomized due to KASLR [11].

Code and *Modules* contain the instructions of the kernel binary and inserted modules. The *DPM* is a virtual memory area of (typically) the entire physical memory, mostly mapped with 2 MB pages. Crucially, the kernel heap allocated by the slab allocator accesses the DPM directly. On x86_64 systems, the kernel places the DPM at a random 1 GB-aligned address – the `page_offset_base` – between `ffff888000000000` and `ffffc88000000000`. The *vmalloc* area (mapped with 4 kB pages) contains memory slots allocated with the virtual allocator. This area is randomly located to a 1 GB-aligned address – the `vmalloc_base` – between the DPM's end and `vmemmap`'s start. The *vmemmap* area is a virtual memory mapping that stores metadata for each physical page. More specifically, this virtual mapping is an array of 64 B `page` objects that store this metadata and is indexed by the physical frame number. It is located at a 1 GB-aligned address – the `vmemmap_base` – between `vmalloc`'s end and `ffffffffe0000000`.

Kernel Exploitation. Most kernel exploits that exploit memory-corruption vulnerabilities work similarly: A bad actor initially triggers the vulnerability, such as Out-Of-Bounds (OOB) or Use-After-Free (UAF), to falsely put an object – often referred to as the victim object – in a free state. They then reuse the victim's memory slot for a different object. There are two variants of reuse attacks: First, they perform an in-cache reuse and reuse the victim object as another object from the same allocator cache, e.g., `kmalloc-*`. However, this limits reuse to objects with the same (or similar) size and allocation properties, as heap separation prevents direct reuse of the victim as a security-critical object. Second, they perform a cross-cache reuse by freeing all memory slots on the slab page that contains the victim object, causing page recycling. They then reclaim the page (typically) used for a security-critical purpose, e.g., DirtyCred [35] as a credential reference, CVE-2022-27666 [72] as a `msg_msg`, DirtyPage [36]/CVE-2020-29660 [19] as a `pipe_buffer`, or Dirty PageTable [66]/SLUBStick [44] as a page table.

Figure 10.2: The high-level overview of our work.

Translation-Lookaside Buffer. The TLB is a per-core CPU cache that stores virtual-to-physical address translations. When an address translation is found in a TLB entry, it eliminates the need to perform a page-table walk for memory access, resulting in a speedup. There are typically two levels of TLBs, with the first level split between data (DTLB) and instructions (ITLB), while the second level (STLB) is shared between data and instructions. Each TLB is a set-associative cache, i.e., split into sets and ways. For example, an Alder Lake CPU might feature two STLBs of 128 sets with 8-way associativity each, one for 4 kB and 2 MB/4 MB pages, and one for 4 kB and 1 GB pages. When a virtual address is not found in any TLB, a page-table walk is performed, and an existing entry in the TLB is evicted to store the new translation.

3. High-Level Overview

Our work is based on 3 components, as shown in Figure 10.2.

First, we show that several kernel defenses leave exploitable, fine-grained TLB contention patterns (see Section 4). Initially, we analyze all 127 defenses recommended by the KSPP [61] or used by KernelCTF [12], which protects against techniques, e.g., code manipulation [10] and cross-cache reuse [33, 44, 68]. We categorize them into 5 categories based on how they improve security. We then analyze whether they change the memory mapping to create fine-grained TLB patterns, resulting in 3 defenses of a particular category.

Second, we show that combining kernel allocator massaging with these patterns enables the leaking of the location of target objects (see Section 5). To achieve this, we perform an Evict+Reload TLB side-channel attack and obtain the leakage of each exploitable defense. Using these leaks, we show that due to the prior massaging, we can deduce the location of most security-critical objects, i.e., heap objects, page tables, and kernel stacks, which are all popular exploitation targets such as `msg_msg` [3, 9, 23, 40, 52, 72], `pipe_buffer` [36, 53, 63], `cred` [16, 35], `seq_file` [18, 27, 59], `file` [35, 56, 64, 69], page table [44, 51, 65, 66], and kernel stack [26, 67, 70]. We demonstrate the practicality of these attacks on recent Intel CPUs and multiple kernel versions (see Section 6).

Third, we show that by using our disclosure attacks, we can build reliable and stable exploits (see Section 7): First, they re-enable exploit techniques

that have been largely prevented. Second, they enable a new technique that was not previously possible due to the limited capabilities of most vulnerabilities in their initial state. Third, for one defense, we even argue that the defense provides less security due to disclosure attacks. We then discuss (see Section 8) the security implications of the disclosure attacks and the challenges of full mitigation.

Threat Model. We assume an unprivileged user with code execution, a typical scenario for the last stage of a full-chain exploit [58]. We also consider the presence of an exploit primitive, such as kernel UAF or OOB write, due to a kernel heap bug. We assume that all upstream defenses available in v6.9 (i.e., the latest version when we started our work) are enabled and exclude defenses that require paid subscriptions (e.g., AUTOSLAB [33]), in line with prior work [16, 35, 44, 70]. In line with our evaluation CPUs, while KPTI is included in the kernel binary, it is disabled by the CPU.

4. Side-Channel Analysis of Kernel Defenses

In this section, we detail our systematic analysis of all 127 kernel defenses recommended by the KSPP [61] or used within Google's KernelCTF [12] bug bounty program. We aim to determine which kernel defenses introduce exploitable fine-grained TLB contention patterns. We first organize all kernel defenses into 5 categories. We then show that 3 defenses of one category introduce exploitable fine-grained leakage.

4.1. Requirements

There are two requirements for defenses to leave exploitable contention patterns in the TLB. We first discuss these and then describe how kernel defenses satisfy these requirements.

R1. The identified kernel defenses must produce different TLB contention patterns when applied than when not applied. Thus, the first requirement is that applying a defense must change the memory mapping. We refer to a mapping change in *where* or *how* kernel objects are mapped, potentially creating exploitable TLB contention patterns on object access.

R2. Since most of the accessed kernel memory is mapped as 2 MB pages [7], TLB contention patterns mainly occupy 2 MB entries. Even if we leak

which 2 MB entries the target object occupies, the offset within that 2 MB page remains unknown. Thus, as a second requirement, the defense must alter the mapping of objects from 2 MB to 4 kB pages, creating contention patterns in 4 kB TLB entries, so that we can leak the 4 kB location of the object first and narrow it down to its sub-page granular locations subsequently. This can be done either by statically switching from a 2 MB to a 4 kB mapped region or by dynamically changing the mapping.

Since defenses satisfying **R2** are a subset of defenses satisfying **R1**, **R1** helps to filter out defenses that do not satisfy **R2**, allowing for efficient and accurate kernel defense analysis.

Unexploitability of 2 MB Mappings. As we will show in Section 5, our disclosure attacks require full control over the smallest memory granularity leaked by TLBs. With 2 MB mappings, adversaries must ensure all controlled objects occupy and map exclusively to an entire 2 MB memory chunk, which is infeasible because: First, the largest slab size is 32 kB and, second, slab adjacency to fill exclusively only to a 2 MB memory chunk is almost impossible, e.g., due to `CONFIG_SHUFFLE_PAGE_ALLOCATOR`.

4.2. Classification of Kernel Defenses

From the 127 defenses, we filter out those that do not apply to any of our evaluated kernels (i.e., v5.15, v6.5, v6.6, and v6.8), e.g., `CONFIG_RANDOM_TRUST_BOOTLOADER` is not available in these versions. This leaves us with 114 defenses (see Figure 10.3), which we divide into the *Memory Mapping Change*, *Reduce Attack Surface*, *Add Checks*, *Poisoning/Cleanup*, and *Others* categories. We then analyze whether these categories have the potential to satisfy our requirements and find that *Memory Mapping Change* satisfies **R1**. Finally, we show that multiple defenses within this category satisfy **R2**, leaving exploitable contention patterns in 4 kB TLB entries.

Non-Exploitable Categories. To filter the non-exploitable categories from the *Memory Mapping Change* category that satisfy **R1**, we automatically identify all files containing an `#ifdef` of a kernel defense. We then automatically determine whether the file is directly responsible for the memory mapping, e.g., `vmlinux.lds.S`, or is located in a directory that handles mappings, e.g., `arch/*/mm/`. In these cases, we manually analyze the files and determine whether they contribute to memory mapping changes. Several automatically identified defenses turned out as

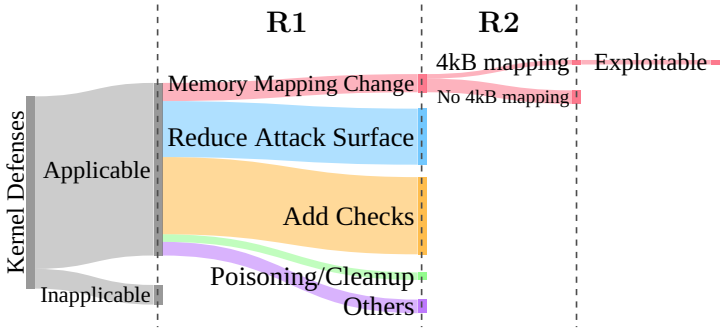


Figure 10.3: Our classification of kernel defenses shows the *Memory Mapping Change* defenses inducing exploitable TLB contention patterns that leak the precise location of objects.

false negatives on manual analysis, e.g., `CONFIG_STRICT_DEVMEM` includes checks in `arch/x86/mm/pat/memtype.c` but does not contribute to mapping changes. After the analysis, we are left with 12 defenses contributing to mapping changes.

For completeness, we briefly describe the other non-exploitable categories *Reduce Attack Surface*, *Add Checks*, *Poisoning/Cleanup*, and *Others*, where we use manual classification based on the following definitions: *Reduce Attack Surface* limits the amount of kernel code accessible to an application [20, 25], e.g., resetting `CONFIG_X86_VSYSCALL_EMULATION` removes virtual syscalls handling. *Add Checks* defenses insert security checks, ensuring the integrity of parts of the kernel, e.g., `CONFIG_SLAB_FREELIST_HARDENED` protects slab metadata. We classify defenses as *Poisoning/Cleanup* if they poison or cleanup registers (e.g., `CONFIG_ZERO_CALL_USED_REGS`) or memory (e.g., `CONFIG_PAGE_POISONING` or `CONFIG_INIT_ON_FREE_DEFAULT_ON`). Finally, *Others* defenses do not fit in the other categories, e.g., `CONFIG_SCHED_CORE` permits core scheduling. To summarize, none of these categories contribute to memory mapping changes, and, thereby, they do not satisfy **R1**.

Defenses that Change the Memory Mapping. *Memory Mapping Change* defenses can be divided into those that change the object's mapping to 4 kB pages (satisfying **R2**) and those that do not. For those that do not, several of them change the object's location by separating them according to their security context. Examples include `CONFIG_KMALLOC_CG` with coarse-grained separation, `CONFIG_KMALLOC_SPLIT_VARSIZE` to separate elastic objects [3], and `CONFIG_RANDOM_KMALLOC_CACHES` to ran-

domly assign dedicated caches to allocation sites. Other defenses that also do not change the mapping are `CONFIG_SHUFFLE_PAGE_ALLOCATOR` and `CONFIG_RANDSTRUCT_FULL`, which randomize page allocations or the members within a struct. Another example is `CONFIG_STRICT_KERNEL_RWX`, which only sets code permissions on a 2 MB granularity. Since none of these defenses change the object's mapping to 4 kB pages, they do not satisfy **R2**.

This leaves 3 defenses that change the object's mapping to 4 kB pages. `CONFIG_STRICT_MODULE_RWX` satisfies **R2** as follows: It must set the permission of the virtual memory and the DPM of the module code to non-writable [10, 61]. To achieve permission settings in the DPM, this defense splits the 2 MB pages of the DPM into 4 kB, allowing legitimate use of other pages from the former 2 MB page, e.g., for the heap. `CONFIG_SLAB_VIRTUAL` and `CONFIG_VMAP_STACK` satisfy **R2** as they use virtual memory mapped with 4 kB pages.

Discussion. A false positive is when an identified defense leaves no exploitable TLB contention patterns. However, all of our identified defenses leave exploitable patterns that leak locations of target objects. A false negative is when a defense allows location leakage, but we missed it. To minimize false negatives, we strictly defined our requirements **R1** and **R2** and systematically analyzed the defenses. We also performed a bottom-up approach and tried to find defenses that include mapping changing functions such as `set_memory_ro`.

We reveal 3 exploitable and 124 non-exploitable defenses. This provides encouraging results for defenders, as only 3 defenses require hardening against our attacks. Researchers can focus on these instead of all 127 defenses.

Takeaway 10.1

Our analysis shows that 3 defenses change the memory mapping of kernel objects to 4 kB pages, which creates exploitable contention patterns in 4 kB TLB entries.

5. Exploitation of Kernel Defenses

To leak the location of a target object, we need to load its page-aligned address into the TLB. However, even the simplest syscall accesses multiple kernel addresses and loads them into the TLB. Instead, we call a so-called access primitive multiple times with different arguments. These access primitives load the addresses of a couple hundred to a thousand accessed objects – including the target address – and, therefore, create TLB contention patterns. By strategically massaging the kernel allocators beforehand, we use these patterns to infer the page-aligned location of the target object. We then deduce all sub-page granular object locations within the leaked page, all controlled by the attacker. Below, we set the challenges of exploiting **D1-3** and massaging the allocators for location leakage, while Sections 5.1 to 5.3 solve them.

C1. The first challenge is to ensure that our target object is located on 4 kB mappings. Therefore, on object access, its page-aligned address is now loaded into a 4 kB TLB entry, resulting in 4 kB TLB entries are occupied.

C2. The second challenge is to minimize TLB noise from uncontrolled accesses within the access primitives. Reducing the noise is critical, as otherwise, it could lead to the target TLB entry being misoccupied, resulting in misclassification or the inability to locate the target 4 kB TLB entry.

C3. The third challenge is to deduce the location of the target kernel object from the contention patterns in 4 kB TLB entries. To achieve this, we first need to determine the 4 kB TLB entry of our target kernel object (**C3.1**). For reference, when leaking `msg_msg`, we need to non-trivially reduce the 976 TLB entries of its access primitive to 1 target entry. Finding this TLB entry gives us the page-aligned kernel address of the object. Next, we must derive the object locations within this leaked page address (**C3.2**).

5.1. D1: Strict Kernel Memory Permissions

By massaging the kernel allocators combined with exploiting the `CONFIG_STRICT_MODULE_RWX` defense, we can solve **C1-3** and leak the location of heap objects and page tables.

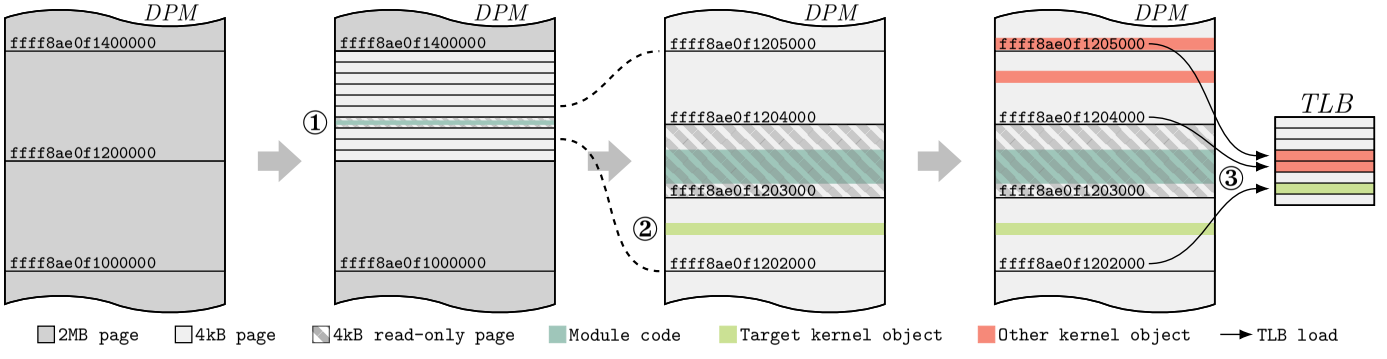


Figure 10.4: Exploiting `CONFIG_STRICT_MODULE_RWX` to create TLB contention patterns via an access primitive. The insertion of a module ① causes a split of a 2 MB page into 4 kB pages to set the module's page to read-only. We then allocate the target kernel object we want to leak ②, which will be on a 4 kB page. Finally, we execute the access primitive ③, which accesses the target with other objects, loading their page-aligned addresses into the TLB and creating the contention pattern on the 4 kB entries.

Protection Scheme. Kernel memory containing writable code is an easy target for control-flow redirection [10, 61]. As a mitigation, the kernel supports `CONFIG_STRICT_*_RWX` to guarantee no kernel code is writable [61]. Specifically, the kernel can protect dynamically loaded module code (i.e., `CONFIG_STRICT_MODULE_RWX`) this way: When loading module code into a virtual memory range, the kernel restricts this range to be executable and read-only and the corresponding range in the DPM to be read-only. However, as the DPM is mapped mainly as 2 MB pages, the kernel has to split – prior to changing permissions – this 2 MB page into 4 kB pages. Hence, physically adjacent pages can remain writable and be used by the kernel, e.g., for the kernel heap.

TLB Contention Pattern. Figure 10.4 shows exploiting TLB contention patterns created by this defense combined with allocator massaging. In the first stage, we load a module, forcing a 2 MB to 4 kB page split ①. The kernel then sets the page within the DPM that contains the module's code (i.e., blue memory range) to read-only. Unprivileged kernel module loading can be done by opening a socket that does not have its kernel driver loaded.

In the second stage, we allocate a target object (i.e., green memory range) to claim a memory location mapped as a 4 kB page ②, using its so-called allocation primitive. To claim a 4 kB mapped slot, we drain the lower page-order free lists of the page allocator before loading the module by allocating many dummy objects. Due to the page allocator's intrinsic behavior, when the lower page-order free lists are drained, it partitions higher page-order chunks and uses them for the lower page-order allocations [16, 44, 65]. After these free lists are drained, the module allocation uses a chunk with an adjacent free chunk, located on the split 2 MB page. Now, subsequent object allocations likely claim one of these adjacent chunks, solving **C1**.

In the third stage, we use an access primitive that accesses multiple kernel addresses and induces TLB contention. The TLB contention pattern consists of the page-aligned address of our target kernel object (i.e., green memory range) and addresses of other accessed objects (i.e., red memory areas).

Leaking the Object's Location. While inferring the location of our target object from TLB contention is a generic approach, we explain it using the `msg_msg` as an example. For the `msg_msg` object, the `msgsnd` and `msgrcv` functions act as an allocation and access primitive. Figure 10.5

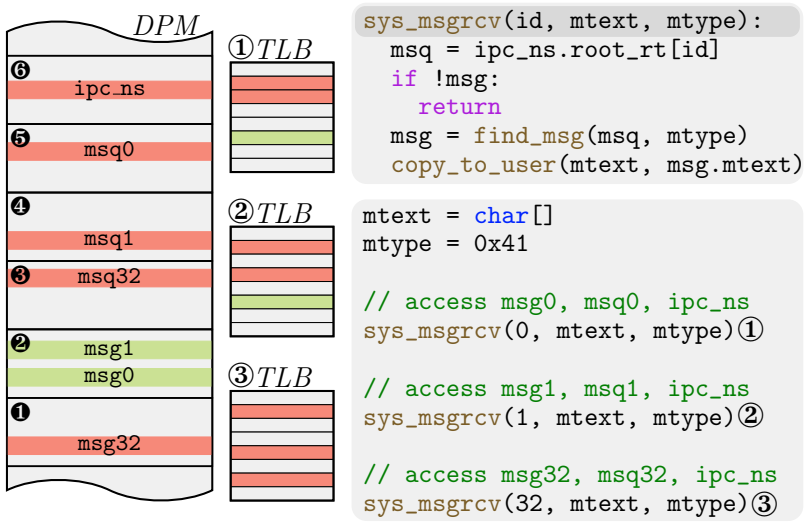


Figure 10.5: TLB contention patterns by calling `msgrcv` ①-③.

depicts the access primitive that first accesses the Inter-Process Communication (IPC) root tree `ipc_ns.root_rt` to obtain `msq` that matches `id`. It then accesses `msq` to obtain `msg_msg` matching the type `mtype` and copies the data stored within this object to user space. Executing this primitive with different `id`s creates different access patterns in the TLB. This simplified example shows a contention pattern with three 4 kB TLB entries, whereas in reality, the access primitive creates patterns with 976 entries across all TLB levels.

In Figure 10.5, we aim to leak the address of `msg0` residing on page ❷. To minimize the noise on the TLB, we exploit the caching property of the slab allocator, solving **C2**. Specifically, we make sure that the slab page of our target object (i.e., `msg0`) contains only `msg_msg`s: First, by allocating multiple `msg_msg`s, and second, by validating via the SLUBStick timing side channel [44] that only `msg_msg`s are stored on this slab page (see Section 2). We refer to the objects on page ❷ as `msg0` to `msg31`. We repeat this process so that we have a second slab page ❶ occupied with `msg_msg`s.

Next, we separate other objects, such as `msg_queue`, on different slabs via dummy allocations between two different `msq_queue` (i.e., `msq0` and `msq1`) allocations so that all other memory slots within the slab page are used for dummy objects. For example, page ❻ contains one `msg_queue` object,

and the other slots of the slab page are dummy objects. Figure 10.5 shows the memory layout of the DPM when using both approaches.

With the object layout we crafted, executing `msgrcv` with different arguments results in different TLB entries being occupied. Executing this primitive with an `id` of 0 ① results in entries 2, 5, and 6 being occupied, while running it with an `id` of 1 ② results in 2, 4, and 6 being occupied. The execution with an `id` of 32 ③ results in occupancy of entries 1, 3, and 6. Hence, we can distinguish common and differential TLB access patterns: The patterns of ① and ② are common, as their accessed objects (i.e., `msg0/1`) are on the same slab page. Pattern ① is differential to ③ as it does not access the same target slab page. We determine the common entries of ① and ②, resulting in 2 and 6. We remove the common entries with ③, resulting in entry 2, the correct page ❷. Thus, we obtained the page-aligned address of `msg0`, solving **C3.1**.

