Uncovering New Classes of Kernel Vulnerabilities

Jakob Koschel

VRIJE UNIVERSITEIT

UNCOVERING NEW CLASSES OF KERNEL VULNERABILITIES

PH.D. THESIS

Jakob Koschel

The research reported in this dissertation was conducted at the Faculty of Science, at the Department of Computer Science, of the Vrije Universiteit Amsterdam.

This work was supported by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788.

Cover design by Daniel Bornmann & Lennard Makosch

VRIJE UNIVERSITEIT

UNCOVERING NEW CLASSES OF KERNEL VULNERABILITIES

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op donderdag 30 januari 2025 om 13.45 uur
in een bijeenkomst van de universiteit,
De Boelelaan 1105

door

Jakob Koschel

geboren te Aalen, Duitsland

promotor:	prof.dr.ir. H.J. Bos
copromotor:	dr. C. Giuffrida
promotiecommissie:	dr. I. Malavolta
	dr. S. Shinde
	dr. O. Oleksenko
	prof.dr. V. Moonsamy
	prof.dr. T. Holz

Acknowledgements

My journey started with coming to Vrije Universiteit Amsterdam for my graduate studies. Very fast after arriving, I decided to shift my focus from my original track in Internet and Web Technology to Computer Security. All credit goes to Herbert Bos for sharing his excitement for security and great teaching to motivate myself and others to learn more about security. After my second year of graduate studies, I had a great time working on research projects with Kaveh, Cristiano and Herbert and my original goal of *never doing a PhD* turned out to be more than wrong. On one of my last days before graduating, I was approached by all three individually, with the same question: "So when are you starting your PhD?". Before even knowing myself, it turns out they were all right, and their plan worked out. I am glad to have gone on the path of pursuing my PhD and want to thank all of you for believing in me.

Thank you, Herbert, for being such a great supervisor, sparking the joy of security in the next generations of students, and always being there when support was needed. I have learned a lot from you from the way to approach research to becoming better in academic writing.

Cristiano was the best mentor and copromotor I could have asked for. Thank you for all the inspiration and infinite source of great, but often crazy, ideas. You were always there, when I needed someone to discuss new ideas or good advice. I am truly grateful for you, sharing your incredible systems knowledge, and motivating myself in desperate times when I was close to dropping an almost finished project.

Unfortunately, Kaveh's and my path didn't overlap during my time as a PhD student. However, I still want to thank you for all the time you took to discuss projects and crazy ideas when I was still a master's student. No matter what, you were always in a good mood and pitching great ideas. While you, unfortunately, left before I joined for my PhD, you were a big part of motivating myself to pursue my PhD.

Next, I would like to thank my reading committee, Thorsten Holz, Ivano Malavolta, Veelasha Moonsamy, Oleksii Oleksenko and Shweta Shinde, for taking their valuable time to review my thesis and their suggestions for improvement.

Brian, thank you for the great teamwork Kasper has been. We have both put sweat and tears into the project and I am grateful of the way it turned out. It was essential to work on it together to stay motivated and find the best possible way forward.

Beginning of 2023, Pietro Borrello joined VUSec for a few months and Cristiano asked if he can pitch the project idea we just finalized to Pietro. I was sure nobody would be crazy enough to pick that project out of all. Turns out luckily I was wrong and Pietro was crazy enough. Thank you for doing the project together and being such a very talented and creative teammate.

There are plenty more that make the VUSec group a great place to work and laugh together. Thank you, Alvise, Alyssa, Andrea D.D., Andrea J., Andrei, Asia, Bala, Ben, Dave, Elia, Emanuele, Enrico, Erik B., Erik vdK., Floris, Hany, Johannes, Koen, Klaus, Manuel, Marius, Mathé, Pietro, Raphael, Robin, Saideh, Sander, Sebastian, Taddeüs, Victor D., Victor vdV., and Mojca and Cynthia for shielding me from the university's bureaucracy.

I am grateful to Anil Kurmus for inviting me to join their systems security group at IBM Research in Zurich for an internship. Thank you for all your support both professionally and personally. You want out of your way helping me to have a great time in Zurich, lending me an awesome vintage road bike, taking me on a "rafting" tour on the river, and inviting me to a full day of board games with your friends. Thank you, Andrea Mambretti, and Alessandro Sorniotti for a great time at IBM Research.

Next, I am grateful to KP Singh for inviting me to intern with the kernel security team at Google in Zurich. I know you were moving stones to make this possible, which I deeply appreciate. I want to thank all the other members of the kernel security team and additionally Alexandra Sandulescu, Eduardo A. Vela N, Jordy Zomer, Matteo Rizzo, and Andy Nguyen.

Personally, I am grateful for all the friends that supported and distracted me during my PhD journey. Thank you for the all the wonderful memories in Amsterdam: Caro, David, Johnny, Linda, and Selam. Also thanks to Johannes for hosting me in Zurich and the extended time in Los Angeles and Nico for the countless online board game nights. Thanks to my friends Simon and Paul for the good moments whenever we see each other. Caro, you were the best friend and flatmate I could have imagined. Thank you for making our home such a great place. I want to extend my gratitude to my parents, Gabi and Knut, thank you for supporting any choice I made and having faith in me. Lastly, I would like to thank Hanna, for all her support and pushing me to actually start this journey. Without your love and patience I could not have done it. You mean everything to me.

Jakob Koschel
Amsterdam, The Netherlands, October 2024

Contents

Acknowledgements **vii**

Contents **ix**

List of Figures **xi**

List of Tables **xii**

Publications **xv**

1 Introduction **1**

1.1 Kernel defenses 2
1.2 Bug Discovery in the kernel 3
1.3 Goals 4
1.4 Contributions 5

2 TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs **7**

2.1 Introduction 8
2.2 Background 9
2.3 Threat Model 11
2.4 Attack Overview 12
2.5 Reverse Engineering the TLB 13
2.6 TagBleed 15
2.6.1 Forcing Kernel Memory Access 15
2.6.2 Leaking Through Tagged TLB 15
2.6.3 Confused Deputy Attack with AnC 16
2.6.4 Derandomizing Kernel Modules 18
2.6.5 Derandomizing Physmap 18
2.7 Evaluation 18
2.7.1 Side channel by TLB set eviction 18
2.7.2 Side channel by cache line eviction 19
2.7.3 Combining the two side channels 20
2.7.4 Success rate 21

2.7.5 Attack time 22
2.7.6 Noise 22
2.7.7 Comparison against other KASLR attacks 23
2.8 Mitigations 23
2.9 Related Work 25
2.10 Conclusions 27

3 Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel 29
3.1 Introduction 30
3.2 Background 32
3.2.1 Speculative execution attacks and defenses 32
3.2.2 Gadget scanning 34
3.3 Threat Model 35
3.4 Problem Analysis 35
3.5 Overview 36
3.6 Speculative Emulation 37
3.6.1 Transactions and rollbacks 38
3.6.2 Challenges unique to the kernel 39
3.7 Taint Policies 40
3.7.1 Vulnerability detectors 41
3.7.2 Injection policies 42
3.7.3 Access policies 43
3.7.4 Leakage policies 45
3.8 Implementation 46
3.9 Evaluation 47
3.9.1 Comparison with previous solutions 47
3.9.2 Gadgets found in the kernel 50
3.10 Limitations 50
3.11 Case Study 51
3.11.1 List implementation of the kernel 52
3.11.2 A `list_for_each_entry` gadget in `keyring.c` 52
3.11.3 Exploitation 53
3.12 Related Work 54
3.13 Conclusion 56
Appendix 3.A Coverage Evaluation 56
3.A.1 Normal execution coverage 56
3.A.2 Speculative emulation coverage 57
Appendix 3.B Performance Evaluation 59
Appendix 3.C Large-Scale Exploitability Evaluation 60
Appendix 3.D Additional Case Study 61
Appendix 3.E Developer Interface 62