To obtain the location of all objects within the leaked and controlled slab page (solving **C3.2**), we consider the alignment enforced by the page and slab allocators [46]. The page allocator enforces that the base address of its slab is always aligned to its page order. Using the `kmalloc-cg-128` cache as an example, since this cache contains 0-order slab pages[1], their base addresses are page-aligned[2]. Objects allocated from this cache are then located at $128 \cdot n + slab\_base$ where n is between 0 and 31, and $slab\_base$ is the page-aligned address leaked by **C3.1**. Since we have occupied the entire slab page with `msg_msg`s, we obtain the location of all 32 objects within the page. While we now have all locations within the slab page, we do not know which object is at which specific leaked location but we do not need to as all objects are adversary controlled. We show in Section 7, this is sufficient for escalating privileges without crashes and nearly 100 % reliability.

Besides `msg_msg`, we perform a similar approach with allocator massaging and common/differential access patterns for other objects. We leaked the locations of `pipe_buffer`, `cred`, `file`, `seq_file`, and page-tables, i.e., Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table (PT). Table 10.3 shows their allocation and access primitives. For heap objects, we achieve the common/differential access patterns by allocator massaging, while for the page tables, we place the page-table levels in our favor (see Section 12.2).

[1] n-order slab pages refer to a slab of size $2^n \cdot$ `PAGE_SIZE`.

[2] `/sys/kernel/slab/kmalloc-*` contains these details, which remain consistent across the same kernel version.

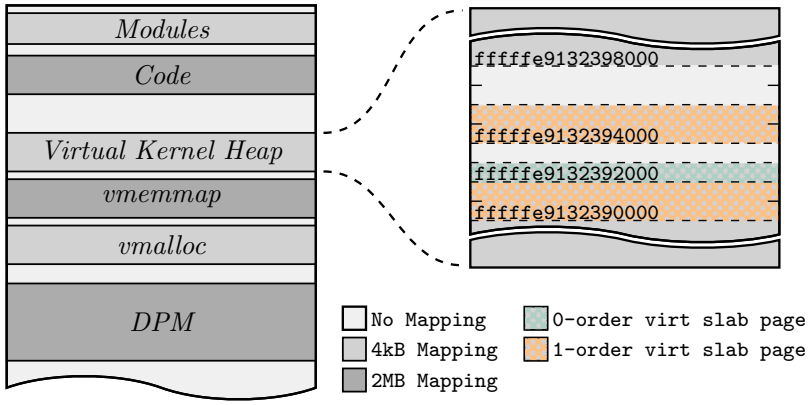


Figure 10.6: Memory layout when using `CONFIG_SLAB_VIRTUAL`, which now has a virtual kernel heap. This virtual heap contains the virtual slabs where the heap objects are located.

Takeaway 10.2

While **D1** prevents tampering with module code, it creates exploitable TLB contention patterns and, thus, allows location leakage of heap objects and page tables.

5.2. D2: Virtual Memory for Kernel Heap

The Linux kernel has enhanced heap defenses that separate objects into different sets of allocator caches based on their security context. This separation prevents in-cache reuse, where a victim object's memory slot is directly reused for security-critical objects. While this separation makes vulnerability exploitation more difficult, cross-cache reuse [68] has been proposed to circumvent this separation. It exploits the memory reuse of the page allocator and has received considerable attention from academia [16, 35, 43, 44, 68] and industry [33, 49, 62]. To counter this development, security experts presented the defense `CONFIG_SLAB_VIRTUAL` [49] and deployed it in Google's hardened system for KernelCTF [12]. While this defense provides significant value in mitigating cross-cache reuse, we show that it creates TLB contention patterns that enable leaking the location of target heap objects.

Protection Scheme. `CONFIG_SLAB_VIRTUAL` [49] deterministically prevents the reclaiming of heap memory that has been returned to the page

allocator. They achieve this by using a virtual mapped area as heap instead of the DPM, which is mapped with 4 kB pages. We refer to this area as the virtual kernel heap, as illustrated in Figure 10.6. Since the slab pages are now in the virtual heap, we refer to them as virtual slab pages. A unique feature of this defense is that it continuously increases the heap and never returns memory used by virtual slab pages back to the page allocator. This prevents the corresponding memory chunk from being reused in different slab caches, which deterministically prevents cross-cache attacks.

Leaking the Object's Location. Since this defense uses 4 kB mapped virtual memory areas, applying this defense causes accesses to heap objects (except for DMA-related memory [49]) to occupy 4 kB TLB entries, satisfying **C1** by design. However, accessing nearly the entire heap via 4 kB mappings also introduces TLB noise since the previously used 2 MB mappings are now all 4 kB. To compensate for the higher TLB noise floor, we use the same approach as in Section 5.1 to create common and differential patterns in the TLB, but with more varying arguments. From these patterns, we deduce the target kernel page, solving **C2** and **C3**.

Takeaway 10.3

While **D2** provides significant value in mitigating cross-cache attacks, it enables leaking heap object locations.

5.3. D3: Virtual Memory for Kernel Stack

The `CONFIG_VMAP_STACK` kernel defense [37] uses a virtual stack with guard pages instead of physically-mapped kernel stacks, preventing kernel stack overflows. However, we show that it also allows the location of kernel stacks to be leaked.

Protection Scheme. With this defense disabled, Linux uses the DPM directly for kernel stacks. Since the DPM is (mostly) continuous, a stack overflow would corrupt pages adjacent to the kernel stack, which may cause a difficult-to-diagnose corruption. To detect these overflows, `CONFIG_VMAP_STACK` allocates the stack with `vmalloc` and includes virtual guard pages. Now, the stack is located within the virtual memory area (see Figure 10.1), mapped with 4 kB pages.

Leaking the Object's Location. With this defense enabled, the thread's stack is accessed with memory mapped via 4 kB pages, solving **C1** by design. To exploit this defense, we need a syscall that accesses

the kernel stack with minimal other objects. We use an invalid syscall (i.e., syscall number -1) as an access primitive that only accesses the kernel stack and a few other kernel objects, such as `current`. To reduce the TLB noise and solve **C2**, we consider the stack's alignment, including the used guard pages. To solve **C3.1**, we repeatedly call the invalid syscall and determine the 4 kB TLB entry that occupies the page-aligned kernel stack. Lastly, since even with the kernel defense `CONFIG_RANDOMIZE_KSTACK_OFFSET_DEFAULT` enabled, the invalid syscall only accesses the top page from the kernel stack, the leaked TLB entry is the current kernel stack, satisfying **C3.2**.

Takeaway 10.4

While **D3** comfortably detects kernel stack overflows, it allows the location of the kernel stack to be leaked.

6. Location Disclosure Attacks through Defense Side-Channel Leakages

In this section, we first describe the side-channel primitives to leak the TLB contention patterns via an Evict+Reload TLB side-channel attack (see Section 6.1). We then evaluate the location disclosure attack using Evict+Reload and leak most security-critical kernel objects (see Section 6.2).

Evaluation Setup. We evaluate a wide range of Intel CPUs (i.e., Kaby, Coffee, Alder, Raptor, and Meteor Lake), which are all vulnerable to our disclosure attack. The used kernels for the **D1/3** exploit are the generic v5.15, v6.5, and v6.8 ones, while for the **D2** exploit, we applied the `CONFIG_SLAB_VIRTUAL` patch to its intended v6.6 kernel (see Table 10.2). We extensively evaluate our attack on the Intel i7-1360 and kernel v6.8 for **D1/3** and v6.6 for **D2** to leak the location of heap objects, page tables, and thread stacks (see Table 10.1).

6.1. Distinguish Different Memory Mappings

The `prefetch` instruction [14] allows us to measure whether or not a page is currently in the TLB by measuring its execution time. As it does not architecturally access the data, this does not cause an access violation,

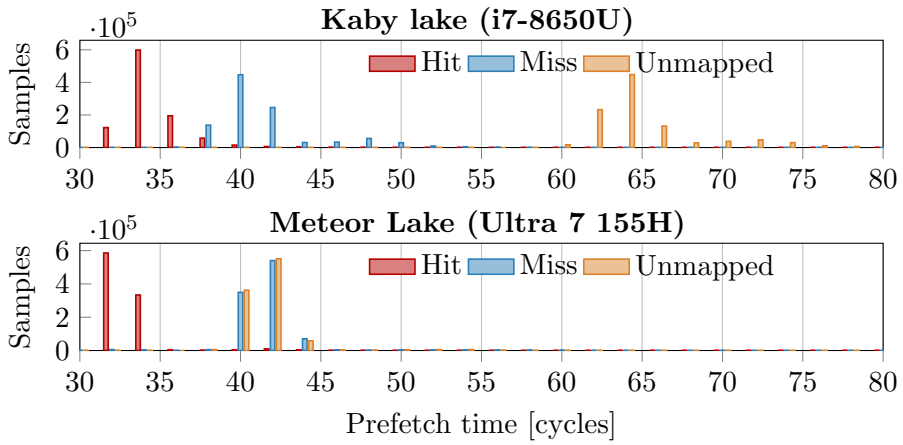


Figure 10.7: Prefetch timings of 4 kB pages.

even if performed on kernel pages from user space. We implement a fast, set-targeted TLB eviction for both TLB levels to repeatedly sample pages. We base this on the reverse-engineering of TLB addressing functions done by prior work [13, 60, 71]. While the overall TLB structure on Intel CPUs has changed significantly since the Coffee Lake architecture evaluated in prior work, we find that for 4 kB pages, the appropriate indexing functions still perform well for set eviction on Ice Lake and later generations. Combining `prefetch` and TLB set eviction, we get an equivalent attack to Evict+Reload [15] for pages.

For our attack, we need three distinguishing primitives: First, we need to distinguish TLB hits from misses on mapped pages. This allows us to locate specific kernel objects for which we have an access primitive, e.g., `msgrcv` for `msg_msg`. Second, distinguishing mapped from unmapped pages allows us to find coarse-grained memory areas, e.g., DPM or `vmemmap` (see Figure 10.1). Third, our novel primitive allows us to recognize whether an address is mapped to a 2 MB or a 4 kB page, which enables us to find pages that a kernel defense has split.

Figure 10.7 shows the prefetch timings for Kaby and Meteor Lake. We find that `prefetchnta` and `prefetcht2` combined in one measurement leads to good results on all tested CPUs. We see that TLB hits are distinguishable from misses. Thus, for our *hit/miss primitive*, we evict the TLB set corresponding to the page we are testing and then measure the

timing of prefetch on that page. As shown, the distributions are separable, so few repetitions are sufficient for stable results.

On Kaby Lake, we see that mapped and unmapped pages are also clearly distinguishable. However, changes in the newer Meteor Lake microarchitecture cause TLB misses on mapped pages to no longer show a different timing from unmapped pages with our measurement setup. We, therefore, do not rely on this difference but simply repeatedly measure the prefetch timing of a page without evicting it from the TLB to get the *mapped/unmapped primitive*. If the tested page is mapped, accesses after the first one will produce a hit timing because Intel TLBs do not cache unmapped page translations.

For the *2 MB/4 kB primitive*, we combine the two prior approaches. We repeatedly evict the target TLB entry, then prefetch another address in the same 2 MB-aligned memory at least 4 kB away, and lastly, measure the prefetch timing of the target address. If our target is mapped on a 2 MB page, accessing another address on the same page will load it into the TLB, causing a hit on the measured target access.

6.2. Evaluation of Disclosure Attacks

In this section, we evaluate our location disclosure attacks by exploiting **D1-3** via the prefetch side channel and allocator massaging. We first leak the coarse-grain location where the object resides, and then leak the fine-grain location of that object, e.g., DPM with subsequent kernel heap object leak.

Leaking Coarse-Grained Kernel Sections. We leak the base of the DPM, `vmalloc`, `vmemmap`, and virtual heap for **D2** as follows: We iterate through kernel memory in 1 GB steps since the mappings are 1 GB-aligned. At each step, we use the *mapped/unmapped primitive* to determine whether the first page is mapped. If it is, we have found the region's base. For the DPM base (i.e., `page_offset_base` of Figure 10.1), we start at `ffff888000000000`; for `vmalloc` at the end of the identified DPM region; and for the virtual heap, we start at `ffffffe8000000000`, its lowest possible address. For `vmemmap`, we search backward from `ffffffe0000000000`. Leaking these locations requires less than 1 s.

Leaking Fine-Grained Locations. For defenses that access all kernel objects with 4 kB page mappings, we iterate in 4 kB steps over the entire

Table 10.1: Evaluation results of exploiting defenses **D1-3**, showing their Success Rate (SR), the Time (T) required, the Corrected Rate (CR), where we consider that we can repeat detect false negatives, † indicating that no stable exploit was possible, and ✻ indicating that we can reallocate it in-cache from a leaked `msg_msg` with a CR of 100 % instead.

Objects	**D1**			**D2**			**D3**		
	SR %	T s	CR %	SR %	T s	CR %	SR %	T s	CR %
`msg_msg`	78	12.3	**100**	66	0.6	**100**	-	-	-
`cred`	†	†	†	77	6.6	**98**	-	-	-
`file`	80	8.1	**100**	82	0.4	**100**	-	-	-
`seq_file`	77	6.6	**100**	93	0.4	**98**	-	-	-
`pipe_buffer`	54	15.6	**96 ✻**	51	1.0	**100**	-	-	-
PT	83	17.8	**100**	-	-	-	-	-	-
PMD	93	14.5	**100**	-	-	-	-	-	-
PUD	85	14.0	**100**	-	-	-	-	-	-
Kernel Stack	-	-	-	-	-	-	98	0.3	**100**

possible memory area, i.e., virtual heap for **D2** and `vmalloc` region for **D3**. At each step, we call the access primitive for the corresponding target object and use the *hit/miss primitive* to leak the contention of the tested 4 kB TLB entry. We repeat this with different arguments for the access primitive and reconstruct the common and differential patterns to deduce the target address. For **D1**, we only perform this iteration for areas within the DPM mapped with 4 kB pages using the *2 MB/4 kB primitive*. Since most of the DPM is mapped with 2 MB pages, we can skip most addresses. On our test system more than 98 % of the DPM is typically mapped with 2 MB pages.

For the evaluation, we perform the disclosure attacks 200 times with 5 reboots in between[3]. Table 10.1 shows the evaluation results, where we leak heap objects by exploiting **D1/2**, page tables by exploiting **D1**, and kernel stacks by exploiting **D3**. The Success Rate (SR) represents the true positives divided by the total number of runs within the averaged required Time (T). The Corrected Rate (CR) considers that we can repeat detected false negatives[4]. Examples of detected false negatives are no address found

[3]For the **D1** exploit, we only insert modules in the first run after boot.

[4] $\frac{\frac{TP}{TP+FP} \cdot FN + TP}{total}$, with true/false positives TP/FP, and false negatives FN.

or the contention pattern does not satisfy alignment constraints. The † denotes that exploiting **D1** to leak `cred` was not stable because we perform `cred` spraying with `fork`, which allocates numerous other objects, resulting in the credentials rarely being allocated to 4 kB mapped memory. Table 10.1 shows that most attacks have no false positives – referred to as misidentified addresses – and the attacks that do have only about 2 %. The main cause of false positives for the **D1** exploit is that its allocation creates numerous other objects, resulting in, e.g., `pipe_buffer` may not be located on 4 kB mapped memory. However, instead of leaking the `pipe_buffer` ✻, we can leak `msg_msg` and reallocate its slot in-cache with a CR of 100 %, working with all defenses in v6.9, including **D2**. The main cause of false positives for the **D2** exploit is that we could not apply the slab side channel [44] to `cred` and `seq_file`.

We also evaluated other Intel CPUs from the 8th to the most recent 14th generation with generic kernels ranging from v5.15 to v6.8. In particular, we exploited **D1** and **D3** as these defenses are integrated in these generic kernels. Table 10.2 shows the evaluation results of the stack disclosure attacks.

Stress. To test the resilience of our side channel, we test the stack disclosure attack against TLB pressure. We start `stress -m <nr_cpus-1> --vm-keep` to create memory stress with TLB pressure on all other cores, including the exploit's sibling hyperthread. Performance counters show that it causes around 60×10^6 dTLB misses per second, while the exploit causes around 4×10^6 misses, representing considerable stress. On our Raptor Lake, stressing the system causes the CR to drop to around 93 %. By monitoring the CPU frequency, we find this is caused almost entirely by the fluctuating frequency resulting from adaptive power management on the laptop CPU in line with prior findings [46]. When fixing the frequency, this TLB pressure has a negligible effect on the CR.

7. Reliable and Stable Kernel Exploitation

In this section, we discuss that by enabling defenses such as `CONFIG_SLAB_VIRTUAL` or `CONFIG_KMALLOC_CG`, kernel exploitation by pure vulnerabilities has been made much more difficult. In particular, we discuss that the inclusions prevent or severely limit the use of existing exploit techniques.

We then show that with our location disclosure attack, which is counterintuitively possible due to some defenses, we re-enable the prevented exploit techniques or enable a new one.

In the following, we present three exploit techniques as case studies. These techniques exploit write primitives to perform privilege escalation with an exceptional reliability of more than 99.99 % on real hardware. First, we exploit the unlink primitive (see Section 7.1), which has been used by several real-world exploits [43, 52, 54, 55, 57, 59] but has also been largely mitigated by modern defenses. Second, we exploit the more generic UAF and OOB write (see Section 7.2), which is considered a weak exploitation primitive [44]. While it typically requires complex primitive conversions for privilege escalation, we present a stable and reliable exploit technique. Third, we exploit a constrained write primitive (see Section 7.3) where no read primitive is available. Prior work requires either complex primitive conversions [9, 17, 23, 26] or re-triggering the same [50] or another vulnerability [24, 52], all of which come with the risk of crashes. For this primitive, we present a novel exploit technique for privilege escalation.

Setup. We implement Proof-Of-Concepts (POCs) for the following exploit techniques to escalate privileges. We also implement a helper kernel module that provides the initial primitive. We evaluate them on two configurations, Ubuntu 24.04 with the generic kernel v6.8 (i.e., **D1** and **D3** enabled) and the kernel v6.6 with the **D2** enabled (i.e., all three enabled). We run Ubuntu on real hardware, i.e., Intel 13th Gen i7-1360P and 32 GB of RAM. To show the reliability of our techniques, we repeat their execution 1 000 times. A run is considered successful if we achieve privilege escalation. If a run results in a system crash, we explicitly mark it as a system crash. We repeat the 1 000 executions for 10 reboots, also demonstrating reliability between different reboots.

7.1. Unlink Primitive

In this case study, we discuss the problem statement of exploiting the unlink primitive. We discuss that modern defenses either prevent or significantly increase the difficulty of its exploitation. We then show that with our exploit technique (exploiting **D1/2**), we obtain an arbitrary physical read/write, allowing us to escalate privileges.

To exploit the unlink primitive, a bad actor typically exploits a UAF to gain overwrite capability on an object with a linked-list member. They

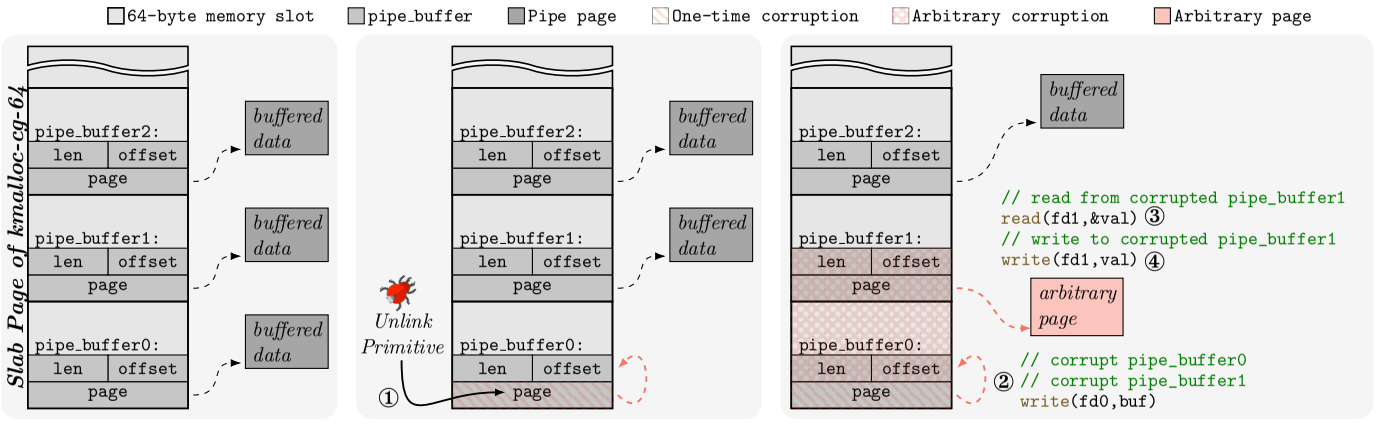


Figure 10.8: The exploit technique uses the unlink primitive and the known location of this slab page to obtain an arbitrary r/w. Initially, the unlink primitive ① corrupts the pipe buffer, i.e., `pipe_buffer0.page`, to point to its physical page. Writing with `fd0` ② corrupts the adjacent `pipe_buffer1`, whereby reading from ③ or writing to ④ (via `fd1`) enables the arbitrary r/w.

then trigger the overwrite to corrupt the linked-list member. When the corrupted object is unlinked, the following 2 writes are performed instead of unlinking the element: `*(next + 8) = prev; *(prev) = next;`. Various prior exploits convert this primitive to a more stable read or write primitive, e.g., exploits [43, 52, 54, 55, 59] do this by corrupting objects, such as `msg_msg` [52], or `seq_file` [54]. They typically trigger the unlink primitive multiple times to get a prior read primitive for leaking heap pointers. However, with the `CONFIG_KMALLOC_CG` defense, these pointer leaks have become much more difficult because the used security-critical objects are separated from vulnerable objects. Therefore, bad actors would need more powerful read primitives, which are typically not present at this stage and are difficult to transform by the unlink primitive itself.

Reliable and Stable Exploit Technique. We demonstrate the exploit technique to convert the unlink primitive into an arbitrary read/write by corrupting security-critical objects. While this is a generic technique usable with objects such as pipe_buffer [36], `file` [24, 56], or `seq_file` [54, 59], we detail it using the `pipe_buffer` as an example.

Figure 10.8 illustrates the high-level technique of exploiting `pipe_buffer`s, which we detailed in Section 12.1. Here, we require that the slab page's address is leaked and that all memory slots on this page are populated with `pipe_buffer`s. We showed in Section 5 how to achieve both. The `pipe_buffer` is the kernel object created when a user calls `pipe2`, and acts as a physically-backed ring buffer. It provides operations to read data from and write data to this buffer, which contains `page`, `len`, and `offset` as members. While `pipe_buffer.page` refers to its physically-backed page used as the ring buffer, `pipe_buffer.len/offset` store the read and write end within the physical page. We initially trigger the unlink primitive ① to refer the target `pipe_buffer0.page` to the physical page it resides on. Consequently, writing to this physically-backed page via `fd0` now corrupts `pipe_buffer0/1` ②. We corrupt `pipe_buffer0` to enable arbitrary corruption of `pipe_buffer0/1`, while `pipe_buffer1` corruption allows us to read ③ from and write ④ to the controlled kernel address, i.e., `pipe_buffer1.page`. This results in the arbitrary physical read/write primitive.

We also provide an alternative technique, where we are required to leak a slab page populated with `pipe_buffer`s and a page table, i.e., PUD. Here, we first use the unlink primitive to overwrite `pipe_buffer0.page` with the leaked page table. We then convert the single page-table overwrite to a physical read/write primitive similar to SLUBStick [44].

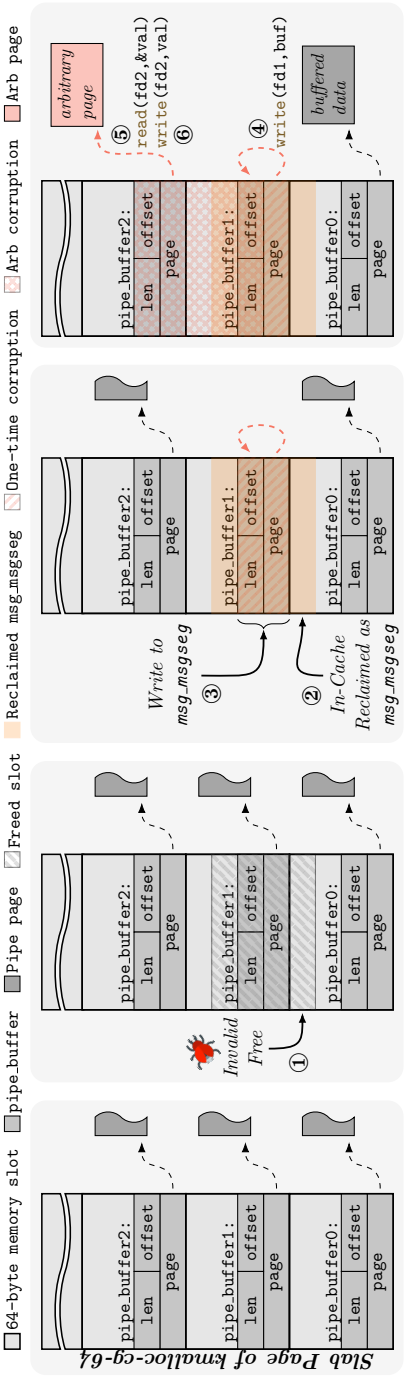


Figure 10.9: The exploit technique uses an IF and the known location of this slab page to obtain an arbitrary r/w. Initially, the IF ① frees a memory slot containing `pipe_buffer1`. In-cache reclaiming of this slot ② as a `msg_msgseg` enables corrupting `pipe_buffer1` to point to its physical page ③. Writing to with `fd1` ④ corrupts `pipe_buffer2`, while ⑤/⑥ allows arbitrary r/w.

Evaluation. We implement a helper, allowing us to trigger an unlink primitive. We then implement two POCs that leverage this primitive to manipulate `pipe_buffer` for v6.8 and a `pipe_buffer`/page table for v6.6, obtaining the arbitrary read/write. Evaluating them results in 100 % reliability.

Takeaway 10.5

While modern kernel defenses largely mitigate the unlinking primitive, our exploit technique re-enables it.

7.2. Use-After-Free & Out-Of-Bounds Write

A common technique is to convert an in-cache write primitive due to a UAF or OOB bug into a Double-Free (DF) or an Invalid-Free (IF), as they are typically more powerful. Prior research [35, 43, 44] and real-world exploits [34, 50, 51, 52, 56, 65, 66] have followed this approach. In particular, prior work [44] has shown how to convert it to a DF or IF, which, in the following, we transform to an arbitrary read/write primitive.

Reliable and Stable Exploit Technique. Figure 10.9 illustrates the exploit technique to convert an IF to privilege escalation. Again, we require that the slab page address be leaked and that all memory slots of that slab page be populated with `pipe_buffer`s. Since we know the memory slot layout of the slab page and its base address, we leverage the IF to free a slot, marking `pipe_buffer1` as free ①. Subsequently, we in-cache reclaim the invalid slot as a `msg_msgseg` object ② via the `msgsnd` syscall. This is possible because both objects are allocated for the control-group allocator cache `kmalloc-cg-*`. After the reclaiming, the `msgsnd` syscall overwrites `pipe_buffer1` with attacker-controlled data, corrupting its members to refer to the physical page it resides on ③. Writing the corrupted physically-backed page via `fd1` now allows arbitrary corruption and control of `pipe_buffer1/2` ④. Controlling `pipe_buffer2` enables to read ③ from and write ④ to the controlled kernel address, i.e., `pipe_buffer2.page`, resulting in the arbitrary physical read/write.

Evaluation. We first implement a helper that allows us to free a controlled address, mimicking the capabilities of a UAF or OOB write to a free-able heap pointer. We then implement two POCs that leverage this free to manipulate `pipe_buffer` for v6.8 and a `pipe_buffer`/page table

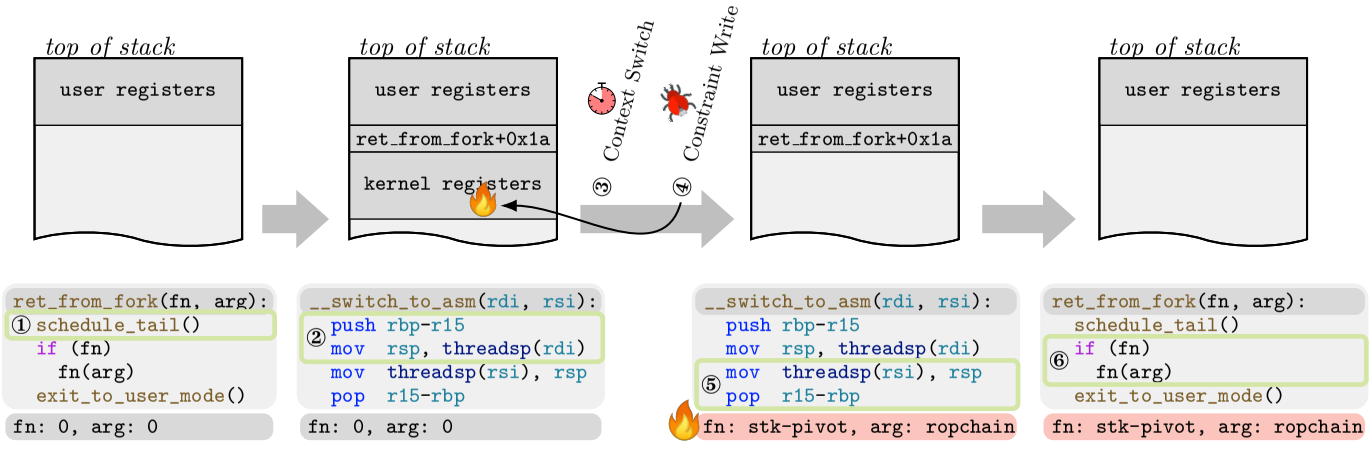


Figure 10.10: The exploit technique of register corruption for control-flow hijacking. A freshly cloned thread first executes `ret_from_fork`, which internally calls `schedule_tail` ① and saves the callee-saved registers to the kernel stack ②. It then performs a context switch ③, which takes a considerable time to continue. During this time, a bad actor uses the constrained write to corrupt the kernel registers ④. On register restoration ⑤, `fn`/`arg` are corrupted, resulting in kernel code execution ⑥.

for v6.6, obtaining the arbitrary read/write. Evaluating them results in a 100 % and 99.99 % reliability, respectively, with no crashes.

Takeaway 10.6

The UAF or OOB write is a generic exploitation primitive. We showed a compelling and reliable exploit technique to convert this primitive to an arbitrary physical r/w.

7.3. Constrained Write Primitive

Exploits that only have a constrained write with no read primitive either perform complex primitive conversions [9, 17, 23, 26] or re-trigger the same [50] or another vulnerability [24, 52] to escalate privileges, all of which have the risk of crashes. Instead, we demonstrate 3 reliable and stable exploit techniques: First, we perform an approach similar to the technique presented in Section 7.1. Second, we overwrite a leaked `cred/file` directly. Third, we perform the following novel exploit technique for privilege escalation, which builds on a control-flow hijacking primitive from prior work [47].

Reliable and Stable Exploit Technique. Figure 10.10 illustrates the technique that hijacks the control flow, whereas Section 12.1 details it. Its prerequisites are to leak the location of the kernel stack and an object that contains the ROP chain. The first is solved by exploiting the side-channel leak of **D3** (see Section 5.3). The second is solved by exploiting the leakage of **D1/2** (see Section 5.1 or 5.2) and storing the ROP chain in all controlled objects (i.e., `msg_msg.mtext`) from the leaked slab page. Other options are to leak the DPM base address, store the ROP chain on user pages, and access one via the DPM, i.e., ret2dir [28].

A thread (i.e., *T0*) initially calls `clone`, which creates a new thread (i.e., *T1*) with a leaked stack location. On first *T1* scheduling, `ret_from_fork` inherently calls `schedule_tail` ①. This schedule function initiates a context switch, saving all *T1* callee-saved registers to its stack ②. It keeps the *T1* to sleep and switches the execution context to the next thread ③. At this stage, the entire state of the *T1* is stored to memory, while it will be restored at its next scheduling. During this time window, *T0* leverages the constrained write ④ to tamper with the stack, specifically where `fn` and `arg` are located. Since the kernel does not include any randomization at this stage, these locations can be determined with our known kernel

stack location. After restoring the corrupted state of *T1* ⑤, it returns to `ret_from_fork+0x1a`, where it calls `fn(arg)` ⑥ and redirects the control flow.

Evaluation. We implement a helper for a constrained write primitive and a v6.8 POC for privilege escalation. The evaluation results in 100 % reliability.

Takeaway 10.7

While the security benefit of **D3** is low, it reliably allows a novel control-flow hijacking technique.

8. Discussion and Related Work

This section discusses the security implications of our attacks, the mitigation challenges, related work, and future work.

Security Implications. We have shown that while certain kernel defenses improve security in one dimension, they can reduce it in another. For our 3 identified exploitable defenses, we discussed that **D1/2** substantially limits the exploitation of kernel memory-corruption bugs. In contrast, **D3** provides little security benefit compared to other stack-based defenses, e.g., `CONFIG_STACKPROTECTOR` or `CONFIG_RANDOMIZE_KSTACK_OFFSET_DEFAULT`, in conjunction with the low incidence of stack-based corruption flaws. Since **D3** allows the kernel stack to be leaked and the exploit technique in Section 7.3, we argue that its security benefit doesn't outweigh its security drawback.

While we have shown the threat of our disclosure attacks, the actual impact is more severe. Consider kernel attacks such as the file UAF [56, 64], which requires a kernel heap pointer leak. With our location disclosure attacks, this heap pointer leak can be done reliably without the risk of crashing the system. The same is true for other exploit techniques.

Going one step further, in addition to the objects we leak due to kernel defense leakages, our disclosure attacks can be performed on other objects residing in 4 kB mappings. For instance, objects allocated via `vmalloc` (see Section 2) can be equally susceptible. If these objects also have appropriate allocation and access primitives, bad actors can leak their locations, making their exploitation more reliable. A notable example of such objects is the

bytecode for the extended Berkeley Packet Filter (eBPF), where eBPF is widely used for network packet filtering, profiling, and monitoring. A recent exploit technique [2] demonstrated how a limited OOB write could manipulate the eBPF bytecode and achieve privilege escalation on the latest Ubuntu systems. A critical component of this attack is heap shaping, where the bytecode must be positioned at a specific offset to align with the OOB write constraints. With our location disclosure attacks, this heap-shaping step can be stabilized and verified, increasing the reliability of this exploit technique. As a result, beyond the security-critical kernel objects discussed in this paper, our disclosure attacks can be used generically to leak the locations of kernel objects allocated in memory regions mapped with 4 kB pages, further extending the scope of potential kernel exploitation.

Mitigations. The key factors of our exploit techniques are the leverage of *location disclosure attacks*, *exploit primitives*, or *TLB side channels*. Eliminating any of these can partially or fully prevent the techniques. First, the underlying problem with *disclosure attacks* is prevented by never placing kernel objects in memory slots mapped with 4 kB pages, thus eliminating **C1**. For **D1**, one solution is to have a dedicated page allocator cache for the physical pages of kernel modules. This results in the module code never sharing the same 2 MB memory area as kernel objects. Therefore, the mapping of kernel objects cannot be changed to 4 kB using **D1**, which prevents leakage of their location. For **D2/3**, one solution is to map the memory of these allocators with 2 MB. However, while these two solutions seem appealing, they must be developed with memory performance and memory reuse in mind, and require significant engineering effort to redesign various allocators.

Second, if bad actors cannot obtain *exploit primitives*, they cannot perform the exploit techniques. Security experts continue incorporating defenses that complicate converting UAF or OOB vulnerabilities into write primitives. For instance, combining the more-fined separation of heap objects – e.g., per call-site [6], which is under discussion, or user controllable [4], which will be included – with the cross-cache mitigation [49] complicates obtaining write primitives from UAF vulnerabilities. However, this only partially prevents the exploit techniques, as UAF [24, 59], which provides a more powerful write primitive, or OOB writes [2, 9, 40, 50], still cannot be mitigated.

Third, *TLB side channels* can be prevented either by software or hardware. Software mitigations include designing existing kernel memory manage-

ment such that information disclosure is limited or defenses like FLARE [1], preventing the distinction between mapped and unmapped pages. Another mitigation is KPTI, which comes with significant performance overhead. In hardware, starting with Sierra Forest and Lunar Lake architectures, Intel CPUs will feature Linear Address-Space Separation (LASS) [22], separating kernel and user space addresses by the MSB of the virtual address. This terminates illegal accesses before any paging-based timing differences become visible and should prevent access-based attacks like double-page fault or prefetch [14, 21], though a different distinguishing mechanism might exist and be used.

Prior Kernel Exploit Techniques. Recently, there has been a burst of novel exploit techniques. Some also presented exploits combined with side-channel leaks [26, 32, 39, 41, 42, 44]. Prior work has used the prefetch side channel [14] to leak the per-CPU entry area, either to hijack the control flow [26] or to store attacker-controlled data [12]. Liu et al. [39, 41] demonstrated a KASLR break even with KPTI enabled and a vulnerability exploit combined with a kernel base leak via the prefetch side channel. However, these works only leak coarse-grained locations, e.g., the kernel base and per-CPU entry area, while we presented location leakage of kernel heap objects, page tables, and stacks. Lee et al. [32] and Maar et al. [44] presented a side channel on the slab allocator to make heap spraying and cross-cache attacks more reliable. In addition, many other privilege escalation techniques have been proposed. These include DirtyCred [35], which exchanges low-privileged with high-privileged `cred`s for privilege escalation and Dirty PageTable [66], Dirty PageDirectory [51], and SLUBStick [44], which exploits a write primitive on a page table for an arbitrary read/write.

TLB-based Side-Channel Attacks. Prior work has used TLB side channels to break aspects of KASLR. In 2013, Hund et al. [21] used the timing difference of page faults that depend on the TLB to detect the mapping layout of the kernel and locate specific code executed by a syscall or driver. Gruss et al. [14] demonstrated that the `prefetch` instruction can be used to break code KASLR and find driver locations on Intel, and Lipp et al. [38] later extended this work to AMD. TLB reverse-engineering efforts have revealed the workings of TLBs and demonstrated that they can also be attacked in similar ways as caches [13, 31, 60, 71]. They reverse-engineer the dimensions and properties of the TLB structures on Intel CPUs and their addressing functions and tagging functionality, which can also be used to break KASLR.

Software-Only Location Leaks. While most prior work has focused on leaking code or physical KASLR [14, 21, 31, 41], Maar et al. [46] have shown how to leak kernel heap pointers via a hardware-agnostic software side channel.

Future Work. We may extend to AMD and ARM, as the TLB side channel is present [14, 38]. However, AMD has the additional complication that the TLB caches unmapped translations for `prefetch`, which would require a slightly different treatment of the mapped/unmapped distinction.

9. Conclusion

Based on a systematic analysis of 127 defenses, we showed that 3 of them create exploitable, fine-grained TLB contention patterns. By combining strategic kernel allocator massaging with these patterns, we presented location disclosure attacks that leak the locations of security-critical kernel objects. We demonstrated that our attacks enable reliable and stable exploitation of kernel vulnerabilities even on the latest Linux kernel and across a wide range of Intel CPUs and kernel versions. With an attack runtime of 0.3 s to 17.8 s and almost no false positives, we showed that our attack is highly practical. We concluded that while defenses close the door to one attack variant, e.g., vulnerability exploitation, they may open the door to another, e.g., side-channel leakage.

Acknowledgements

This research was funded in whole or in part by the Austrian Science Fund (FWF) [SFB project SPyCoDe 10.55776/F85], the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087), and the European Research Council (ERC project FSSec 101076409). Additional funding was provided by a generous gift from Intel. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

10. Ethics Considerations

We have followed standard responsible disclosure practices by disclosing our findings to the Linux kernel security team's mailing list before the submission, leaving more than 90 days until the earliest date of publication.

Our experiments were conducted locally on our own machines without involvement of third-party participants or data.

11. Open Science

All distinct experiments/exploits discussed in the paper will be published and made available as POCs for the artifact evaluation[5].

References

[1] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 362, 394).

[2] Pumpkin Chang. How I use a novel approach to exploit a limited OOB on Ubuntu at Pwn2Own Vancouver 2024. 2024. URL: https://u1f383.github.io/slides/talks/2024_POC-How_I_use_a_novel_approach_to_exploit_a_limited_OOB_on_Ubuntu_at_Pwn2Own_Vancouver_2024.pdf (p. 393).

[3] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In: CCS. 2020 (pp. 368, 371).

[4] Kees Cook. mm/slab: Introduce kmem_buckets_create and family. 2024. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b32801d1255be1da62ea8134df3ed9f3331fba12 (p. 393).

[5] Jonathan Corbet. A slab allocator (removal) update. May 2023. URL: https://lwn.net/Articles/932201/ (p. 365).

[6] Jonathan Corbet. Per-call-site slab caches for heap-spraying protection. 2024. URL: https://lwn.net/Articles/986174/ (p. 393).

[5] https://zenodo.org/records/14736361

[7] Jonathan Corbet. Solutions for direct-map fragmentation. May 2022. URL: https://lwn.net/Articles/894557/ (p. 369).

[8] Jonathan Corbet. The proper time to split struct page. 2023. URL: https://lwn.net/Articles/937839 (p. 404).

[9] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel. 2022. URL: https://syst3mfailure.io/corjail/ (pp. 368, 385, 391, 393).

[10] Jake Edge. Control-flow integrity for the kernel. 2020. URL: https://lwn.net/Articles/810077/ (pp. 363, 368, 372, 375).

[11] Jake Edge. Kernel address space layout randomization. 2013. URL: https://lwn.net/Articles/569635/ (pp. 362, 366).

[12] Google. kernelCTF rules. 2023. URL: https://google.github.io/security-research/kernelctf/rules.html (pp. 362, 368, 369, 378, 394).

[13] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security. 2018 (pp. 381, 394).

[14] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 362, 380, 394, 395).

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (p. 381).

[16] Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing. Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation. In: USENIX Security. 2024 (pp. 368, 369, 375, 378).

[17] h0mbre. Escaping the Google kCTF Container with a Data-Only Exploit. 2023. URL: https://h0mbre.github.io/kCTF_Data_Only_Exploit%5C# (pp. 385, 391).

[18] Hongli Han, Rong Jian, Xiaodong Wang, and Peng Zhou. Typhoon Mangkhut: One-click Remote Universal Root Formed with Two Vulnerabilities. 2021. URL: https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Typhoon-Mangkhut-One-Click-Remote-Universal-Root-Formed-With-Two-Vulnerabilities.pdf (p. 368).

[19] Jann Horn. How a simple Linux kernel memory corruption bug can lead to complete system compromise. 2021. URL: https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html (p. 366).

[20] Jann Horn. Mitigations are attack surface, too. 2020. URL: https://googleprojectzero.blogspot.com/2020/02/mitigations-are-attack-surface-too.html (p. 371).

[21] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (pp. 362, 394, 395).

[22] Intel. Intel Architecture Instruction Set Extensions Programming Reference. 2024 (p. 394).

[23] javierprtd. No CVE for this bug which has never been in the official kernel. 2023. URL: https://soez.github.io/posts/no-cve-for-this.-It-has-never-been-in-the-official-kernel/ (pp. 368, 385, 391).