4 uncontained: Uncovering Container Confusion in the Linux Kernel 65
4.1 Introduction 66
4.2 Background 67
4.2.1 Type Confusion Bugs in C++... *and in C* 67
4.2.2 Sanitizers 69
4.3 Container Confusion in the Linux Kernel 69
4.3.1 Security Implications 69
4.3.2 Running Example 70
4.3.3 Type Graph Complexity 71
4.4 UNCONTAINED Overview 72
4.5 Container Confusion Sanitizer 73
4.5.1 Design 74
4.5.2 Implementation 77
4.5.3 Evaluation 78
4.6 Retrospective Analysis and Bug Patterns 79
4.7 Static Analyzer 83
4.7.1 Design 83
4.7.2 Implementation 86
4.7.3 Evaluation 87
4.8 Discussion 88
4.9 Related Work 89
4.10 Conclusion 90
Appendix 4.A Assigned CVEs 91
Appendix 4.B LMbench Evaluation 91
Appendix 4.C Static Analysis Rules 91

5 Conclusion 95
5.1 Future directions 96

6 Contributions to papers 99

References 101

Summary 115

List of Figures

2.1 High level overview of TagBleed 12
2.2 TLB sets for 256 virtual consecutive huge pages 14
2.3 Overview of virtual address bits used by TLB indexing bits & KASLR randomization 16
2.4 Slowdown of an invalid system call when flushing TLB sets 19
2.5 Sliding signal using a kernel module 20
2.6 System call slowdown evicting cache lines 21
2.7 EVICT+TIME side channel accessing a user-level huge page 22
2.8 EVICT+TIME side channel on the invalid system call 23

3.1 Essential steps in a Spectre attack 31
3.2 Components of KASPER mapped to the essential steps of a Spectre attack 37
3.3 Overview of taint policies to detect gadgets 41
3.4 Workflow of KASPER 46
3.5 Speculative type confusion for the `find_keyring_by_name` gadget . . . 54
3.6 Gadgets found by KASPER over time 56
3.7 KASPER covered edges fuzzing with syzkaller 57
3.8 Calltraces for gadgets found with KASPER 60
3.9 Speculative lengths for gadgets found with KASPER 61
3.10 Screenshot of KASPER's web interface 63

4.1 Type graph of `container_of` instances 73
4.2 UNCONTAINED's redzone layout 74
4.3 Distribution of `container_of` invocations 76

List of Tables

2.1 KASLR entropy in Linux 10
2.2 TLB architecture for Skylake architecture 15
2.3 Microarchitectures affected by TagBleed 18
2.4 KASLR attacks vs. defenses 24

3.1 Overview of Spectre gadget primitives 32
3.2 Exploitability of Spectre variants 36
3.3 Lines of code for KASPER 47
3.4 Kocher's Spectre samples detection by KASPER 48
3.5 Cache gadgets reported by KASPER running `ls` 49
3.6 Gadgets discovered by KASPER 50
3.7 Causes for speculative rollbacks in KASPER 58

4.1 Reports found by UNCONTAINED's static analyzer 87
4.2 CVEs found by UNCONTAINED 91
4.3 LMbench results of UNCONTAINED 92
4.4 Static analysis rules of UNCONTAINED 93

Publications

This dissertation includes several research papers, as appeared in the following conference proceedings. The text differs from the published versions in minor editorial changes made to improve readability:

Jakob Koschel, Cristiano Giuffrida, Herbert Bos and Kaveh Razavi. **TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs.** In *Proceedings of the Fifth IEEE European Symposium on Security and Privacy (EuroS&P '20).* September 7–11, 2020, all-digital.
[Appears in Chapter 2]

Jakob Koschel[1], Brian Johannesmeyer[1], Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. **KASPER: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel.** In *Proceedings of the Twenty-Ninth Network and Distributed System Security Symposium (NDSS '22).* April 24–28, 2022, San Diego, USA.
[Appears in Chapter 3]

Jakob Koschel[1], Pietro Borrello[1], Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. **UNCONTAINED: Uncovering Container Confusion in the Linux Kernel.** In *Proceedings of the Thirty-Second USENIX Security Symposium (USENIX '23).* August 9–11, 2023, Anaheim, USA.
[Appears in Chapter 4]

[1]Equal contribution shared first authors. Refer to Chapter 6 for contributions per author.