[24] Seth Jenkins. Analyzing a Modern In-the-wild Android Exploit. 2023. URL: https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html (pp. 385, 387, 391, 393).

[25] Seth Jenkins. Driving forward in Android drivers. 2024. URL: https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html (p. 371).

[26] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack. 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html (pp. 362, 368, 385, 391, 394).

[27] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel. 2021. URL: https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf (p. 368).

[28] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security. 2014 (p. 391).

[29] Imran Khan. Linux SLUB Allocator Internals and Debugging. 2022. URL: https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-1 (p. 365).

[30] Kenneth C Knowlton. A fast storage allocator. In: Communications of the ACM (1965) (p. 365).

[31] 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 (pp. 362, 394, 395).

[32] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In: USENIX Security. 2023 (p. 394).

[33] Zhenpeng Lin. How AUTOSLAB Changes the Memory Unsafety Game. 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game (pp. 363, 368, 369, 378).

[34] Zhenpeng Lin, Yueqi Chen, Xinyu Xing, and Kang Li. Your Trash Kernel Bug, My Precious 0-day. 2021. URL: https://www.blackhat.com/eu-21/briefings/schedule%5C#your-trash-kernel-bug-my-precious--day-24849 (p. 389).

[35] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In: CCS. 2022 (pp. 366, 368, 369, 378, 389, 394).

[36] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android. 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf (pp. 366, 368, 387).

[37] Linux Kernel Driver DataBase. CONFIG_VMAP_STACK: Use a virtually-mapped stack. 2024. URL: https://cateee.net/lkddb/web-lkddb/VMAP_STACK.html (p. 379).

[38] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In: USENIX Security. 2022 (pp. 362, 394, 395).

[39] William Liu. corCTF 2023 sysruption - Exploiting Sysret on Linux in 2023. 2023. URL: https://www.willsroot.io/2023/08/sysruption.html (p. 394).

[40] William Liu. CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers. 2022. URL: https://www.willsroot.io/2022/01/cve-2022-0185.html (pp. 368, 393).

[41] William Liu. EntryBleed: Breaking KASLR under KPTI with Prefetch (CVE-2022-4543). 2022. URL: https://www.willsroot.io/2022/12/entrybleed.html (pp. 362, 394, 395).

[42] EntryBleed: A Universal KASLR Bypass against KPTI on Linux. 2023 (pp. 362, 394).

[43] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024 (pp. 378, 385, 387, 389).

[44] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 363, 365, 366, 368, 369, 375, 376, 378, 384, 385, 387, 389, 394).

[45] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025 (p. 359).

[46] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025 (pp. 377, 384, 395).

[47] Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In: AsiaCCS. 2024 (p. 391).

[48] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (p. 365).

[49] Ingo Molnar. Re: [RFC PATCH 00/14] Prevent cross-cache attacks in the SLUB allocator. 2023. URL: https://lore.kernel.org/all/CAHKB1wLetbLZjhg1UVhA1QwZHo226BRL=Khm962JEfh0F+CVbQ@mail.gmail.com/T/ (pp. 378, 379, 393).

[50] Andy Nguyen. CVE-2021-22555: Turning x00x00 into 10000$. 2021. URL: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html (pp. 385, 389, 391, 393).

[51] Lau Notselwyn. Flipping Pages: An analysis of a new Linux vulnerability in nf_tables and hardened exploitation techniques. 2024. URL: https://pwning.tech/nftables/ (pp. 368, 389, 394).

[52] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html (pp. 368, 385, 387, 389, 391).

[53] James Randall. pipe_buffer arbitrary read write. 2022. URL: https://www.interruptlabs.co.uk/articles/pipe-buffer (p. 368).

[54] Eloi Sanfelix. A bug collision tale. 2020. URL: https://labs.bluefrostsecurity.de/files/OffensiveCon2020_bug_collision_tale.pdf (pp. 385, 387).

[55] Blue Frost Security. Exploiting CVE-2020-0041 - Part 2: Escalating to root. 2020. URL: https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/ (pp. 385, 387).

[56] Maddie Stone. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain. 2022. URL: https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html (pp. 368, 387, 389, 392).

[57] Maddie Stone. Bad Binder: Android In-The-Wild Exploit. 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html (p. 385).

[58] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022. 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html (p. 369).

[59] Zi Fan Tan, Gulshan Singh, and Eugene Rodionov. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938. 2024. URL: https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938%5C#unlink-primitive (pp. 368, 385, 387, 393).

[60] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In: USENIX Security. 2022 (pp. 381, 394).

[61] The Linux Kernel. Kernel Self-Protection. 2024. URL: https://docs.kernel.org/security/self-protection.html (pp. 362, 368, 369, 372, 375).

[62] Eduardo Vela. Making Linux Kernel Exploit Cooking Harder. 2022. URL: https://security.googleblog.com/2022/08/making-linux-kernel-exploit-cooking.html (p. 378).

[63] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects. In: USENIX Security. 2023 (p. 368).

[64] Yong Wang. Ret2page: The Art of Exploiting Use-Afer-Free Vulnerabilities in the Dedicated Cache. 2022. URL: https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf (pp. 368, 392).

[65] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way! 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf (pp. 363, 368, 375, 389).

[66] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html (pp. 366, 368, 389, 394).

[67] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: USENIX Security. 2019 (pp. 368, 404).

[68] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (pp. 363, 368, 378).

[69] Ptr Yudai. Understanding Dirty Pagetable - m0leCon Finals 2023 CTF Writeup. 2023. URL: https://ptr-yudai.hatenablog.com/entry/2023/12/08/093606 (p. 368).

[70] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In: CCS. 2023 (pp. 368, 369, 404, 405).

[71] Weixi Zhu. Exploring Superpage Promotion Policies for Efficient Address Translation. MA thesis. 2019 (pp. 381, 394).

[72] Xiaochen Zou and Zhiyun Qian. Exploit esp6 modules in Linux kernel. 2022. URL: https://etenal.me/archives/1825 (pp. 366, 368).

Table 10.2: Evaluation results on various CPUs and kernel versions of the **D3** exploit to leak the location of kernel stacks.

CPU Model	Architecture	Kernel	SR	T	CR
i7-8650U	Kaby Lake (8th Gen)	v6.8	100	0.2	100
i9-9900K	Coffee Lake (9th Gen)	v5.15	97	1.4	100
i7-1260P	Alder Lake (12th Gen)	v6.5	92	0.3	99
		v6.8	97	0.3	100
i7-1270P	Alder Lake (12th Gen)	v5.15	99	0.4	100
i7-1360P	Raptor Lake (13th Gen)	v6.8	98	0.3	100
Ultra 7 155H	Meteor Lake (14th Gen)	v6.8	96	0.2	97

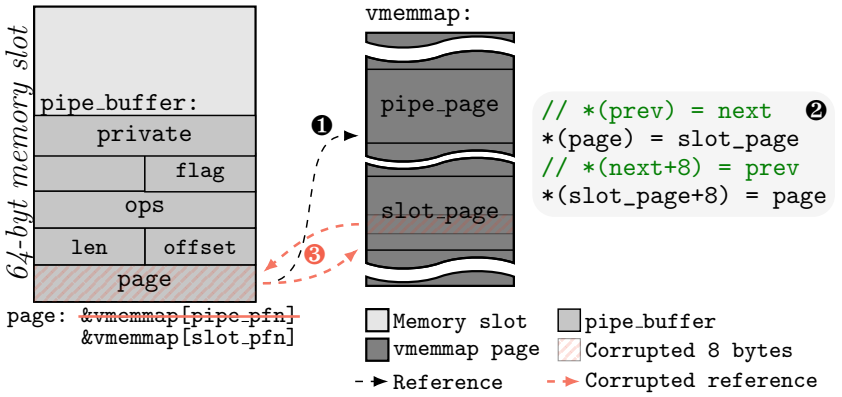


Figure 10.11: The detailed exploit technique of using the unlink primitive to allow arbitrary overwriting of a page containing `pipe_buffer`.

12. Appendix

12.1. Detailed Exploitation

This section details the reliable and stable exploitation.

Unlink Primitive. As shown in Figure 10.11, the `pipe_buffer` stores the physical-backed page via a `page` reference of the `vmemmap` region. Before corruption, the `pipe_buffer` refers to the `pipe_page` ❶ that stores the user data. On triggering the unlink primitive ❷, `*(pipe_buffer.page)` is overwritten with `slot_page`, while `*(slot_page+8)` is overwritten with `pipe_buffer.page`. The first overwrite is willing by the bad actor to refer the `pipe_buffer` with the physical page it resides on ❸, while the

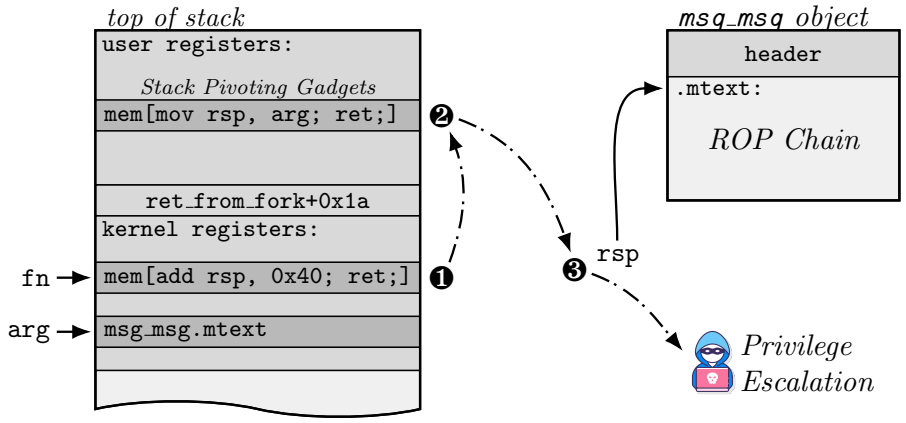


Figure 10.12: Technique to turn a write primitive into privilege escalation. First, a bad actor tampers ❶ with of `fn` and `arg` located on the stack. The call to `fn` adjusts the `rsp` to point to stack pivoting gadgets ❷ passed to the kernel via user registers. Pivoting then overwrites the `rsp` with `msg_msg.mtext` where the ROP chain for privilege escalation is located ❸.

second overwrite is an unwilling artifact of the unlink primitive. However, unwilling writing does not affect any functionality as it corrupts currently unused data [8]. As a result of this unlink primitive triggering, writing to the pipe via its file descriptor now corrupts this `pipe_buffer` and all adjacent ones.

In addition to the leaked slab page, we need to leak `anon_pipe_buf_ops` (stored in `ops`) and `vmemmap[slot_pfn/pipe_pfn]`. Obtaining `anon_pipe_buf_ops` is straightforward, as we can use the TLB side channel to leak the kernel base address and increment the `ANON_PIPE_BUF_OPS_OFFSET`, obtained by the kernel binary. Obtaining `vmemmap[pfn]` works as follows: First, we leak `vmemmap_base` via the TLB side channel. Second, since the `vmemmap` is indexed by the physical frame number, we can reconstruct the virtual address with the leaked `vmemmap_base` and the physical address of `slot_page`.

Constrained Write Primitive. Since a gadget that performs stack pivoting within one instruction sequence is hard to find in the Linux kernel [67, 70], we leverage user space data already present on the kernel stack as a first stage of the ROP attack. As depicted in Figure 10.12, we prepare user registers, which are then spilled to the kernel stack of *T0* by the syscall, i.e., `clone`. These user registers contain the location of ROP

gadgets to perform stack pivoting within multiple instruction sequences and will be copied to the kernel stack of *T1* during cloning. When *T1* is put to sleep by the context switch, *T0* overwrites `fn` with the location of a gadget that increases `rsp` to reference the stack pivoting gadgets, i.e., `mem[add rsp, 0x40; ret;]` ❶. *T0* also overwrites `arg` with the ROP chain location, i.e., `msg_msg.mtext`. When the control flow is redirected to the stack pivoting gadgets ❷, they overwrite the stack pointer with `msg_msg.mtext` and initiate the ROP chain to perform privilege escalation ❸. While we performed an ROP attack in this example, JOP would be another possibility. Zeng et al. [70] have demonstrated a systematic analysis of the use of user registers to initiate control-flow hijacking attacks.

12.2. Page-Table Contention Patterns

Figure 10.13 illustrates the workflow of leaking the location of the page table ❸. We first allocate three pages with fixed virtual addresses, whose address translation is as follows: The `addr0` uses page tables ❶→❷→❸→❹ to refer to its page, the `addr1` uses page tables ❶→❷→❸→❺ to refer to its page, and the `addr2` uses page tables ❶→❷→❻→❼ to refer to its page. Executing the `mprotect` syscall performs a software page-table walk and loads the physical addresses (accessible via the DPM) of the page tables into the TLB, while also accessing other kernel objects, e.g., `vma_struct`. Similar to the approach in Section 5.1, we call `mprotect` with different addresses to create common and differential patterns. From these patterns, we deduce the physical address of the page table ❸. In particular, calling `mprotect` with `addr0` ① creates a common pattern with `addr1` ②, as both use page table ❸, resulting in a pattern of ❶, ❷, and ❸. Conversely, calling `mprotect` with `addr2` ③ creates a differential pattern to `addr0` of ❶ and ❷. Eliminating this pattern from the common one results in the correct derived page table ❸.

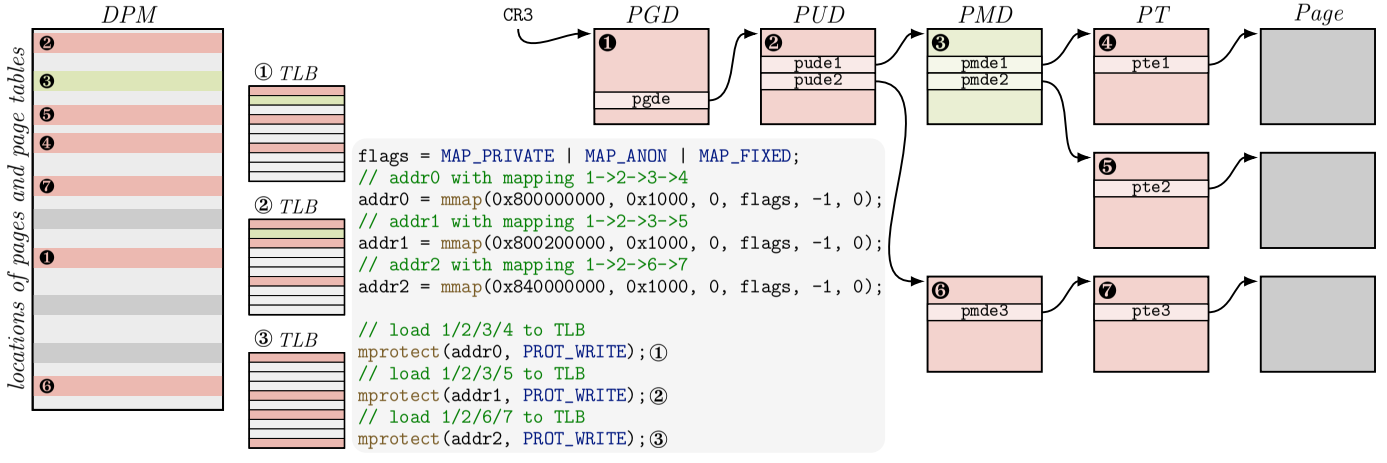


Figure 10.13: By strategically allocating user memory, we get an address translation for `addr0` with ❶/❷/❸/❹, `addr1` with ❶/❷/❸/❺, and `addr2` with ❶/❷/❻/❼. Executing `mprotect` performs a software page-table walk (i.e., ①, ②, and ③) that creates contention patterns of the page table's physical addresses, allowing to leak the address of *PT*.

Table 10.3: Allocation and access primitives for allocating and accessing kernel objects.

Kernel Objects	Allocation Primitive	Access Primitive

msg_msg	`int qid = msgget(IPC_PRIVATE, 0666 | IPC_CREAT);`
	`struct msg message = {.mtype = MSG_TYPE};`
	`msgsnd(qid, &message, MSG_SIZE, 0);`
		`struct msg message = {.mtype = MSG_TYPE};`
		`msgrcv(qid, &message, MSG_SIZE, 0, MSG_COPY|IPC_NOWAIT);`

cred	`unshare(CLONE_NEWUSER);` `open("/proc/$PID/ns/user");`	`getuid();`
file	`int fd = open("/path/to/file");`	`struct stat buf;` `fstat(fd, &buf);`
seq_file	`int fd = open("/proc/self/stat");`	`lseek(fd, 0, SEEK_SET);`

pipe_buffer	`int pipe[2];`
	`pipe2(pipe, O_NONBLOCK);`
	`fcntl(pipe[0], F_SETPIPE_SZ, 8192);`
	`char buffer[0x1000] = {0};`
	`write(pipe[1], buffer, 8);`
		`read(pipe[0], (void *)0xdeadbeef000, 8);`

PUD, PMD, and PT	`void *addr = mmap(ADDR, PAGE_SIZE, 0,`
	`  MAP_PRIVATE | MAP_ANON | MAP_FIXED, -1, 0);`
		alternating between:
		`mprotect(addr, PROT_WRITE);`
		`mprotect(addr, PROT_READ);`

Kernel stack	`clone();`	`syscall(-1);`

11

The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities

Publication Data

Lukas Maar, Florian Draschbacher, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard. The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities. In: USENIX Security. 2025

Contributions

The author of this thesis is the main author of this work. The author's contributions are *comprehensive kernel attack surface analysis*, *reconstruction of kernel vulnerabilities*, and *insights into vulnerability trends*. From the *n-day patch inclusion analysis*, the author's contributions mainly performed the semi manual detection as well as the PoC development/adaptation/-analysis. Finally, the author's contributions are also most of the written text.

The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities

Lukas Maar[1] Florian Draschbacher[1,2] Lorenz Schumm[1]
Ernesto Martínez García[1] Stefan Mangard[1]

[1] Graz University of Technology [2] A-SIT Austria

Abstract

Android's security landscape is constantly evolving to counter increasingly sophisticated attacks, with the kernel as a prime focus. Past device compromises required complex exploit chains pivoting to privileged contexts before targeting the kernel. Recently, however, the trend has been to exploit kernel GPU drivers accessible to untrusted apps to bypass privileged pivoting. While significant efforts have been made to secure GPU drivers, the broader risks of untrusted apps compromising Android devices remain underexplored at a large scale.

In this paper, we perform the first comprehensive analysis of kernel drivers accessible to untrusted apps on a representative set of 131 Android devices. Using our mostly automated approach to recover access control policies from device firmwares, we identify a significant attack surface beyond GPUs, comprising 11 drivers. From public information about these drivers, such as git repositories, we reconstruct 50 known vulnerabilities, including highly critical issues that allow exploit primitives such as use-after-free and out-of-bounds writes. Our subsequent vulnerability patch inclusion analysis reveals that many of these vulnerabilities remain unpatched, acting as n-days at the time of analysis[1] or for extended periods: More than 59 % of the analyzed devices can be exploited by highly critical n-day vulnerabilities.

We uncover novel insights into the disparity in patch timelines and vendor practices. Our findings show that malicious actors can exploit n-day vulnerabilities accessible to untrusted apps, bypassing the need for complex

[1] December 2024: time of analysis.

zero-day vulnerabilities. We conclude that urgent action must be taken to improve overall Android security.

1. Introduction

In today's interconnected world, mobile phones are essential to daily life, with Android powering billions of devices. Ensuring their security is critical, as compromises can allow surveillance, expose sensitive data, or enable identity theft. With increasingly sophisticated attacks, timely vulnerability identification and mitigation are becoming more critical.

Past full Android device compromises required complex exploit chains, as Android tightly restricts the kernel attack surface exposed to most apps. Consequently, these exploit chains often pivoted through elevated processes before targeting the kernel. For example, a 2022 attack detailed by Google Project Zero [26] began with remote code execution in Chrome. Like most apps, Chrome runs in an untrusted security context with only a limited kernel attack surface. The attack exploited a system service vulnerability to gain system privileges, increasing the reachable kernel attack surface. Finally, with system privileges, it targeted a kernel sound driver vulnerability for an arbitrary read/write, enabling device compromise. Such full chains are commonly used to secretly install spyware—like Pegasus [25] or the newly discovered NoviSpy [22]—which is eventually used for surveillance [77].

However, recent kernel attacks bypass the need for complex chains. Experts [8, 14, 16, 17, 18, 56, 58, 59, 75, 76, 77, 85, 88] highlighted that GPU drivers are directly accessible from untrusted contexts. Hence, full-chain exploits are now targeting the GPU directly from untrusted apps, eliminating the need for privilege pivoting. This strategy shift has made GPU drivers prime targets. With four GPU suppliers covering the entire Android market—ARM Mali, Qualcomm Adreno, Samsung Xclipse, and Imagination Technologies PowerVR—a single vulnerability in any of the drivers can impact a wide range of devices. Additionally, the inherent complexity of GPU drivers made them particularly susceptible to vulnerabilities.

Google's 2023 annual review [77] underscored this, attributing 4 of 5 device compromises to GPU driver vulnerabilities, with only 1 involving the core Linux kernel. In response, Google has prioritized GPU security [88],

collaborating with the Android Red Team and ARM to enhance GPU vulnerability detection, mitigation, and hardening. This raises a critical question: *Are GPUs the sole attractive target for malicious actors, or do other kernel components pose similar or even greater risks that have yet to be addressed comprehensively?*

In this paper, we comprehensively analyze the kernel attack surface accessible to untrusted apps and show that multiple kernel drivers remain vulnerable to n-day exploits, i.e., exploiting vulnerabilities that remain unpatched despite fixes being known. At the time of our analysis, 59.1 % of recent Android devices in our representative set can be exploited by highly critical n-day flaws, with 61.4 % affected by vulnerabilities of any severity. Highly critical flaws include Use-After-Free (UAF) [50, 61, 87] and Out-Of-Bounds (OOB) writes [9, 85], while moderate ones include Uninitialized Variables (UV) [11, 38, 47] and Information Disclosures (ID) [43, 44, 48]. Our findings show that these n-day driver vulnerabilities are even more attractive targets than GPU ones, as they are similarly accessible from untrusted apps and affect multiple devices but remain unpatched for extensive time. Malicious actors can, therefore, exploit these n-day vulnerabilities without needing to find zero-day vulnerabilities. By highlighting this gap, we envision improving the security maintenance of device drivers and ultimately enhancing Android security.