1 | Introduction

In today's world there is barely any computing without operating systems. Historically, electronic computers were developed in early 1940s. Back then, computers were designed and used for dedicated mathematical computation tasks written directly in machine language. Such programs were entered into the computer using punched paper cards or tape, the user had full control over the machine and only one program ran at a given time. A program either ran to completion or to failure, crashing the machine. This overall resulted in the need for the first operating system created in the early 1950s to submit batches of tasks and cleanup and transfer control to the next on completion. With better hardware, in the late 1960s, the next generation of operating systems was capable to perform multiple tasks simultaneously, implementing a fundamental task of operating systems: scheduling tasks.

Modern computing came a long way in the last 60 years, from mainframes as big as a room to personal computers in almost all electronic devices, interacting with a wide variety of hardware, such as monitors, network, printers and other peripherals. With the complexity of today's world, the kernel of the operating system of personal computers is responsible for managing and securing access to hardware and for isolating processes while providing access to computation resources. In order to perform such tasks, the operating system usually runs at a higher privilege level compared to normal user-space programs and has full control of the CPU and memory. This makes the kernel a primary target for attacks. Compromising the operating system breaks the isolation between processes and allows an attacker to gain control of almost the entire system. Computers are much more versatile nowadays, requiring drivers for all kind of peripherals, different architectures and use cases such as mobile phones, desktop computers or servers. With such complexity, modern kernels like the Linux kernel have grown to tens of millions lines of code.

It goes without saying that software developers are not flawless, especially with complex systems such as the kernel. It is therefore inevitable to have bugs in such large code bases. Some of those bugs can be elevated into security vulnerabilities. Commonly found software vulnerabilities in the kernel are buffer overflows, use-after-frees, race conditions or uninitialized memory use [192]. Judging by the numbers of potential vulnerabilities reported by syzbot [51], a continuous kernel fuzzer, the number of known vulnerabilities is limited by our current detection capabilities, indicating the actual number of undiscovered vulnerabilities is multiple orders of magnitudes larger. The kernel, however, is not only susceptible to attacks using software vulnerabilities but also those using hardware based vulnerabilities such as side-channel attacks. Such side-channel vulnerabilities, demonstrated against crypto implementations as early as 1996 by Paul C. Kocher [96], allow an attacker to leak sensitive information by timing cache accesses from a different security domain. The kernel has also been a target of such side-channel vulnerabilities [59, 71], often to break the randomization of the kernel location in virtual memory. In 2018, a new type of vulnerabilities demonstrated the use of transient execution to leak sensitive information. While some vulnerabilities, such as Meltdown [108] were mitigated in new hardware generations, others such as Spectre [95] are fundamental to the way transient execution is implemented on modern CPUs. In order to leak sensitive data, an attacker looks for vulnerable code gadgets, similar to an attacker looking for software vulnerabilities.

Kernel exploits have demonstrated how to turn such hardware or software vulnerabilities into privilege escalation or leaking sensitive kernel data. For instance, Dirty COW (CVE-2016-5195), a local privilege escalation vulnerability in the Linux kernel, existed for 10 years in the code base until it was fully fixed. It is a prime example of a vulnerability that was used by attackers in the wild. Another use case of kernel exploits is to remove software restrictions imposed by manufacturers. For example, iPhones run the iOS operating system, which restricts the possibility to run your own software freely on your device. Kernel exploits have gained a lot of publicity since it enables *jailbreaking* such restricted devices by patching parts of the kernel. While personal machines remain attractive targets, e.g., for ransomware attacks that encrypt the victim's system until the attacker receives a ransom payment, machines in the cloud also became an attractive attack target. Anyone is able to execute untrusted code *in the cloud*, often on servers co-scheduled with other customers' processes [3]. Exploiting the kernel or hypervisor managing the virtual machines, breaks the guaranteed isolation between customers. In particular, hardware based side-channel attacks, such as Spectre, are dangerous in the cloud, allowing an attacker to leak customer data through shared hardware resources.

1.1 Kernel defenses

With such high impact exploits, protecting the kernel became significantly more important in the last decades. Many defenses originally designed for user-space programs made their way into the kernel. One such example is Kernel Address Space Randomization (KASLR), randomizing the location of the kernel in virtual memory, requiring an additional information disclosure step in most exploits. Kernel data is also mapped as non-executable, to prevent easy ways to inject shellcode. Other mitigations are more specific to the kernel such as Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) that protect against accessing or executing user-space memory without explicit permission. With recent Intel hardware, the Linux kernel supports hardware assisted control-flow integrity (CFI) with Intel CET, or even fine-grained CFI with FineIBT. CFI helps to fend off return-oriented programming (ROP) attacks. Similar to user-space heap mitigations, the kernel adapted similar approaches to defend its heap. One such mitigation, `SLAB_VIRTUAL`, forces a virtual memory region to be only used by one object type, to avoid type confusions caused by use-after-free vulnerabilities.

Transient execution attacks [8, 22, 95, 99, 109, 118, 122, 142, 150, 174, 176, 177, 181, 191] require a completely different and unique defense strategy. Most of these defenses [11, 90, 92, 100, 151, 168, 194, 197, 200] are specific for a subset of currently known attacks. One of the first defenses is Kernel Page Table Isolation (originally presented as KAISER [58] to protect the kernel against breaking KASLR with side-channel vulnerabilities). It isolates the kernel into its own address space, separate from the address space of user-space processes and was originally deployed against Meltdown. Since these attacks are fundamental to how the CPU operates, they could not be fixed by hardware updates, in the form of microcode updates, requiring mitigation in the kernel. Some attacks such as RIDL [181] or Retbleed [191] were mitigated by flushing specific microarchitectural buffers and untraining certain branch predictors at privilege boundaries. With microcode updates, CPU manufacturers allowed the kernel to enable Indirect Branch Restricted Speculation (IBRS) to prevent lower privilege branch predictions to influence the branch prediction state in the kernel. Other attacks such as Spectre v1 [95] are fundamental to every conditional branch and can only be mitigated on the spot by fencing detected gadgets with speculation barriers. Often fixing transient execution attacks in the kernel is the last resort as it also comes with insignificant performance impact [55], which results in many mitigations not being deployed in practice until exploitation is demonstrated.

1.2 Bug Discovery in the kernel

Since many existing defenses are impractical for production systems due to the high performance overhead, finding and fixing bugs remains an important task to protect systems with insufficient mitigations. However, finding bugs in a code base such as the Linux kernel with more than 30 million lines of code, is a challenge in its own. Such large code bases with around 10,000 patches for each new version cannot rely on manual efforts to find bugs. Such a massive and fast changing code base requires automated bug discovery tools. The most trivial bugs can often be spotted with static analysis [18, 112, 146, 189], e.g. the format string bug. However, more complex, interprocedural or temporal bugs are difficult to determine statically.

Fuzzing, an automated software testing technique, can often find such bugs by feeding the program, in this case the kernel, with unexpected, pseudo-random input to monitor crashes or possible memory corruptions. In recent years, multiple fuzzers dedicated to finding bugs in the kernel were developed [52, 149]. These fuzzers however focus on the most common and established bug types, such as architectural spatial memory errors (e.g., out-of-bounds) and temporal errors (e.g., use-after-free). Fuzzers facilitate sanitizers, runtime bug detectors, to spot specific bug types.

Specifically for the kernel, there is a range of such sanitizers. KASAN [98] focuses on detecting out-of-bounds accesses, use-after-free bugs, and double (or invalid) free operations. Other kernel sanitizers are KMSAN [139] detecting uninitialized memory use, KCSAN [41] detecting data races, and UBSAN [114] to detect various types of undefined behavior. While this covers the most basic types of software vulnerabilities, more insidious software bugs or even hardware issues are currently undetected by fuzzing.