To achieve this, we perform three analyses: First, we analyze the kernel attack surface of drivers accessible to untrusted apps. Second, we reconstruct vulnerabilities in these drivers using public information. Third, we evaluate the vulnerability patch inclusion, which indicates the prevalence of n-days.

For the kernel attack surface analysis, we present a mostly automated approach for extracting access control data from device firmwares. This method recovers Linux's user-group-based access control and SELinux's policies, which form the fortified environment isolating security domains [2, 55]. Using this, our approach identifies kernel drivers accessible from untrusted security contexts, where most apps run. We analyze 493 firmwares from the most recent 131 devices of 7 OEM vendors (Samsung, Xiaomi, Asus, Realme, OnePlus, Oppo, and Vivo). These OEMs represent more than 75 % of Android's market share. Our results show that, apart from GPUs, 11 drivers are accessible from untrusted contexts, including components like the DSP, JPEG decoder, and AI coprocessor.

For driver vulnerability reconstruction, we collect publicly available git repositories and bug reports for 7 of the 11 drivers from two chipset ODM vendors (Qualcomm and MediaTek), covering low- to high-end devices. We then reconstruct 50 vulnerabilities from the last 4 years by searching for security-relevant keywords and manually identifying security patches.

For the patch inclusion, we semi-automatically detect the absence of patches for 21 of our 50 identified vulnerabilities. Analyzing 493 firmware images across multiple OEM vendors, we find that many vulnerabilities remain unpatched for extended periods, some exceeding one year, while others are still unpatched at analysis time. Notably, 61.4 % of devices are affected by at least one known vulnerability, with 59.1 % exposed to highly critical issues. We support n-day exploitability by triggering 5 such n-day vulnerabilities from the DSP driver on a real device, highlighting cross-OEM susceptibility.

We present 5 key findings: *First*, if a driver contains one n-day vulnerability, it is highly likely to contain more. For example, 71.4 % of our Xiaomi devices have at least one, while 49 % contain three or more. *Second*, OEMs primarily address vulnerabilities by releasing new devices rather than issuing updates for existing ones. *Third*, patch delays vary widely across OEMs, ODMs, and vulnerability types, with OOB experiencing the fastest patch inclusion, followed by UAF and ID. When comparing ODMs, MediaTek devices are more than 2 times slower to receive patches than Qualcomm devices. *Fourth*, n-day proof-of-concept exploits targeting these drivers are versatile and can be reused across OEMs. *Fifth*, the presence of n-day vulnerabilities in drivers accessible to untrusted apps enables exploitable pathways, reducing the need for time-consuming zero-day development.

In conclusion, our findings highlight the urgent need for stronger defensive measures in Android security, especially as concurrent research [22, 23, 33] reveals that malicious actors actively exploit accessible drivers in the wild.

Contributions. The main contributions of this work are:

(1) Comprehensive Kernel Attack Surface Analysis: We present the first comprehensive analysis of kernel drivers accessible to untrusted apps, identifying a broader attack surface beyond GPU drivers, comprising 11 drivers.

(2) Reconstruction of Kernel Vulnerabilities: We reconstruct 50 vulnerabilities, including high-critical issues like use-after-free and out-of-bounds writes, showing the prevalence of exploitable vulnerabilities in drivers.
(3) N-Day Patch Inclusion Analysis: We conduct a semi-automated analysis that reveals significant patch delays, with 59.1 % of devices vulnerable to high-critical n-day exploits, demonstrating persistent security gaps.
(4) Insights into Vulnerability Trends: We uncover that n-day kernel driver vulnerabilities are more attractive to malicious actors than GPU vulnerabilities and highlight disparities in patch timelines and vendor practices.

Outline. Section 2 provides background. Section 3 shows the high-level overview. Section 4 presents the kernel attack surface analysis to untrusted apps. Section 5 reconstructs vulnerabilities. Section 6 detects patches, showing multiple vulnerabilities act as n-days. Section 7 discusses security implications and related work. Section 8 concludes our work.

2. Background

This section covers the kernel exploitation terminology, Generic Kernel Image (GKI) project, Android's access control, and full-chain exploits targeting Android devices.

Kernel Exploit Terminology. We refer to exploit-specific definitions from prior work [6, 48]. A *zero-day* exploits a *zero-day vulnerability* before it is publicly disclosed or patched, while an *n-day* targets an *n-day vulnerability*, a known security issue with existing patches or mitigations that may not yet be applied. A *full exploit chain* consists of multiple stages that typically exploit a messenger [24, 62] or browser [26] and progress to the kernel with intermediate stages, ultimately compromising the device. *Exploit primitives* are basic capabilities obtained through vulnerability exploitation (e.g., out-of-bound writes), and *exploit techniques* convert primitives into more impactful outcomes (e.g., an arbitrary read/write).

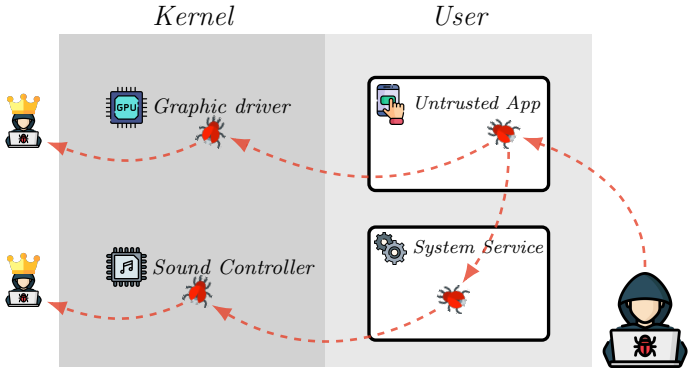
Generic Kernel Images and Kernel Drivers. The Android OS is based on the Linux kernel, which faced challenges adapting to different devices. Before the GKI project [4], OEM vendors maintained product kernels for each device model, derived from the upstream Android Linux

kernel and heavily modified. The wide range of devices resulted in a large number of product kernels and kernel fragmentation, which had negative security consequences. These include significant delays in rolling out security-critical updates—also highlighted by prior work [10, 35, 65, 90, 96]—or difficulty in merging upstream changes. To counter this trend, Google initiated the GKI project [4]. With GKI 1.0, introduced in Android version 11, devices running 5.4 kernels must pass GKI tests., i.e., from the compatibility test suite. With GKI 2.0, devices running 5.10 or later kernels must ship with the GKI kernel maintained and built by Google. GKI 2.0 has security benefits, as these kernels are updated with long-term stable changes and critical bug fixes, resolving the kernel fragmentation issue. To compensate for device customization, OEMs now rely on introducing customization via kernel modules.

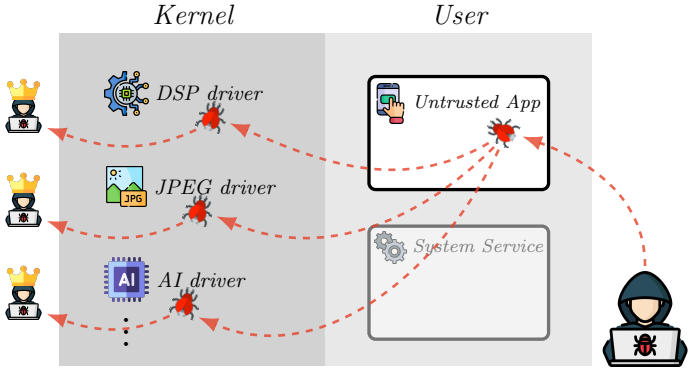
SELinux and Android's Access Permissions. Android combines Linux's user-group-based access controls with Security-Enhanced Linux (SELinux)'s mandatory policies, creating a fortified environment where different security domains are isolated [2, 55]. This reduces the risk of malicious interference and enhances the overall system security.

Linux offers Discretionary Access Control (DAC) for managing file system permissions so that users cannot alter or access other users' resources, which Android uses to isolate applications from each other. Each app runs under its own Linux user, and files created by one app cannot be accessed by other apps unless explicitly granted permission to share. For more fine-grained access control, Android establishes Mandatory Access Control (MAC) on processes with SELinux. SELinux offers this by integrating into the Linux Security Module (LSM) framework and using syscall hooks and policies to enforce access control decisions. The system follows a default-denial principle, allowing only explicitly permitted actions. It operates in permissive or enforcing mode, where, per default, Android runs in enforcing mode.

SELinux policies define rules for allowing actions by a particular object on a specific subject. The subject is commonly a set of processes that run in the same security domain, also called a security context. On Android, the untrusted security context (i.e., `untrusted_app`) is the domain assigned to third-party applications installed from the Google Play Store or other sources. It ensures that apps are restricted from performing unauthorized actions or accessing sensitive system resources, e.g., most of the kernel and its drivers. This context prevents apps from directly interacting with other applications, enforcing strict boundaries unless explicitly permitted



(a) **Prior Exploitation Chains:** Attack directly the graphics driver from an untrusted or the kernel from a system security context.



(b) **Presented Exploitation Chains:** Attack directly kernel drivers which are accessible from an untrusted security context.

Figure 11.1: Exploitation chains of attacking Android devices to get full root.

by mechanisms like Inter-Process Communication (IPC), e.g., through Android's binder IPC. In addition to `untrusted_app`, Android defines other SELinux contexts for different types of apps or system components, such as `system_app` for privileged apps. The SELinux policies for each context ensure that processes operate within predefined boundaries, minimizing security risks and maintaining system integrity.

3. High-Level Overview

This section provides an overview of our work, first highlighting how past full-chain exploits have primarily targeted Android devices. While prior attacks typically exploited well-studied vulnerabilities in GPU drivers or the kernel alongside privileged process exploitation, we analyze alternative, underexplored attack surfaces for large-scale root compromise.

Prior Exploitation Chains. Full-chain Android exploits (see Figure 11.1a) typically achieve code execution in an untrusted security context by exploiting vulnerabilities in an application [75, 76, 77], such as browsers [26] or messengers [24, 62]. With code execution, these exploits have typically followed one of two pathways to root, compromising the device: First, the attack targets a vulnerability in a higher-privileged process, allowing it to elevate from the untrusted to the higher-privileged security context, e.g., the system context. This escalation significantly increases the kernel attack surface. With higher privileges, the attack then exploits one or more kernel vulnerabilities that are accessible from this context [26, 36, 46, 70], such as the `io-uring` subsystem [46] or the higher-privileged sound device drivers [26]. Second, the attack targets one or more vulnerabilities that are accessible from the reduced kernel attack surface within the untrusted context. In this case, malicious actors mainly focus on vulnerabilities in the GPU driver. In fact, according to Google's annual report in 2023 [77], GPU drivers were targeted by 4 out of 5 full-chain exploits, with one taking the first pathway.

From an attacker's perspective, both approaches face a similar problem: Security researchers are aware of them and have made significant advances in suitable detection and mitigation. For instance, collaborative efforts by Google, the Android Red Team, and ARM have substantially enhanced the security of Android GPU drivers [88]. These improvements have largely concentrated on GPU driver vulnerabilities, leaving other kernel components less explored and vulnerable.

Presented Exploitation Chains. In contrast to the past focus on GPU drivers, we identify and analyze alternative kernel components as equally—if not more—critical exploit targets (see Figure 11.1b). We show that these components meet the following generalized criteria for exploitation:

C1: Accessibility. The target kernel component is directly accessible from untrusted security contexts.

C2: Broad Impact. A vulnerability in the target component can affect a wide range of Android devices.

C3: Susceptibility. The target is highly susceptible to including unintentionally exploitable vulnerabilities.

In Section 4, we present a mostly automatic approach to identify other components accessible from untrusted contexts. Our findings reveal that beyond GPUs, 11 device drivers, such as those for the Digital Signal Processor (DSP), JPEG decoding accelerator, and Artificial Intelligence (AI) coprocessor, meet **C1**. In Section 5, we perform a semi-automated analysis to find n-day vulnerabilities by inspecting publicly available git repositories and bug reports. We identify 50 vulnerabilities within 7 device drivers that affect a wide range of devices, satisfying **C2**. In Section 6, we reveal that many identified n-day vulnerabilities remain unpatched for an extensive amount of time or unpatched till December 2024, i.e., the date of the analysis. This leaves 61.4 % of devices vulnerable, with 59.1 % exposed to highly critical vulnerabilities. The lack of n-day patches eliminates the need for the time-consuming discovery of complex zero-days, satisfying **C3**.

Threat Model. In our threat model, we assume a malicious actor who has already achieved code execution in an untrusted security context, e.g., by exploiting an application like Chrome. The malicious actor's goal is to compromise the device's kernel and take full control of the device with minimal effort and resources. Given the time and resource intensity of discovering zero-day vulnerabilities, the malicious actor aims for alternative pathways, including the exploitation of n-day vulnerabilities. This aligns with the expectations of real-world Android exploitation [26, 54, 70, 71, 74].

Collection and Extraction of Firmwares. We automatically collect firmwares not protected by captchas and manually collect those that are protected by captchas. We implement a web crawler based on Python Selenium to download firmwares from different points in time where possible. We consider 7 OEM vendors, accounting for more than 75 % of the Android market [5]. These OEMs comprise the top 5 (Samsung, Xiaomi, Vivo, Oppo, Realme) as well as 2 well-recognized (OnePlus, Asus), and use the chipset of 3 ODM vendors (Qualcomm, MediaTek, Samsung). We consider devices of OEMs released between October 2022 and December 2024 (completion of the analysis). Our focus lies on recent devices as these are more likely to receive security updates [1, 48, 96]. Overall, we collect and extract 493 firmwares for 131 Android devices

which is a representative sample of the 488 devices produced in this time span. For Samsung and Xiaomi, our collection includes version lineages, i.e., multiple different firmware versions for the same device.

4. Attack Surface Analysis of Android Kernels

This section conducts a large-scale analysis to identify the kernel attack surface accessible for the untrusted security context. This involves generalizing Android's approach to minimizing the kernel attack surface and detailing how this information can be extracted from device firmwares (see Section 4.1). Using this approach, we analyze the kernel attack surface of 493 firmwares (see Section 4.2). We show that this surface includes multiple kernel devices, satisfying criteria **C1**. To validate these findings, we perform dynamic access tests (see Section 4.3) by implementing an application in the untrusted context to confirm access to the previously identified drivers.

4.1. Determining the Minimum Kernel Attack Surface

To determine the kernel attack surface accessible to untrusted contexts, we address two key questions: What kernel components are potential attack targets, and which are accessible to unprivileged apps? We focus on device drivers, the most vulnerable part of the kernel [7, 53]. Hence, to determine the attack surface of kernel drivers, we need the drivers themselves and each access permission.

Prior to GKI 2.0, these kernel drivers were typically included in the kernel binary. However, starting GKI 2.0, OEMs were forced to use the generic Android kernel image, implying that ODM-specific drivers are no longer included in the binary. Instead, they are loaded as kernel modules typically during boot. The storage locations of these drivers embedded within the device's firmware vary and depend on the OEM and model. After obtaining the drivers, we determine which are accessible to untrusted security contexts. To achieve this, we extract two access control data from the firmware: SELinux policies and user-group-based Linux permission settings, both defining the access to these drivers.

Figure 11.2 illustrates our high-level workflow for determining the kernel attack surface by analyzing the SELinux policies (in Section 4.1.1) and the

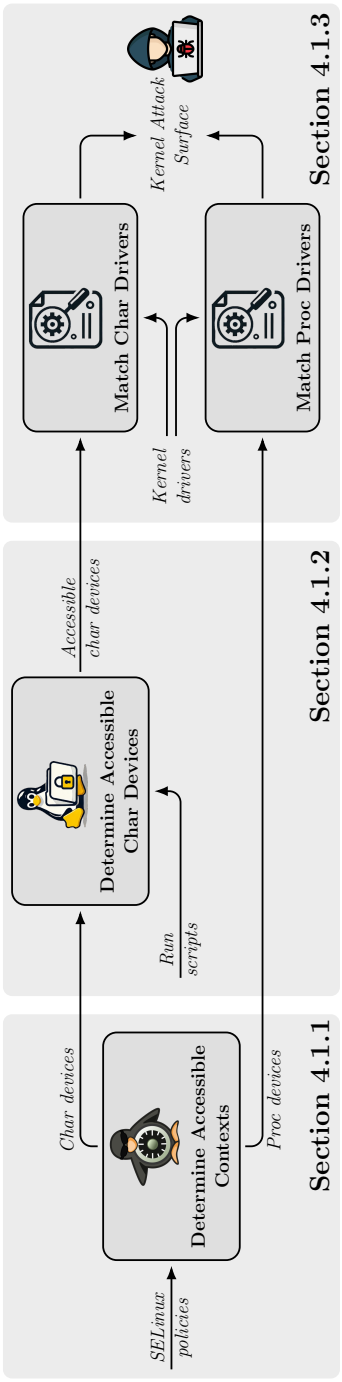


Figure 11.2: High-level workflow for determining the kernel attack surface from SELinux policies, Linux permissions and drivers.

```
1 allow appdomain vendor_qdsp_device:chr_file { ioctl read };
2 allow domain zero_device:chr_file { append getattr ioctl lock map
  -> open read watch write };
3 allow untrusted_app gpu_device:chr_file { append getattr ioctl lock
  -> map open read watch write };
4 allow untrusted_app sound_device:chr_file { append getattr ioctl
  -> lock map open read watch watch_reads write };
5 allow untrusted_app_all untrusted_app_all_devpts:chr_file { getattr
  -> ioctl open read write };
```

Listing 11.1: SELinux access control for `untrusted_app` on Xiaomi Redmi Note 14 Pro+.

Linux permission settings (in Section 4.1.2) to find the matching kernel drivers accessible by unprivileged security contexts (in Section 4.1.3).

4.1.1. Analyzing SELinux Policies

To reconstruct SELinux access control policies, two configuration files are critical [3]: The precompiled policies (i.e., `precompiled_sepolicy`), which configure allowed access of SELinux contexts to specific domains; and the domain mappings (i.e., `vendor_file_contexts`), which assign file paths to domains. Together, they allow the identifying actions of a SELinux context to be performed on a file at a given path. Depending on the device, the configurations are stored in three possible locations for `precompiled_sepolicy` and two for `vendor_file_contexts`, e.g., `/etc/selinux` in partition `odm` or `vendor`. Our approach extracts these policy-related files for subsequent analysis. We then use the official SELinux policy query tool, `sesearch`, to obtain access control rules for character devices from untrusted contexts.

Character Devices. Android's hardware resources are typically managed by kernel drivers that expose higher-level functionality to user space via character device files. These character devices usually mount virtual files in the `/dev` directory. User-space apps can interact with them via syscalls like `open` and `ioctl`. To find character devices accessible in the unprivileged `untrusted_app` context, we execute queries against `precompiled_sepolicy`. For instance: `sesearch --allow -s untrusted_app -c chr_file -p ioctl precompiled_sepolicy` finds all character devices accessible via the `ioctl` syscall. Listing 11.1 illustrates a simplified SELinux policy output by `sesearch` on the Xiaomi

```
1 /dev/kgsl                u:object_r:gpu_device:s0
2 /dev/adsprpc-smd         u:object_r:vendor_qdsp_device:s0
3 /dev/xlog                u:object_r:sound_device:s0
```

Listing 11.2: Mounting points for domains accessible to `untrusted_app` on Xiaomi Redmi Note 14 Pro+.

Redmi Note 14 Pro+. It shows domain-specific permissions, which control resource access for processes running with the `untrusted_app` context. The `appdomain` (a context despite its name) covers most Android apps (including `untrusted_app`) and, in this example, can access files in the `vendor_qdsp_device` domain. The broader `domain` (a confusingly named context that comprises all processes on the device) is granted permission on the `zero_device` domain. The `untrusted_app` context, a subset of `appdomain`, has stricter controls but full access to `gpu_device`. The more restrictive `untrusted_app_all` context allows access to `untrusted_app_all_devpts` on the specific firmware.

SELinux policy only allows us to learn about access to domains, e.g., `vendor_qdsp_device`. To resolve these domains to mounting points of a character device within the `/dev` directory, we use the `vendor_file_contexts`. Listing 11.2 illustrates the content of `vendor_file_contexts` relevant to recover the device's mounting point. For instance, the mounting point for the `vendor_qdsp_device` domain is `/dev/adsprpc` on Xiaomi Redmi Note 14 Pro+.

ProcFS Files. The Process File System (ProcFS) provides an alternative mechanism for interacting with kernel device drivers [27]. Kernel drivers can expose user-space interfaces by creating virtual files using the `proc_create` kernel function, which takes the file name and access permissions as arguments. Accessing these files through syscalls prompts its kernel driver functions, enabling communication between the user and kernel. Similar to character devices, we use `sesearch` to find the access permissions for ProcFS files. However, we do not need `vendor_file_contexts`, as their domain names already contain the path, e.g., `allow appdomain proc_ged:file {...}` refers to the path `/proc/ged`.

Pivoting Contexts. As observed in Listing 11.1, permissions for character devices vary. For instance, while `zero_device` permits mapping, `vendor_qdsp_device` does not. SELinux policies may allow certain operations (e.g., `ioctl`) on a file but restrict others, such as opening (e.g.,

```
1 allow appdomain appdomain:binder { call transfer };
2 allow appdomain appdomain:fd use;
```

Listing 11.3: Transfer from `untrusted_app` to `platform_app`.

`vendor_qdsp_device`). This limitation can be bypassed by legally pivoting to contexts with open permissions for the file [19, 33], avoiding the need for a vulnerability. To identify pivoting contexts, we query contexts that allow the target device to be opened and locate transitions from `untrusted_app` to those contexts. This allows transitioning to a context that can share device references with `untrusted_app`. For example, we identified `platform_app` and `vendor_dspservice` as pivoting contexts that enable interaction with `vendor_qdsp_device`.

Listing 11.3 illustrates these transfers. Line 1 permits `appdomain` (including `untrusted_app`) to use Android's Binder IPC for calls and data transfer between `appdomain`s, e.g., `platform_app`. Line 2 permits shared file descriptors, enabling `untrusted_app` to access resources used by `platform_app`. Thus, Binder IPC and shared file descriptors allow `untrusted_app` to open access `vendor_qdsp_device`.

4.1.2. Analyzing Linux Permission Settings

Linux executes Run Control (RC) scripts during startup to set up services and configs, including permissions. These scripts use commands like `chmod` and `chown` to define user-group-based permissions to read, write, and execute for files and devices. For character devices, the permissions are typically set based on predefined policies in scripts or config files.

We implement an approach that extracts all found RC scripts (in `/etc/` or `/etc/init`) to examine these permission settings. We mark a character device as accessible if the RC permissions permit it to unprivileged `others` users, and a SELinux policy rule allows access to unprivileged contexts. There are cases where only one is true. For instance, SELinux's access control allows the `ioctl` syscall on `/dev/sdsprpc-smd` while its permissions is `0660 system:system`, indicating that only the system user/group has read/write access to this device but no unprivileged `others` user. Another example is `/dev/elliptic`, which has `0644 system:system` per-

missions, but no SELinux access control rule allows access to it from the `untrusted_app` context.

4.1.3. Matching Kernel Drivers

Since GKI 2.0, OEMs are forced to use the Google-maintained GKI kernel and are required to move kernel drivers to external modules. There are 2 possible locations of kernel drivers, either in the `vendor_dlkm` partition or compressed inside a ramdisk stored within the `vendor_boot` partition. Android uses a ramdisk to initialize the system before mounting the main file systems. A ramdisk is a temporary file system loaded into RAM during the boot process. The driver extraction varies depending on the storage: If drivers are located within `vendor_dlkm`, we mount the partition and copy the drivers for further analysis. Extraction from the `ramdisk` requires decompressing the disk image, then extracting the drivers from the ASCII cpio archive using a tool like `binwalk`. `binwalk` extracts a file system from the cpio archive, storing the drivers in `/vendor` or `/vendor_dlkm`.

With the reconstructed driver modules and the list of accessible device nodes in the file system, we match each device node—either in `/dev` or `/proc`—to its driver. Our approach depends on the device node's mounting point. For `/dev`, we automatically match the file name (e.g., `adsprpc-smd`) with all strings contained in kernel modules. We then manually verify the mapping by comparing it with the driver's source code. For `/proc`, we automatically scan driver modules for the file name (e.g., `jpeg_driver`) and the `proc_create_file` symbol. We then manually inspect the identified kernel module and its source code. This verifies the mapping and confirms that the `mode` argument passed to `proc_create_file` renders the device node accessible to untrusted contexts.

By combining these methods, we establish a mapping of each mounting entry to its corresponding kernel driver. To ensure that these drivers are loaded at startup, we verify that the matched module appears in the `modules.load` file, which lists all kernel modules to be loaded automatically.

4.2. Large-Scale Analysis

This large-scale analysis is a fully automated process to determine all kernel drivers that are accessible to untrusted contexts. The manual

Table 11.1: Accessible kernel attack surface from the untrusted security contexts, showing the percentage of devices per OEM vendor permitting access to the corresponding kernel driver.

Category	Device Driver's Entry	Module	Accessibility per OEM Vendors						
			Samsung	Xiaomi	Asus	Realme	OnePlus	Oppo	Vivo
AI	/dev/apuext	apusys.ko	2	12		12		29	40
DSP	/dev/adsprpc-smd	frpc-adsprpc.ko	48	52	83	56	71	43	40
DSP	/dev/fastrpc-[acs]dsp	frpc-adsprpc.ko		10					
NPU	/dev/vertex	npu.ko	38						
AI	/dev/apusys	apusys.ko		4					
Audio	/dev/xlog	xlogchar.ko		60					
GPU Extention	/proc/ged	ged.ko	14	38	17	38	29	57	60
Monitor	/proc/perfmgr	mtk_perf_ioctl.ko	7	38	17	38	29	57	60
JPEG	/proc/mtk_jpeg	jpeg-driver.ko	2	25	17		14	29	40
Memory	/proc/secmem	trusted_mem.ko	12	25					
Monitor	/proc/mi_log	mi_log.ko		21					
Camera	/proc/camera	camera.ko	10						

preprocessing phase, outlined in Section 4.1, serves as an initial phase for this automated analysis, where we map each driver's entry point to its corresponding kernel module. The automated process analyzes SELinux policies, Linux permission settings, and kernel drivers across 493 firmware versions of 131 Android devices from 7 OEMs. This analysis determines drivers that untrusted contexts can access. Table 11.1 presents the results, highlighting the extent of the kernel attack surface across OEMs. These findings reveal that multiple drivers are accessible and that most remain accessible across OEMs, satisfying **C1**.

Table 11.1 categorizes the accessible kernel drivers based on their intended functionality (derived from `modinfo`) and lists their entry points and module names. For each OEM and driver module, the table indicates the percentage of devices permitting access or, if access is absent, leaves the cell blank. The most contributing reason for inaccessibility is the lack of hardware support for the corresponding software driver module. For instance, the `npu.ko` driver is found exclusively on Samsung-ODM devices, present in 38 %. Another less dominant contributing factor to variability in access is hardware support combined with device-specific access permission settings. For example, while the `apusys.ko` driver is included in devices from Xiaomi, Oppo, OnePlus, and Realme, only a subset of Xiaomi devices allows access to untrusted contexts.

Device configurations show mutually exclusive driver sets based on the ODM chipset. For example, Xiaomi devices are either Qualcomm- or MediaTek-based, leading to Qualcomm drivers such as `/dev/adsprpc-smd` or `/dev/fastrpc-[acs]dsp`, or MediaTek drivers such as `ged.ko` and `mtk_perf_ioctl.ko`. This pattern extends to other OEMs. Among Xiaomi's MediaTek devices (38 % of the lineup), about 65 % include the `jpeg-driver.ko` driver. Those findings highlight the nuanced variability in kernel driver accessibility across devices, ODMs, and OEMs.

4.3. Validity of Analysis

Our analysis relies on static interpretation of kernel drivers, SELinux policies, and RC scripts, all of which contribute to driver accessibility from unprivileged contexts. To verify the validity of our statically determined results, i.e., to confirm they reflect behavior on real devices, we perform dynamic testing on a representative subset of Android devices. We pick 15 devices from 4 OEM (Samsung, Xiaomi, Vivo, Oppo) and 3 ODM vendors,

reflecting the 4 most popular Android OEMs by market share [5]. To span the performance spectrum, we test 3 Samsung models (S24 Ultra, A55, A14), and the Xiaomi Redmi 12, Vivo Y36, and Oppo A58. Additionally, we validate driver accessibility on nine more Samsung devices via the Remote Test Lab. In total, our test set includes 5 Qualcomm-based, 6 MediaTek-based, and 4 Samsung ODM.

To determine device driver interfaces accessible from untrusted contexts at runtime, we implement an unprivileged Android application. It attempts to invoke a series of syscalls on each file in `/dev/` or `/proc/` whose accessibility we wish to determine. These syscalls are `open`, `close`, `read`, `write`, `ioctl`, `fgetxattr`, `mmap` and `flock`, representing SELinux's access permissions (see Listing 11.1). We consider a file as accessible if the syscall yields success or the resulting error code does not indicate a lack of permission. Listing the contents of `/dev/` or `/proc/` from an untrusted context, such as our test app, may be forbidden, even if access to the contained files is possible. Hence, we list directory contents as the higher-privileged shell user. We pass the obtained list of files to our unprivileged app to test the syscalls. For some subfolders of `/dev/` or `/proc/`, not even the shell user may list contents. As this only affects a small number of paths across all firmwares, we hardcoded them into the app.

For all evaluated devices, the results of our static analysis align with those obtained at runtime. Hence, we conclude that our large-scale results estimate real devices.

5. Analysis of N-Day Vulnerabilities

In this section, we present a systematic analysis of n-day vulnerabilities. These are security flaws that have been publicly disclosed and have available patches but remain unpatched on devices. While some security flaw sources—such as public vulnerability disclosures [88] and write-ups [58]—explicitly reveal the nature of the flaw, others—such as Security Bulletins from Google or Qualcomm—provide less direct information, making the identification process more complex.

A notable observation in our analysis is that most publicly available write-ups and vulnerability disclosures focus disproportionately on GPU driver vulnerabilities [18, 56, 57, 58, 59, 77, 88]. This focus has led to significant progress in understanding and addressing GPU-related security issues.

However, it has also created a gap in public knowledge about exploiting vulnerabilities in other types of drivers, e.g., other drivers accessible from untrusted contexts. To address this imbalance and ensure a broader vulnerability exploitation coverage, our approach extends beyond GPU drivers and analyzes driver-specific vulnerabilities. We achieve this by identifying vulnerabilities using a history tree search across different drivers. Through this analysis, we successfully identify 50 vulnerabilities, detailed in Table 11.2. Section 6 then shows that multiple of these vulnerabilities are either n-days at the time of analysis or for extended periods. By showing that each of these n-day vulnerabilities affects multiple devices, it meets **C2**.

5.1. N-Day Vulnerability Identification

Android adheres to a strict open-source policy to ensure transparency and encourage collaboration within the security community. This includes maintaining public access to *bug reports* and releasing *kernel modifications*. While this openness empowers security researchers to identify, analyze, and address potential vulnerabilities, it also provides malicious actors with the means to identify n-day vulnerabilities.

To identify such n-day vulnerabilities, *bug reports* can potentially serve as a direct source of information. Ideally, these reports should only be publicly available after the associated vulnerabilities have been patched in all systems, including downstream versions. However, in practice, they typically become public after exceeding the disclosure deadline or following a grace period after a patch is released. Google Project Zero, for instance, follows a 90+30 disclosure deadline policy. If a patch is released within a 90-days time period, details are disclosed 30 days later. If no patch has been released, the reports are publicly disclosed after 90 days.

Bug reports can be generalized into two categories: First, vulnerability disclosures reveal official disclosure information, which provides detailed vulnerability descriptions. Examples include CVE-2022-22706/CVE-2021-39793 [20] and issue trackers highlighting CVE-2024-23384 [93] and CVE-2024-23698 [12]. Second, exploit write-ups and analyses encompass publicly available exploit details contributed by the security research community. Examples include works by Mo [56, 57, 58, 59] and studies of zero-day and n-day vulnerabilities found in the wild, e.g., done by Google Project Zero [27]. As described above, both categories of bug reports provide direct

```
commit 2466bcf3cea4ed9b37b7e8983e7e6b7ffd92e8fc
Author: quic_anane <quic_anane@quicinc.com>
Date:   Tue Jul 16 23:37:45 2024 +0530

    msm: ADSPRPC: Avoid Out-Of-Bounds access

    Currently, when adding duplicate sessions to an array that
    holds session information, no check is performed to avoid
    going out of bounds. Add a check to confirm that the index
    is not out of bounds.

    Change-Id: Ib7abcc5347ba49a8c787ec32e8519a11085456d9
    Signed-off-by: quic_anane

diff --git a/dsp/adsprpc.c b/dsp/adsprpc.c
index d7e2c3e..631d1b3 100644
--- a/dsp/adsprpc.c
+++ b/dsp/adsprpc.c
@@ -8172,6 +8172,12 @@ static int fastrpc_cb_probe(struct device
-> *dev)
      for (j = 1; j < sharedcb_count &&
          chan->sesscount < NUM_SESSIONS; j++) {
        chan->sesscount++;
+       VERIFY(err, chan->sesscount < NUM_SESSIONS);
+       if (err) {
+         ADSPRPC_WARN("failed to add shared session, maximum
-> sessions (%d) reached \n", NUM_SESSIONS);
+         break;
+       }
        dup_sess = &chan->session[chan->sesscount];
        memcpy(dup_sess, sess,
          sizeof(struct fastrpc_session_ctx));

```

Listing 11.4: Git commit of an adsprpc out-of-bounds access.

information about vulnerabilities and, if not patched, offer the possibility of exploiting them in an n-day scenario.

Another critical source of information is the source code for any *kernel modifications*, including driver code, which must be released under the GNU General Public License version 2 (GPLv2). This obligation arises from Android's use of the GPLv2 licensed Linux kernel: Any distributed modified versions must also have their corresponding source code made publicly available under the same terms. Here, we exploit this open-source policy to identify n-day driver vulnerabilities.

```
commit 3a1e7d811168a32b10171905503d724605064238
Author: DEEPAK SANNAPAREDDY <quic_sdeeredd@quicinc.com>
Date:   Fri Sep 22 16:32:06 2023 +0530

    msm: adsprpc: Handle UAF in process shell memory

    Added flag to indicate memory used
    in process initialization. And, this memory
    would not removed in internal unmap to avoid
    UAF or double free.

    Change-Id: Ie470fe58ac334421d186feb41fa67bd24bb5efea
    Signed-off-by: DEEPAK SANNAPAREDDY

diff --git a/dsp/adsprpc.c b/dsp/adsprpc.c
index 2c28969..43648e9 100644
--- a/dsp/adsprpc.c
+++ b/dsp/adsprpc.c
@@ -4351,6 +4351,8 @@ static int
 -> fastrpc_init_create_static_process(struct fastrpc_file
      mutex_lock(&fl->map_mutex);
      err = fastrpc_mmap_create(fl, -1, NULL, 0, init->mem,
        init->memlen, ADSP_MMAP_REMOTE_HEAP_ADDR, &mem);
+     if (mem)
+       mem->is_filemap = true;
      mutex_unlock(&fl->map_mutex);
      if (err || (!mem))
        goto bail;

```

Listing 11.5: Git commit of an adsprpc UAF access.

History Tree Search. Multiple kernel driver source codes are available via git repositories hosted by Google or ODM vendors, such as Qualcomm. For example, Qualcomm's DSP `frpc-adsprpc` kernel driver repository is publicly accessible at `https://git.codelinaro.org/clo/la/platform/vendor/qcom/opensource/dsp-kernel.git`. While we demonstrate our approach using two security flaws in the `frpc-adsprpc` driver, this method can be applied generically to all publicly available repositories.

Our approach involves searching the entire repository history for keywords that suggest a commit addresses a security flaw. Examples of such keywords include *bug*, *use-after-free*, and *out-of-bounds*. Using these, we identify vulnerabilities in the git repository. One example is commit `2466b` in mid-2024, as shown in Listing 11.4. This patch introduces a check to ensure that the `sesscount` member variable from the `struct fastrpc_-`

Table 11.2: The capability granted by n-day vulns per driver.

Driver Module	Capability				
	UAF	OOB	Others	ID	Total
frpc-adsprpc.ko	12	3	1	3	19
jpeg-driver.ko	2	1	0	0	3
mtk_perf_ioctl.ko	1	0	4	7	12
apusys.ko	0	1	1	1	3
ged.ko	1	2	3	3	9
trusted_mem.ko	0	2	1	0	3
xlog.ko	0	0	1	0	1
	16	9	11	14	50

UAF: Use-after-free and double-free **OOB:** Out-of-bound write
Others: Uninit variable, null pointer deref and denial of service
ID: Out-of-bound read and information disclosure

`channel_ctx` remains within its intended range of `[0, NUM_SESSIONS-1]`. This patch adds the following check: `VERIFY(err, chan->sesscount < NUM_SESSIONS)`. Before, a malicious actor could exploit the vulnerability to perform an Out-Of-Bounds (OOB) write in the `memcpy` function. Exploitation is possible as `chan->session[chan->sesscount]` (aliased as `dup_sess`) would be misinterpreted as `fastrpc_channel_ctx`. OOB writes are a common initial exploit primitive with the potential for system compromise [9, 45, 48, 50, 92].

Another example of an identified n-day vulnerability is the end-2023 Use-After-Free (UAF) flaw involving the `fastrpc_mmap`, as shown in Listing 11.5. We identified this flaw by searching the history for *UAF*. According to the commit message, this vulnerability can pivot to a Double-Free (DF) scenario, also a robust exploit primitive [45, 50, 87, 92].

5.2. N-Day Analysis

We manually collect publicly available git repositories and bug reports. The repositories are sourced from device OEMs (e.g., Xiaomi and OnePlus), ODMs (e.g., Qualcomm and MediaTek), and Google. The bug reports are obtained from platforms such as Google Project Zero's issue tracker. None of the bug reports referenced the specific patches that fixed the discovered vulnerabilities. It was, therefore, part of our analysis to identify the corresponding patches for the report.

Our analysis involves a two-step process. First, we automatically filter commit messages from these repositories using security-related keywords, as described in Section 5.1. We started with commit messages from 2020 to December 2024 (date of analysis). Then, we manually verified the filtered commits to confirm whether the changes addressed security-critical bugs. Only commits with clear evidence are included in our analysis, e.g., in Listings 11.4 and 11.5. Using this approach, we identified 50 security-critical issues: 45 initially from git repositories and 7 from bug reports, with 2 representing duplicates found in both sources. For the analysis, we categorized them based on their exploit capabilities. While several stem from race conditions, their capabilities result in a UAF or an OOB write. The n-day vulnerabilities are classified into four categories (see Table 11.2):

UAF: Use-after-free access and double-free.
OOB: Out-of-bound write.
Others: Uninitialized variable access, null pointer dereference and denial of service.
ID: Out-of-bound read and information disclosure.

These categories are critical for compromising Android devices. UAF, DF, and OOB write capabilities are particularly notable, as they serve as initial exploit primitives with numerous exploit techniques for system compromise [9, 13, 21, 45, 50, 61, 85, 87, 92]. Other vulnerabilities, such as null pointer dereferences and Denial Of Service (DOS), also offer pathways to root. For example, Jenkins demonstrated an innovative approach to exploiting these issues [29]. Similarly, prior work has shown how to effectively exploit Uninitialized Variables (UV) [11, 38, 47]. Information Disclosure (ID) capabilities are essential in end-to-end exploitation, as demonstrated by prior research [43, 44, 48, 51]. These vulnerabilities allow malicious actors to locate target kernel objects, which most kernel exploits require [26, 28, 51, 64, 66, 67, 79].

Table 11.2 shows the results of our analysis of n-day driver vulnerabilities (excluding GPU drivers), with 50 vulnerabilities identified. Dividing the results into different categories, we observe that vulnerabilities affecting UAF capabilities are the most prevalent, while OOB writes are the least prevalent. We also observe a variation in the number of vulnerabilities identified per driver. We did not find matching git repositories for Samsung's `npu.ko` and `camera.ko`, Xiaomi's `migt.ko` and the `apuext` part of `apusys.ko`. For `mi_log.ko`, we found a repository, but no commits indicating security-related fixes.

Validity of Analysis. We reconstructed n-day vulnerabilities using open-source information from the respective git repositories. Our reconstruction is based on security experts identifying patches that fix vulnerabilities, including capabilities such as UAF, DF, or OOB writes. However, we do not claim or evaluate whether each specific vulnerability enables direct system compromise. We even argue that limiting the focus solely to vulnerabilities already proven to enable direct system compromise overlooks broader security risks and creates a harmful trend in vulnerability prioritization. The rise of novel exploitation techniques [9, 21, 28, 29, 39, 44, 45, 46, 47, 50, 61, 80, 85, 86, 87, 89, 91, 92] demonstrates that increasingly weaker exploit primitives can still lead to system compromise despite modern defenses. For example, overwriting one byte with zero [13, 45, 60] has been shown to compromise modern systems. Similarly, null pointer dereferences, which were considered mitigated after the introduction of `mmap_min_addr`, were re-enabled due to a novel kernel exploit technique to compromise recent Linux systems [29]. This shows that even vulnerabilities perceived as low- to no-risk can be leveraged for system compromise.

Given this, we argue that neither security researchers nor vendors should primarily focus their time and effort on determining whether each n-day vulnerability is directly exploitable for system compromise. This, in turn, can lead to poor prioritisation of vulnerabilities, wasting time and resources that could be spent integrating patches. We, therefore, conclude that if an accessible vulnerability falls into a category known to facilitate system compromise, such as UAF, DF, or OOB writes, this should suffice to classify it as critical. Instead, the focus should be on promptly incorporating patches to mitigate these vulnerabilities and prevent their exploitation. However, while we do not demonstrate the exploitability of each vulnerability, we verify the reachability of vulnerable code for a representative subset of vulnerabilities on a real device, supporting n-day exploitability (see Section 6.3).

6. Detecting N-Day Patches in Kernel Drivers

In this section, we perform a large-scale analysis of the inclusion of n-day patches in kernel modules for ODM-specific device drivers. We examine the absence and delay of patches for a subset of the 50 n-day vulnerabilities identified in Section 5.2. Our dataset comprises 493 firmware versions from 131 devices across 7 OEMs, with multiple versions available for devices

Table 11.3: Device firmware categorized by security patch level and release date by OEM vendor.

OEM	**2023**												**2024**											**Total**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	
Asus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	2	6
OnePlus	0	0	3	1	0	0	0	0	1	0	2	0	0	0	0	1	0	1	0	0	0	0	0	9
Oppo	0	0	0	0	0	0	0	0	1	0	0	1	2	0	0	0	0	2	1	0	0	0	0	7
Realme	0	0	0	2	0	0	0	0	0	0	0	1	0	2	1	0	2	0	1	3	3	0	1	16
Vivo	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1	0	0	1	2	0	0	7
Xiaomi	0	0	1	1	5	2	3	1	2	2	2	12	12	8	21	10	16	16	26	24	30	19	18	231
Samsung	1	3	4	4	0	2	4	5	2	0	6	12	7	8	15	14	10	24	10	19	12	25	30	217

from Samsung and Xiaomi. For other devices, we use the most recent firmware available.