Even with the available sanitizers, current efforts to fix reported bugs often fall short and are lacking, resulting in many bugs discovered by fuzzers unpatched for a long time [183]. But not only such *simple* bugs are worrying, what about new types of bugs, currently not even considered by state-of-the-art fuzzers? Detecting hardware bugs, in particular, is often done with manual inspection assisted by simple static analysis, suffering from a high false positive rate [25].

1.3 Goals

In this dissertation, we demonstrate that it is not enough to focus on currently established bug patterns, but repeatedly need to explore new classes of bugs in all possible areas of systems security, ranging from side-channel attacks to new types of software-based type-confusion bugs. Only by exploring such new angles of exploitation and including them in bug detection systems, we can slowly turn the kernel, currently a pile of sand, into a solid foundation for computing in modern systems.

We set the goal for this thesis to uncover new types of vulnerabilities for each of the major class affecting the kernel: software bugs, side channels, and transient execution vulnerabilities.

For side-channel vulnerabilities, research focuses on finding new means of leakage, such as the often used branch prediction present in modern CPUs [95] or one of the various other prediction buffers like the Return Stack Buffer (RSB) [191]. With this thesis, we explore the possibility to revisit existing side-channels and if a combination of those can find a new application to attack e.g., the kernel. AnC [54], for example, used caching of page tables during a page table walk to break ASLR in user space. In order to make this effective, the attacker required a large range of virtual contiguous memory, difficult to archive targeting the randomization of the kernel text mapping. Our goal is to find a combination with another side-channel to remove such requirements and make it applicable against the kernel. We believe such combinations create a new type of vulnerability that on its own were not possible in the scenario of a user attacking the kernel.

On the transient execution vulnerabilities front, many new types of vulnerabilities were presented in recent years. Some of them, such as Spectre v1, are impossible to mitigate entirely without suffering a big performance impact. Instead, they are only risky in very specific code patterns, often called *gadgets*. Identifying gadgets in large code bases such as the Linux kernel is far from trivial. Manual and static analysis is both error-prone and suffers from a large amount of false positives. This thesis aims to find gadgets in the Linux kernel and improve on accuracy in gadget finding and generalize different transient execution vulnerabilities to easily extend the scanner with newly discovered vulnerabilities.

On the software bug front, the security community currently focuses on the most obvious bug types. Originally, this included buffer overflows and use-after-free bugs, detected by KASAN. Later the search was expanded to prominent issues in the kernel such as uses of uninitialized values (KMSAN) and detecting data races (KCSAN). While this covers a majority of bugs in the kernel, it is also a vicious circle since their prevalence is also highly accelerated with better detectors. We want to explore new bug patterns that are currently not covered by existing automatic detection tools and build new detectors. One such issue we analyzed are type confusion bugs, unique to large C code bases such as the Linux kernel, that were previously overlooked. We discover and pinpoint these specific patterns to build a sanitizer to detect them with continuous fuzzing.

State-of-the-art fuzzers only focus on well-known bug types and still find too many bugs to fix. Therefore, rather than building secure systems on a solid foundation, we build our solutions on a pile of sand. In order to provide secure systems we need to continuously improve kernel security to strengthen the foundation into something solid enough for the software stack to rely on. Since our understanding of both traditional software and recent microarchitectural attack vectors is based on recent developments in OS structure and attack mitigation efforts, we ask whether our understanding is sufficiently complete to serve as a foundation for secure systems. More precisely, we ask whether the kernel is sufficiently protected against new combinations of side channels, and whether our mental model of transient execution attacks and traditional software errors such as type confusions are comprehensive enough or can still be bypassed using new types of attacks that fall outside our current attackers' model.