To assess the susceptibility of devices to n-day vulnerabilities, we evaluate each device based on its most recent available firmware. Specifically, we test whether known vulnerabilities—publicly available before this available firmware release—are still present. We show that 61.4 % of these devices have at least one n-day vulnerability, making them vulnerable to exploitation. Alarmingly, 59.1 % of these devices contain at least one highly critical n-day vulnerability. These highly critical vulnerabilities fall into the UAF or OOB categories. They provide initial exploit primitives with well-researched exploit techniques [9, 13, 21, 45, 50, 61, 85, 87, 92]. We also analyze the patch delay across 131 devices based on their respective security patch levels. Our results show significant delays in patch integration, with delays up to 830 d. These results demonstrate that our identified drivers are susceptible targets for malicious actors, satisfying **C3**.

To perform this large-scale analysis, we present a semi-automatic approach for detecting patch presence for 21 of the 50 n-day driver vulnerabilities (see Section 6.1). Using this approach, we analyze compiled kernel modules extracted from device firmwares (see Section 6.2). Lastly, we demonstrate on a representative subset of these vulnerabilities that they are triggerable on a real device (see Section 6.3).

Firmware Versions. We have a total of 493 firmware versions from 131 devices across 7 OEM vendors. For Xiaomi and Samsung, our collection contains the latest firmware version for multiple devices as well as older versions. For the other OEM vendors (i.e., Oppo, OnePlus, Asus, Realme, and Vivo), we analyze the most recent firmwares available online, ranging between early-2023 and end-2024. Table 11.3 shows the amount of firmwares for each security patch level and OEM.

6.1. Patch Detection Approach

We demonstrate a semi-automated approach for detecting security patches in kernel drivers by analyzing code modifications in the compiled kernel modules. We exclude patches that fall outside our analytical framework's strategies, which focus on two primary methods: symbol-based detection and control-flow analysis through decompilation.

Symbol-Based Detection. We leverage the fact that security patches frequently introduce symbols. These could, e.g., be references to global functions, global variables, or unique string artifacts for error messages and diagnostic output. For globally accessible symbols, our approach analyzes whether the symbols introduced in the source patch are present in the compiled target driver. We exclude inlined function symbols, as these may not result in detectable changes in the compiled binary. To eliminate false negatives, we only consider cases where these symbols are also present in the most recent version of the kernel driver. String artifacts provide an additional possibility for patch detection. Security patches frequently include unique strings, such as warnings or diagnostic messages (see Listing 11.4), that can be used as identifiers in the binary representation of the driver. We exclude cases where introduced strings are not unique, e.g., because they had been used in other places before the patch already.

On symbol absent, further manual verification is required to rule out false negatives, involving two key considerations: kernel driver configurability and code evolution. Kernel drivers often implement configurations that selectively exclude code segments, requiring manual analysis to verify whether patch-affected code exists. Additionally, subsequent or custom commits may modify or replace strings the patch introduces, potentially obscuring its presence. Our approach accounts for these changes to ensure accurate patch detection.

Control-Flow Analysis. Security patches frequently implement modifications to program control flow [35, 94], typically through additional conditional logic, as demonstrated in Listing 11.5. Our control-flow analytical approach focuses on identifying patches that modify program logic on patch-modified functions through a two-phase process. Initially, we perform differential analysis of assembly code across sequential driver versions. The absence of assembly-level differences between versions indicates patch exclusion within that interval. For versions exhibiting assembly modifications, we employ Ghidra to perform decompilation. Although the decompilation output does not perfectly match the original code, it enables the identification of control-flow modifications through comparative manual analysis. Combining both approaches allows for efficient manual verification while not degrading output performance, especially since most changes in driver functions are due to security issues, as observed.

Future Work. While we used a tailored approach to detect patches in ODM-specific drivers, prior work [35, 94] focused on pre-GKI kernel

Table 11.4: Devices' n-day susceptibility to vulnerabilities known as of their most recent firmware release. **All Device Analyzed** includes all studied devices with the most firmware versions, while **Devices with Target Drivers** refers to those having hardware support for at least one of the target drivers.

OEM	All Devices Analyzed		Devices with Target Drivers	
	Crit Vuln	Any Vuln	Crit Vuln	Any Vuln
	%	%	%	%
Samsung	45.5	45.5	74.1	74.1
Xiaomi	67.3	71.4	75.0	79.5
Asus	75.0	100.0	75.0	100.0
Realme	56.2	62.5	56.2	62.5
Vivo	40.0	40.0	40.0	40.0
Oppo	42.9	42.9	42.9	42.9
OnePlus	85.7	85.7	85.7	85.7

images. Notably, PDiff [35] did not release their approach as open source, but Fiber [94] or similar approaches, such as those using angr [68], could be adapted to detect the absence of patches. However, Fiber was originally designed for kernel images rather than kernel modules, and adapting it to our methodology would require significant engineering effort. Fiber's evaluation was also limited to 11 kernel images using custom execution allowlists. To meet the requirements of our dataset, which includes 493 firmwares, each containing 1 to 5 relevant kernel modules, we would need to significantly extend Fiber's capabilities for efficient patch detection. Given these challenges, we suggest extending Fiber or developing a similar framework as future work.

6.2. Large-Scale Analysis on Patch Inclusion

By integrating symbol-based detection and control-flow analysis, our methodology ensures the identification of security-related patches. To evaluate the effectiveness of our approach, we select 21 n-day vulnerabilities from the full set of 50 identified, focusing on vulnerabilities in the DSP, JPEG, and GED kernel drivers, such as Listings 11.4 to 11.10. These vulnerabilities include highly critical ones (i.e., 9 UAF and 6 OOB) as well as moderately critical ones (i.e., 1 Others and 5 ID), representing a balanced mix of different severity levels. Our main goals are: to quantify how many devices—running their respective newest available firmware

versions—remain susceptible to these n-day vulnerabilities and assess security-relevant patch delays.

Device Susceptibility. We assess device susceptibility under two conditions. First, we analyze the most recent firmware available for all 131 Android devices in our dataset (see Table 11.3) to determine what fraction of the 21 n-day vulnerabilities each device is still vulnerable to. Second, we repeat this only for devices with hardware support for at least one of the target kernel drivers identified as potentially vulnerable.

Table 11.4 shows susceptibility rates, separating results for all devices (**All Devices Analyzed**) and those with hardware support of at least one target driver (**Devices with Target Drivers**). In both cases, we observe widespread susceptibility to highly and moderately critical n-day vulnerabilities. For instance, 45.5 % of Samsung devices and 71.4 % of Xiaomi devices were found to be vulnerable to at least one n-day vulnerability. Aggregated across all devices, 59.1 % were vulnerable to at least one highly critical vulnerability, while 61.4 % were vulnerable to at least one of any severity. Susceptibility rates for Samsung and Xiaomi appear lower primarily due to the lack of hardware support for the drivers we analyzed. When excluding devices without the analyzed hardware—like Samsung's NPU—susceptibility rates rise to 74.1 % for Samsung and 79.5 % for Xiaomi.

Crucially, while Table 11.4 shows patching trends, it does not support a direct comparison between lineage-providing OEMs (i.e., Xiaomi and Samsung) and others, as, e.g., many 2024 vulnerabilities cannot be tested on non-lineage vendors that only publicly provide older firmwares.

OEM Breakdown. Susceptibility is often not limited to a single vulnerability. We find a correlation between being affected by one vulnerability and being affected by multiple. For example, 29.5 % of Samsung devices and 49 % of Xiaomi devices were vulnerable to three or more n-day vulnerabilities.

Takeaway 11.1

If a device is susceptible to 1 n-day vulnerability, it is likely susceptible to multiple vulnerabilities.

We also observe a trend in patch behavior across OEMs, with Xiaomi standing out. Active patching of vulnerabilities in Xiaomi devices is limited, with only a few instances observed that patch the identified n-day

Table 11.5: Lower bound for average n-day patch delays in years across device OEM vendors and vulnerability categories.

OEM	Average Delay Time (Lower Bound)			
	UAF	OOB	Others	ID
Samsung	0.32 y±0.4	0.40 y±0.1	0.32 y±0.0	0.79 y±0.7
Xiaomi	0.56 y±0.4	0.70 y±0.9		0.88 y±0.6

vulnerabilities. In many cases, when the earliest firmware version of a device is found to be susceptible to n-day vulnerabilities, subsequent firmware versions tend to remain susceptible. This pattern suggests that security flaws are often addressed indirectly through the release of newer Android devices with updated kernel driver versions that include the necessary patches rather than through firmware updates for existing devices. While this approach is more prominent in Xiaomi's practices, similar tendencies are also observed, albeit notably lesser, in Samsung's handling of such vulnerabilities.

Takeaway 11.2

Security-related flaws are likely addressed through new device releases than through updates to existing devices.

Patch Delays. We define a patch as either integrating a fix into existing software or entirely replacing the affected software with a non-susceptible version. Our analysis reveals variability in how patches propagate across firmware versions. Table 11.5 quantifies the time gap between the released commit date of a patch and the last analyzed firmware version where the patch was missing. The precision of these results depends on the granularity of our firmware dataset, which varies between OEM vendors. Our dataset contains temporal gaps of months. Patch integration could have occurred between the release of the last unpatched and the first patched firmware in our dataset. Our results are, therefore, conservative estimates for the patch integration delay. Specifically, some patches are missing from the latest firmware releases at the time of our analysis. In such cases, we conservatively estimate the delay metrics by assuming that the patch will be included in the next firmware release, which is a lower bound. Hence, our delay measurements will likely underestimate actual patch delays, which may be higher.

Table 11.5 highlights variations in patch delays across OEMs and vulnerability types. Samsung generally patches security-critical flaws faster

Table 11.6: Lower bound for average n-day patch delays in years across device ODMs and vulnerability categories.

ODM	Average Delay Time (Lower Bound)			
	UAF	OOB	Others	ID
Qualcomm	0.38 y±0.4	0.23 y±0.2	0.32 y±0.0	0.65 y±0.6
MediaTek	0.71 y±0.5	2.15 y±0.1		2.00 y±0.3

than Xiaomi. Among vulnerability types, OOB and Others are patched the quickest, followed by UAF, while ID takes the longest. This trend reflects the perception that information disclosure vulnerabilities are less critical than UAF or OOB. However, UAF and OOB vulnerabilities can still take over 500 d to patch, with ID exceeding 800 d. On average, OOB vulnerabilities are patched more quickly than UAF, which are patched faster than ID.

ODM Breakdown. In addition to analyzing patch delays solely by OEM and vulnerability type, we further break down the results by ODM chipset. Samsung's Exynos-based devices are excluded from this analysis due to the lack of publicly available source repositories. Table 11.6 presents the results of this ODM-based breakdown, revealing notable differences in patching delays between Qualcomm- and MediaTek-based chipsets. Compared to their Qualcomm counterparts, MediaTek devices consistently exhibit longer patch delays of over two times across OEMs and multiple vulnerability classes. For example, for UAF vulnerabilities, the average patch delay increases from approximately 4 months for Qualcomm to over 8 months for MediaTek.

Takeaway 11.3

While the patch delay varies by OEM, ODM, and vulnerability, some devices experience delays of over a year.

6.3. Representative Subset of N-Day Vulns

To support the claim that n-day vulnerabilities remain exploitable, we demonstrate that a representative subset can be reliably triggered on a real Android device. Developing Proof-of-Concept (PoC)s is a non-trivial task requiring substantial time, device access, and in-depth driver-specific expertise, even for experienced analysts like those at Google Project Zero [22,

28]. Given this complexity, we focus the following analysis on a single device driver to ensure feasibility. We target `frpc-adsprpc.ko`, which accounts for most identified vulnerabilities. Since we apply the same patch-based approach across all affected modules, dynamically validating one representative driver shows reachability and supports the generalizability of our approach.

We select five vulnerabilities from this driver with their git patch commit messages shown in Listings 11.6 to 11.10. These five vulnerabilities represent a range of bug classes: one OOB read (**vuln1**), three UAF issues (**vuln2-4**), and one information disclosure (**vuln5**). We evaluate these across 29 module versions covering various release dates and 13 Android devices from three major OEMs (Samsung, Xiaomi, and Asus), providing a representative subset of both OEMs and n-day exposure timelines. Our findings (see Section 6.3.1) show that on Samsung devices, the vulnerabilities remain n-day exploitable for 0 to 7 months. In contrast, on all tested Xiaomi and Asus devices, multiple vulnerabilities remained exploitable even at the time of analysis. The results for the representative subset align with the results from our patch detection approach, supporting the validity of our static determination (see Section 6.3.2).

We use a rooted Samsung Galaxy S23 equipped with the Qualcomm SM8550-AC Snapdragon 8 Gen 2 chipset for testing. Even after device rooting, most partitions (including the one containing kernel drivers) are mounted using the Enhanced Read-Only File System (EROFS), preventing direct modification. To circumvent this, we flash TWRP/RO2RW images and remount the `vendor_dlkm` partition as writable, enabling us to replace the `frpc-adsprpc.ko` driver with a test version. Using this setup, we test driver versions from 13 Android phones, spanning multiple timestamps and three OEMs. All 13 devices use the same chipset as our test device, ensuring compatibility. We manually confirm that substituting the driver module from another OEM with the same chipset does not alter functionality. This allows us to test different versions while keeping the engineering effort reasonable.

Triggering the vulnerabilities produces two observable effects: the UAF issues and the OOB read cause a crash, while the information disclosure leaks a kernel pointer. We use publicly available PoCs where possible (e.g., for **vuln1** [32]) or develop one (e.g., for **vuln5** [31]) to trigger each vulnerability.

6.3.1. Analysis

We present our findings in Table 11.7. We begin by testing 10 versions of `frpc-adsprpc.ko` from the original Galaxy S23, covering firmware releases available between March 2024 (the earliest relevant commit) and December 2024 (the time of analysis). We find that the October 2024 release is the first to include all five patches that mitigate the tested vulnerabilities. All vulnerabilities are tested for S23 original drivers. However, we omit **vuln3** in broader tests on devices other than the S23 due to its long triggering time (more than 12 hours).

Next, we test the September and October 2024 driver versions across 4 additional Samsung models (Galaxy S23+, S23 Ultra, Z Flip5, and Z Fold5). Consistent with the S23 results, the October release contains the full set of patches. Patch delays for Samsung devices range from 0 to 7 months, depending on the vulnerability.

We then test the November and December 2024 releases for six Xiaomi models (MIX Fold 3, Redmi K70, POCO F6 Pro, Xiaomi 13 Ultra, 13 Pro, and 13) and the November 2024 release for two ASUS models (ROG Phone 7 and 7 Ultimate). All of these most recent releases remain vulnerable to **vuln2/5**, while four are also vulnerable to **vuln1/4**. Patch delays vary between 0 and over 9 months, depending on the device and specific vulnerability.

Takeaway 11.4

A PoC for an n-day vulnerability in ODM-maintained drivers can be reused across OEMs and timeframes.

6.3.2. Validity Check of Patch Detection

We have two sets of patch inclusion analysis results: the large, statically determined set described in Section 6.2, and the smaller, dynamically determined subset discussed in Section 6.3. We now validate the dynamically determined subset by comparing it against the statically determined results, confirming consistency between them.

7. Discussion and Related Work

This section discusses the security implications and validity of our findings, as well as related work.

Security Implications. Our analysis reveals that modern Android devices have a significant kernel attack surface reachable from untrusted contexts, comprising kernel device drivers. Many of these drivers expose known n-day vulnerabilities for long periods of time, allowing malicious actors to bypass the effort of developing zero-day exploits. Instead, they can exploit these n-day vulnerabilities to compromise devices. Crucially, as a device is only as secure as its weakest point, our research highlights that device drivers represent this weakest link in current Android versions. A single pathway to root access is sufficient to fully compromise Android devices.

Takeaway 11.5

Malicious actors can exploit n-day vulnerabilities, reducing reliance on time-consuming zero-day development.

Validity of our Results. Our findings are mostly derived from static analysis. To ensure consistency with real-world scenarios, we incorporated dynamic testing and manual verification throughout. While we demonstrate n-day triggering for 5 vulnerabilities in Qualcomm-supplied DSP drivers, our evaluation does not include full end-to-end exploitation. Furthermore, static analysis has inherent limitations, such as missing dynamic behaviors, and our evaluation focuses on a subset of drivers and devices. As a result, there may be unknown barriers to triggering the vulnerabilities and achieving full exploitation. The reported numbers should, therefore, be interpreted as estimates of real-world exploitability.

Patch Detection and Propagation. Prior work [35, 42, 94] has demonstrated methods for detecting patches in kernel binaries by, e.g., deriving patch's signatures and testing them against the kernel binary. Numerous studies have explored solutions to mitigate the effects of delayed patch integration. Wang et al. [81] proposed temporary patch integration, while Chen et al. [10] and Xu et al. [90] introduced hot patch techniques to mitigate vulnerabilities dynamically. Talebi et al. [78] prevented harmful side effects of vulnerable code through syscall instrumentation. Another security-related problem is that while patches are available, vendors are

reluctant to apply them. Hence, other research [52] has focused on faster and more correct patch propagation.

Patch and Defense Integration. The deployment of security updates and defenses in Android systems has been a focus of various studies. Wu et al. [82] highlighted that most issues in the Android Security Bulletin (ASB) originate from native code, while Farhang et al. [15] observed that kernel-related CVEs experience the longest delays in propagating to vendor ASBs. This delay creates a window for attackers to exploit vulnerabilities before patches reach end users. Jones et al. [37] and Zhang et al. [96] quantified the time lag, reporting delays of weeks to months for Android security updates. Acar et al. [1] revealed significant fragmentation in Android's security update ecosystem, with inconsistent and delayed patch rollouts across devices, vendors, and regions. Maar et al. [48] recently analyzed the challenges of integrating mainline kernel defenses against n-day exploitation. Most of these studies focused on Android versions that predate the GKI initiative, which was intended to address kernel patch delays for good. However, we show that these delays remain a threat to device security as vendors struggle to integrate patches in those parts of the kernel.

Security Analysis on Android. Google Project Zero has been tracking zero-day exploits targeting Android since 2019 [73]. Their annual reviews [69, 72, 73, 77] analyze trends in malicious actor behavior to enhance Android security. These reports emphasize the critical role of timely patch deployment and defense integration in mitigating in-the-wild exploitation [62]. Other research groups, such as the Threat Analysis Group [76], GitHub Security Lab [56, 57, 58, 59], Zero Day Engineering [14], Blue Frost Security [66, 67], Amnesty International's Security Lab [22, 23, 62], and Citizen Lab [40, 41, 54, 62], also analyze exploitation trends.

Android Driver Security. Exploiting vulnerabilities in kernel drivers, particularly the GPU, has been a key focus of recent research [8, 14, 16, 17, 18, 56, 58, 59, 75, 76, 77, 85, 88]. Collaboration between Google, Android, and ARM has contributed to advances in vulnerability detection, mitigation, and hardening [88]. NPU drivers have also been targeted [57, 63, 84, 95], although such exploits have predominantly focused on Samsung devices due to the unclear adoption of similar vulnerabilities in Qualcomm-based devices [57].

Research on DSP-related kernel drivers remains sparse [14], but recent and concurrent exploits [22, 33] show that these drivers were actively exploited to compromise Android devices. Malicious actors exploited these drivers to install spyware, such as NoviSpy, which specifically targets end users in the wild [22]. These findings underscore the relevancy of our work: *the exploitability of device drivers is known to malicious actors, so it is imperative that research catches up.*

Most recently, concurrent work by Amnesty International's Security Lab [23] demonstrated that kernel device drivers remain a primary attack vector for Android compromise. In particular, malicious actors deployed zero-day exploits against Android USB kernel drivers observed in-the-wild. Similar to the 2024 DSP zero-day exploits [22], malicious actors then installed NoviSpy spyware for surveillance.

While some studies have examined vulnerabilities in drivers accessible from trusted contexts [36, 83], the pervasive threat posed by drivers accessible from untrusted contexts remains largely unexplored. To the best of our knowledge, Jenkins has done the closest analysis of the kernel attack surface from untrusted security contexts [27]. Jenkins analyzed the attack surface for 3 devices, i.e., Google Pixel 7, Xiaomi 11T, and Asus ROG 6D, and presented multiple zero-day vulnerabilities, demonstrating that security research on Android drivers is sparse. We, on the other hand, performed a large-scale analysis of 131 devices and discovered a large number of drivers that are accessible and, worse, exploitable with n-day vulnerabilities from untrusted contexts.

8. Conclusion

Prior compromises of Android devices often relied on exploit chains targeting GPU kernel drivers or higher user-space privileges before targeting the kernel. In this paper, we comprehensively analyzed the kernel attack surface exposed to untrusted security contexts. Our analysis reveals that this attack surface is significantly larger than previously known, comprising multiple device drivers. Within these drivers, we identified vulnerabilities that remain unpatched for extended periods or were still unpatched at the time of our analysis. Specifically, 59.1 % of recent Android devices were vulnerable to highly critical n-day exploits, with 61.4 % affected by any vulnerability. These unpatched vulnerabilities present an ideal target for malicious actors, as they eliminate the need to invest substantial time

and effort into developing zero-day exploits. This critical state of Android kernel security highlights the need for urgent action to enhance patch management and improve overall security.

9. Acknowledgements

We thank the anonymous reviewer and shepherd for their valuable feedback. We also thank Jann Horn and Seth Jenkins for their help with triggering the PoCs, and the MediaTek Product Security Team for identifying an error we subsequently fixed. This research was funded in whole or in part by the Austrian Research Promotion Agency (FFG) via the SEIZE and AWARE project (FFG grant numbers 888087 and 915106). Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

10. Ethics Considerations

We responsibly disclosed our findings to all affected parties, including OEMs (Samsung, Xiaomi, Asus, Realme, OnePlus, Oppo, and Vivo), ODMs (Samsung, Qualcomm, and MediaTek), and Google.