1.4 Contributions

This dissertation makes several contributions, with results published in refereed conferences (Page xv). We performed minor editorial changes to improve readability. The remainder is organized as follows:

In **Chapter 2** we present TagBleed, a new way to leak secret kernel information using a side-channel attack despite deployed mitigations. We demonstrate that the addition of a new hardware feature to make kernel mitigations more performant, creates a new attack surface leaking the randomized memory location of the kernel. TagBleed combines two individual side channels to demonstrate a successful attack against the Linux kernel. Chapter 2 appeared in *Proceedings of the Fifth IEEE European Symposium on Security and Privacy (EuroS&P '20)*.

In **Chapter 3** we present KASPER, a scanner for transient execution gadgets in the kernel. The kernel was sparsely mitigated against such transient execution attacks. The previous efforts required manual inspection, difficult for a complex and large scale codebase like the kernel. KASPER automates the process to find possible code gadgets in the kernel that can be utilized in transient execution attacks. Additionally, KASPER is able to scan for new types of transient execution gadgets, missed by previous mitigation efforts. Chapter 3 appeared in *Proceedings of the Twenty-Ninth Network and Distributed System Security Symposium (NDSS '22)*.

In **Chapter 4** we present UNCONTAINED, finding so-called *container confusion* bugs in the kernel. This new type of type confusion bug is caused by the way the programming language C mimics object-oriented programming features. Many of the discovered bugs were missed by the state-of-the-art memory error detectors. Chapter 4 appeared in *Proceedings of the Thirty-Second USENIX Security Symposium (USENIX '23)*.

We conclude in **Chapter 5** by summarizing our key results, and look at future directions for new combinations of side channels, transient execution gadget searching, and more generic type confusion detection for C.

Chapter 6 presents the individual author contributions for all included papers.

2 TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs

Kernel Address Space Layout Randomization (KASLR) has been repeatedly targeted by side-channel attacks that exploit a typical unified user/kernel address space organization to disclose randomized kernel addresses. The community has responded with kernel address space isolation techniques that separate user and kernel address spaces (and associated resources) to eradicate all existing side-channel attacks.

In this chapter, we show that kernel address space isolation is insufficient to harden KASLR against practical side-channel attacks on modern tagged TLB architectures. While tagged TLBs have been praised for optimizing the performance of kernel address space isolation, we show that they also silently break its original security guarantees and open up opportunities for new derandomization attacks. As a concrete demonstration, we present TagBleed, a new side-channel attack that abuses tagged TLBs and residual translation information to break KASLR even in the face of state-of-the-art mitigations. TagBleed is practical and shows that implementing secure address space isolation requires deep partitioning of microarchitectural resources and a more generous performance budget than previously assumed.

2.1 Introduction

Kernel-level Address Space Randomization (KASLR) is a first line of defense against adversaries that aim to exploit software vulnerabilities in the kernel for escalating their privilege. While KASLR raises the bar for attackers, previous work has shown many different possibilities for side-channel attacks that can easily bypass KASLR [44, 59, 71, 81]. These attacks exploit the unified kernel and user address spaces that is exposed through various microarchitectural resources.

To stop these attacks, recent mitigations propose separating kernel and user address spaces on these microarchitectural resources [47, 58]. While secure, these mitigations would be expensive without tagged Translation Lookaside Buffer (TLB) available on all modern Intel processors. Tagging the TLB significantly reduces the overhead of these mitigations by avoiding TLB flushes on every privilege switch which is now necessary. As a result, tagged TLB is praised for enabling deployment of such mitigations in practice [58, 169]. After the public disclosure of speculative execution attacks [94, 109, 176, 181], major operating systems such as Linux and Windows turned these mitigations on-by-default.

In this chapter, we show that while separating kernel and user address spaces mitigates some of the speculative execution attacks, they do not fulfill their original goal of protecting against side-channel attacks on KASLR. Ironically, tagged TLB, the optimization that makes separating kernel and user address spaces efficient, re-enables the sharing of the TLB entries across kernel and user address spaces. Our proof-of-concept exploit, TagBleed, uses the new leakage introduced by this sharing to fully break KASLR in spite of these deployed mitigations.