- Asus, Realme, and MediaTek acknowledged our findings.
- Samsung acknowledged the issues and has initiated patching for affected devices. We had a follow-up meeting focused on improving Android driver security.
- Google responded by stating that the issues fall outside their scope of enforcement and directed us to report the findings to OEMs directly.

Following best practices from Google Project Zero, we gave all participants more than 90 days to address issues, with a 30-day grace period. This timeline left plenty of buffer time for the earliest possible release date of this paper, allowing all participants ample opportunity to develop and implement comprehensive solutions before public release.

Moreover, we believe revealing these findings is crucial to demonstrating how malicious actors can exploit the Android environment. This underscores the need for urgent action to improve overall Android security by addressing patch delays, prioritizing device updates, and securing kernel

attack surfaces. We strongly believe that publishing our findings will improve the security of Android devices in the long term and is, therefore, the most ethical course of action.

We are committed to an ethical approach that balances responsible research with potential security improvements. Our research relies exclusively on publicly available firmware images obtained from multiple sources, including official OEM/ODM vendor repositories and third-party providers. While we recognize that methodological precedent alone cannot justify research ethics, prior work [1, 37, 48, 96] reinforces our belief in the validity of analyzing publicly available data. We have not analyzed how the firmwares have been obtained by the third-party providers.

All dynamic testing was conducted in a controlled laboratory environment using dedicated research devices, further ensuring the integrity and safety of our investigation.

11. Open Science

While we aim to make all datasets, crawling tools, and analysis scripts open source, doing so poses a risk of misuse by malicious actors, as observed in the past [26, 54, 71, 74, 76, 77]. Consequently, we do not recommend open-sourcing these resources. However, if the USENIX committee holds a different opinion, we will share all our findings and tools accordingly.

References

[1] Abbas Acar, Güliz Seray Tuncay, Esteban Luques, Harun Oz, Ahmet Aris, and Selcuk Uluagac. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In: NDSS. 2024 (pp. 419, 445, 448).

[2] Android. Application Sandbox. 2021. URL: `https://source.android.com/security/app-sandbox` (pp. 413, 416).

[3] Android. Build SELinux policy. 2024. URL: `https://source.android.com/docs/security/features/selinux/build` (p. 422).

[4] Android. Generic Kernel Image (GKI) project. 2024. URL: `https://source.android.com/docs/core/architecture/kernel/generic-kernel-image` (pp. 415, 416).

[5] AppBrain. Top manufacturers. accessed: 31.12.2024. 2024. URL: https://web.archive.org/web/20241230133601/https://www.appbrain.com/stats/top-manufacturers (pp. 419, 428).

[6] Brandon Azad. A survey of recent iOS kernel exploits. 2020. URL: https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html (p. 415).

[7] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In: USENIX ATC. 2019 (p. 420).

[8] Ian Beer. Mind the Gap. 2022. URL: https://googleprojectzero.blogspot.com/2022/11/ (pp. 412, 445).

[9] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In: USENIX Security. 2020 (pp. 413, 432–434, 436).

[10] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android Kernel Live Patching. In: USENIX Security. 2017 (pp. 416, 444).

[11] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. In: WOOT. 2020 (pp. 413, 433).

[12] Chromium. PowerVR GPU - Kernel heap OOB write in RGXFWChangeOSidPriority - CVE-2024-23698. 2024. URL: https://apvi.issues.chromium.org/issues/42420036 (p. 429).

[13] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel. 2022. URL: https://syst3mfailure.io/corjail/ (pp. 433, 434, 436).

[14] Alisa Esage. Deep Dive: Qualcomm MSM Linux Kernel & ARM Mali GPU 0-day Exploit Attacks of October 2023. 2023. URL: https://zerodayengineering.com/insights/qualcomm-msm-arm-mali-0days.html (pp. 412, 445, 446).

[15] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An Empirical Study of Android Security Bulletins in Different Vendors. In: WWW. 2020 (p. 445).

[16] Guang Gong. TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices. 2020. URL: https://github.com/secmob/TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices/blob/master/us-20-Gong-TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices-wp.pdf (pp. 412, 445).

[17] Xiling Gong, Xuan Xing, and Eugene Rodionov. The Way to Android Root: Exploiting Your GPU On Smartphone. 2024. URL: https://i.blackhat.com/BH-US-24/Presentations/REVISED02-US24-Gong-The-Way-to-Android-Root-Wednesday.pdf (pp. 412, 445).

[18] Ben Hawkes. Attacking the Qualcomm Adreno GPU. 2020. URL: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html (pp. 412, 428, 445).

[19] Jann Horn. adsprpc: refcount leak leading to UAF in fastrpc_get_process_gids. 2024. URL: https://project-zero.issues.chromium.org/issues/42451711 (pp. 424, 461).

[20] Jann Horn. CVE-2022-22706 / CVE-2021-39793: Mali GPU driver makes read-only imported pages host-writable. 2022. URL: https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2021/CVE-2021-39793.html (p. 429).

[21] Jann Horn. How a simple Linux kernel memory corruption bug can lead to complete system compromise. 2021. URL: https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html (pp. 433, 434, 436).

[22] Amnesty International. "A Digital Prison": Surveillance and the suppression of civil society in Serbia. 2024. URL: https://securitylab.amnesty.org/latest/2024/12/a-digital-prison-surveillance-and-the-suppression-of-civil-society-in-serbia/ (pp. 412, 414, 441, 445, 446).

[23] Amnesty International. Cellebrite zero-day exploit used to target phone of Serbian student activist. 2025. URL: https://securitylab.amnesty.org/latest/2025/02/cellebrite-zero-day-exploit-used-to-target-phone-of-serbian-student-activist/ (pp. 414, 445, 446).

[24] Amnesty International. Europe: Paragon attacks highlight Europe's growing spyware crisis. 2025. URL: https://www.amnesty.org/en/latest/news/2025/03/europe-paragon-attacks-highlight-europes-growing-spyware-crisis/ (pp. 415, 418).

[25] Amnesty International. Forensic Methodology Report: How to catch NSO Group's Pegasus. 2021. URL: https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-how-to-catch-nso-groups-pegasus/ (p. 412).

[26] Seth Jenkins. Analyzing a Modern In-the-wild Android Exploit. 2023. URL: https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html (pp. 412, 415, 418, 419, 433, 448).

[27] Seth Jenkins. Driving forward in Android drivers. 2024. URL: https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html (pp. 423, 429, 446).

[28] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack. 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html (pp. 433, 434, 442).

[29] Seth Jenkins. Exploiting null-dereferences in the Linux kernel. 2023. URL: https://googleprojectzero.blogspot.com/2023/01/exploiting-null-dereferences-in-linux.html (pp. 433, 434).

[30] Seth Jenkins. FASTRPC_ATTR_KEEP_MAP logic bug allows fastrpc_internal_munmap_fd to concurrently free in-use mappings leading to UAF. 2024. URL: https://project-zero.issues.chromium.org/issues/42451725 (p. 461).

[31] Seth Jenkins. Incorrect searching algorithm in fastrpc_mmap_find leads to kernel address space info leak. 2024. URL: https://project-zero.issues.chromium.org/issues/42451713 (pp. 442, 462).

[32] Seth Jenkins. is_compat flag in adsprpc driver leads to access of userland provided addresses as kernel pointers. 2024. URL: https://project-zero.issues.chromium.org/issues/42451710 (pp. 442, 460).

[33] Seth Jenkins. The Qualcomm DSP Driver - Unexpectedly Excavating an Exploit. 2024. URL: https://googleprojectzero.blogspot.com/2024/12/qualcomm-dsp-driver-unexpectedly-excavating-exploit.html (pp. 414, 424, 446).

[34] Seth Jenkins. UAF race of global maps in fastrpc_mmap_create (and epilogue functions). 2024. URL: https://project-zero.issues.chromium.org/issues/42451715 (p. 460).

[35] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In: CCS. 2020 (pp. 416, 437, 438, 444).

[36] Xingyu Jin and Clement Lecigene. CVE-2024-44068: Samsung m2m1shot_scaler0 device driver page use-after-free in Android. 2024. URL: https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2024/CVE-2024-44068.html (pp. 418, 446).

[37] Kailani R. Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G. Bardas. Deploying Android Security Updates: an Extensive Study Involving Manufacturers, Carriers, and End Users. In: CCS. 2020 (pp. 445, 448).

[38] Max Kellermann. The Dirty Pipe Vulnerability. 2022. URL: https://dirtypipe.cm4all.com/ (pp. 413, 433).

[39] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security. 2014 (p. 434).

[40] Citizen Lab. Blastpass NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild. 2023. URL: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/ (p. 445).

[41] Citizen Lab. ForcedEntry NSO Group iMessage Zero-Click Exploit Captured in the Wild. 2021. URL: https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/ (p. 445).

[42] Jakob Lell and Karsten Nohl. Mind the Gap - Uncovering the Android patch gap through binary-only patch analysis. 2018. URL: https://conference.hitb.org/hitbsecconf2018ams/materials/D2T1%20-%20Karsten%20Nohl%20&%20Jakob%20Lell%20-%20Uncovering%20the%20Android%20Patch%20Gap%20Through%20Binary-Only%20Patch%20Level%20Analysis.pdf (p. 444).

[43] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In: NDSS. 2024 (pp. 413, 433).

[44] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In: S&P. 2022 (pp. 413, 433, 434).

[45] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In: CCS. 2022 (pp. 432–434, 436).

[46] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android. 2023. URL: `https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf` (pp. 418, 434).

[47] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In: NDSS. 2017 (pp. 413, 433, 434).

[48] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024 (pp. 413, 415, 419, 432, 433, 445, 448).

[49] Lukas Maar, Florian Draschbacher, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard. The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities. In: USENIX Security. 2025 (p. 409).

[50] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 413, 432–434, 436).

[51] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025 (p. 433).

[52] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In: S&P. 2020 (p. 445).

[53] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In: USENIX Security. 2017 (p. 420).

[54] Bill Marczak, Adam Hulcoop, Etienne Maynier, Bahr Abdul Razzak, Masashi Crete-Nishihata, John Scott-Railton, and Ron Deibert. Missing Link Tibetan Groups Targeted with 1-Click Mobile Exploits. 2019. URL: https://citizenlab.ca/2019/09/poison-carp-tibetan-groups-targeted-with-1-click-mobile-exploits/ (pp. 419, 445, 448).

[55] Rene Mayrhofer, Jeff Vander Stoep, Chad Brubaker, Dianne Hackborn, Bram Bonné, Güliz Seray Tuncay, Roger Piqueras Jover, and Michael Specter. The Android Platform Security Model (2023). In: arXiv:1904.05572 (2024) (pp. 413, 416).

[56] Man Yue Mo. Corrupting memory without memory corruption. 2022. URL: https://github.blog/security/vulnerability-research/corrupting-memory-without-memory-corruption/ (pp. 412, 428, 429, 445).

[57] Man Yue Mo. Fall of the machines: Exploiting the Qualcomm NPU (neural processing unit) kernel driver. 2021. URL: https://github.blog/security/vulnerability-research/fall-of-the-machines-exploiting-the-qualcomm-npu-neural-processing-unit-kernel-driver/ (pp. 428, 429, 445).

[58] Man Yue Mo. Gaining kernel code execution on an MTE-enabled Pixel 8. 2024. URL: https://github.blog/security/vulnerability-research/gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/ (pp. 412, 428, 429, 445).

[59] Man Yue Mo. One day short of a full chain: Part 1 - Android Kernel arbitrary code execution. 2021. URL: https://securitylab.github.com/research/one_day_short_of_a_fullchain_android/ (pp. 412, 428, 429, 445).

[60] Andy Nguyen. CVE-2021-22555: Turning x00x00 into 10000$. 2021. URL: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html (p. 434).

[61] Lau Notselwyn. Flipping Pages: An analysis of a new Linux vulnerability in nf_tables and hardened exploitation techniques. 2024. URL: https://pwning.tech/nftables/ (pp. 413, 433, 434, 436).

[62] Donncha O'Cearbhaill and Bill Marczak. Exploit archaeology a forensic history of in the wild NSO Group exploits. In: Virus Bulletin Conference. 2022 (pp. 415, 418, 445).

[63] Maxime Peterlin. Reversing and Exploiting Samsung's Neural Processing Unit. 2021. URL: https://blog.longterm.io/samsung_npu.html (p. 445).

[64] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html (p. 433).

[65] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In: S&P. 2021 (p. 416).

[66] Eloi Sanfelix. A bug collision tale. 2020. URL: https://labs.bluefrostsecurity.de/files/OffensiveCon2020_bug_collision_tale.pdf (pp. 433, 445).

[67] Blue Frost Security. Exploiting CVE-2020-0041 - Part 2: Escalating to root. 2020. URL: https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/ (pp. 433, 445).

[68] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: S&P. 2016 (p. 438).

[69] Maddie Stone. 2022 0-day In-the-Wild Exploitation...so far. 2023. URL: https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html (p. 445).

[70] Maddie Stone. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain. 2022. URL: https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html (pp. 418, 419).

[71] Maddie Stone. Bad Binder: Android In-The-Wild Exploit. 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html (pp. 419, 448).

[72] Maddie Stone. Déjà vu-lnerability – A Year in Review of 0-days Exploited In-The-Wild in 2020. 2021. URL: https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html (p. 445).

[73] Maddie Stone. Detection Deficit: A Year in Review of 0-days Used In-The-Wild in 2019. 2020. URL: https://googleprojectzero.blogspot.com/2020/07/detection-deficit-year-in-review-of-0.html (p. 445).

[74] Maddie Stone. In-the-Wild Series: Android Post-Exploitation. 2021. URL: https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-post-exploitation.html (pp. 419, 448).

[75] Maddie Stone. The More You Know, The More You Know You Don't Know. 2022. URL: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html (pp. 412, 418, 445).

[76] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022. 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html (pp. 412, 418, 445, 448).

[77] Maddie Stone, Jared Semrau, and James Sadowski. We're All in this Together: A Year in Review of Zero-Days Exploited In-the-Wild in 2023. 2024. URL: https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf (pp. 412, 418, 428, 445, 448).

[78] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo Workarounds for Kernel Bugs. In: USENIX Security. 2021 (p. 444).

[79] Zi Fan Tan, Gulshan Singh, and Eugene Rodionov. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938. 2024. URL: https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938%5C#unlink-primitive (p. 433).

[80] Yong Wang. Ret2page: The Art of Exploiting Use-Afer-Free Vulnerabilities in the Dedicated Cache. 2022. URL: https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf (p. 434).

[81] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel. In: USENIX Security. 2023 (p. 444).

[82] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In: AsiaCCS. 2019 (p. 445).

[83] Le Wu, Xuen Li, and Tim Xia. ExplosION: The Hidden Mines in the Android ION Driver. 2022. URL: https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Wu-ExplosION-The-Hidden-Mines.pdf (p. 446).

[84] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way! 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf (p. 445).

[85] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html (pp. 412, 413, 433, 434, 436, 445).

[86] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: USENIX Security. 2019 (p. 434).

[87] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In: USENIX Security. 2018 (pp. 413, 432–434, 436).

[88] Xuan Xing, Eugene Rodionov, Jon Bottarini, Adam Bacchus, Amit Chaudhary, Lyndon Fawcett, and Joseph Artgole. Google & Arm - Raising The Bar on GPU Security. 2024. URL: https://security.googleblog.com/2024/09/google-arm-raising-bar-on-gpu-security.html (pp. 412, 418, 428, 445).

[89] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (p. 434).

[90] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic Hot Patch Generation for Android Kernels. In: USENIX Security. 2020 (pp. 416, 444).

[91] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In: USENIX Security. 2022 (p. 434).

[92] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In: CCS. 2023 (pp. 432–434, 436).

[93] Google Project Zero. Qualcomm KGSL: reclaimed / in-reclaim objects can still be mapped into VBOs. 2024. URL: https://project-zero.issues.chromium.org/issues/42451701 (p. 429).

[94] Hang Zhang and Zhiyun Qian. Precise and Accurate Patch Presence Test for Binaries. In: USENIX Security. 2018 (pp. 437, 438, 444).

[95] Ye Zhang, Le Wu, Shupeng Gao, and Zheng Huang. Attacking NPUs of Multiple Platforms. 2023. URL: https://i.blackhat.com/EU-23/Presentations/EU-23-Zhang-Attacking-NPUs-of-Multiple-Platforms.pdf (p. 445).

[96] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An Investigation of the Android Kernel Patch Ecosystem. In: USENIX Security. 2021 (pp. 416, 419, 445, 448).

12. Appendix

Table 11.7: Devices which are susceptible to n-day vulnerabilities with the commit message shown in Listings 11.6 to 11.10. The symbol ✓ refers to susceptible, † refers to fixed in October 2024, †† refers to fixed in November 2024, and ☆ refers to not tested.

Vendor	**Device**	**Model**	**OOB read**	**UAF access**			**ID**
			vuln1 (see 11.6)	**vuln2** (see 11.7)	**vuln3** (see 11.8)	**vuln4** (see 11.9)	**vuln5** (see 11.10)
Samsung	Galaxy S23	SM-S911B	†	†	†	†	†
	Galaxy S23+	SM-S916B	†	†	☆	†	†
	Galaxy S23 Ultra	SM-S918B	†	†	☆	†	†
	Galaxy Z Flip5	SM-F731B	†	†	☆	†	†
	Galaxy Z Fold5	SM-F946B	†	†	☆	†	†
Xiaomi	Mix Fold 3	Babylon	✓	✓	☆	✓	✓
	Redmi K70, Poco F6 Pro	Vermeer	††	✓	☆	††	✓
	13 Ultra	Ishtar	✓	✓	☆	✓	✓
	13 Pro	Nuwa	††	✓	☆	††	✓
	13	Fuxi	††	✓	☆	††	✓
Asus	ROG Phone 7 Ultimate, ROG Phone 7	AI2205	✓	✓	☆	✓	✓

```
commit a6b25a6b8b9d1d6dfe1eb743ee39de21485d66da
Author: Ramesh Nallagopu <quic_rnallago@quicinc.com>
Date:   Fri Jun 28 22:17:36 2024 +0530

    dsp-kernel: Fix to avoid untrusted pointer dereference

    Currently, the compat ioctl call distinguishes itself
    using a global flag. If a user sends a compat ioctl call
    followed by a normal ioctl call, it may result in using a
    user passed address as a kernel address in the
    fastrpcdriver. To address this issue, consider localizing
    the compat flag for the ioctl call.

```

Listing 11.6: **Vuln1:** Git commit message of the OOB read vulnerability CVE-2024-21455 [32].

```
commit 6dab51a3af6f217c1729452fa963d0d3568058ec
Author: Abhishek Singh <quic_abhishes@quicinc.com>
Date:   Tue Mar 5 17:19:52 2024 +0530

    dsp-kernel: use-after-free (UAF) in global maps

    Currently, remote heap maps get added to the global list
    before the fastrpc_internal_mmap function completes the
    mapping. Meanwhile, the fastrpc_internal_munmap function
    accesses the map, starts unmapping, and frees the map
    before the fastrpc_internal_mmap function completes,
    resulting in a use-after-free (UAF) issue. Add the map to
    the list after the fastrpc_internal_mmap function
    completes the mapping.

```

Listing 11.7: **Vuln2:** Git commit message of the first UAF vulnerability CVE-2024-33060 [34].

```
commit 2096d42a680640f9fcc02272bf58f9cc7fa74576
Author: ANANDU KRISHNAN E <quic_anane@quicinc.com>
Date:   Wed Aug 14 10:39:55 2024 +0530

    msm: adsprpc: Avoid taking reference for group_info

    Currently, the get_current_groups API accesses group info,
    which increases the usage refcount. If the IOCTL using the
    get_current_groups API is called many times, the usage
    counter overflows. To avoid this, access group info
    without taking a reference. A reference is not required as
    group info is not released during the IOCTL call.

```

Listing 11.8: **Vuln3:** Git commit message of second UAF vulnerability CVE-2024-38402 [19].

```
commit f98ae73093949e9e12f64f28bd6103b5f941d32e
Author: Santosh <quic_ssakore@quicinc.com>
Date:   Tue Nov 19 10:54:19 2024 +0530

    dsp-kernel: Add attribute and flag checks during map
    creation

    A persistence map is expected to hold refs=2 during its
    creation. However, the Fuzzy test can create a persistence
    map by configuring a mismatch between attributes and flags
    using the KEEP MAP attribute and FD NOMAP flags. This sets
    the map reference count to 1. The user then calls
    fastrpc_internal_munmap_fd to free the map since it doesnt
    check flags, which can cause a use-after-free (UAF) for
    the file map and shared buffer. Add a check to restrict
    DMA handle maps with invalid attributes.

```

Listing 11.9: **Vuln4:** Git commit message of the third UAF vulnerability CVE-2024-49848 [30].

```
commit 29cbad25d9bf36341131dcc7dfff75b4255d2111
Author: Abhishek Singh <quic_abhishes@quicinc.com>
Date:   Fri Jun 21 16:04:09 2024 +0530

    dsp-kernel: Do not search the global map in the process-
    specific list

    If a user makes the ioctl call for the
    fastrpc_internal_mmap with the global map flag, fd, and va
    corresponding to some map already present in the process-
    specific list, then this map present in the process-
    specific list could be added to the global list. Because
    global maps are also searched in the process-specific
    list. If a map gets removed from the global list and
    another concurrent thread is using the same map for a
    process-specific use case, it could lead to a use-after-
    free. Avoid searching the global map in the process-
    specific list.

```

Listing 11.10: **Vuln5:** Git commit message of the ID vulnerability CVE-2024-33060 wrongly assigned [31].

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.