KASLR Attacks & Defenses Existing side-channel attacks against KASLR [44, 59, 71, 81] target shared microarchitectural resources to derive information about secret locations in the virtual address space where kernel memory resides. These attacks are possible due to *unified* kernel and user address spaces supported by the CPU for reasons of efficiency. For instance, a unified virtual address space between a user process and the kernel allows the user process to measure timing differences of a triggered CPU exception when accessing a kernel address to determine whether that address is mapped, breaking KASLR [71, 81]. Similar attacks are also possible without even triggering an exception: by measuring the execution time of the prefetch instruction for kernel addresses one can probe the existence of address translation data structures in the CPU's translation caches [59]. This unification of address spaces even extends to microarchitectural resources such as the Branch Target Buffer (BTB), making it possible for attackers to find out the virtual address space of branch targets in the kernel from user space [44].

To mitigate this class of attacks, recent proposals advocate for isolation of the kernel address space. This can be enforced by explicitly flushing microarchitectural resources such as the BTB on privilege switches [47] (or implicitly flushing them with hardware mitigations such as eIBRS [73]) and placing the kernel memory on a separate address space [47, 58]. This stops the attackers' ability for probing the existence of kernel addresses from a user process. On every privilege switch (e.g., due to a system call) the address space translation structures cached by the CPU in the TLB (or translation caches [180]) need to be flushed because of a different kernel address space. The TLB in recent Intel processors improves the performance of these mitigations with tagging. Every entry is tagged with the address space identifier and as a result, it is no longer necessary to flush the (tagged) TLB on every privilege switch. Due to the improved performance, tagged TLBs have been praised for making these mitigations practical for deployment [58].

TagBleed Unfortunately, a tagged TLB is not a panacea and the gain in performance comes at a significant security cost. Tagging implicitly re-enables the sharing of the TLB between different address spaces. This allows an attacker to probe addressing information in the TLB left by kernel execution. As we will show, the leakage surface of tagged TLB is limited compared to known attacks against KASLR [44, 59, 71, 81]. Nevertheless, we show that this leakage is enough to fully derandomize KASLR when used in combination with a secondary side-channel attack that uses the kernel as a confused deputy [64] to leak additional information about its address space. Mounting these attacks is not trivial since kernel memory is mapped with huge and super pages (i.e., 2 MB and 1 GB) and (tagged) TLBs use previously-unexplored hashing functions for storing the translation information for these pages. Our proof-of-concept exploit, TagBleed, uses the information we obtained through reverse engineering to break KASLR in under one second using the aforementioned attacks despite all existing state-of-the-art mitigations.

Contributions In summary, our contributions are:

- We highlight the security implications of tagging (previously-untagged) cache components for the first time. While tagging components improves performance, they can come at a security cost.
- We present an extended analysis of the architecture of the TLB in modern processors. Expanding on the existing knowledge of the TLB for normal pages [53], we reverse engineered the TLB architecture for both 2 MB huge pages and 1 GB super pages used when mapping kernel entries.
- The design and implementation of our practical attack, TagBleed, which derandomizes KASLR on a recent Linux system with current defense mechanisms deployed in under one second. TagBleed targets tagged TLB in combination with a confused deputy attack to fully break KASLR.

2.2 Background

Virtual Memory In modern operating systems, each process has access to a private virtual address space and operates solely on virtual addresses. The translation to the actual physical addresses is the responsibility of the MMU (Memory Management Unit) which walks the multi-level page table structures that contain the virtual-to-physical mappings for each address space, as well as permission flags (such as the supervisor bit that indicates a page can be accessed from user space). Since these address translations are expensive and happen at each memory access, the MMU utilizes the TLB (Translation lookaside buffer) to cache the last few lookups—speeding up subsequent accesses to the same page by orders of magnitude.

In the operating systems with unified kernel and user address spaces that were popular until very recently, the kernel did not have an address space of its own, but rather shared the page tables of the running user process and relied on the supervisor bit to protect its pages from illegitimate accesses from the user process. As neither the address space (page tables) nor the content of the TLB needed to change on kernel boundary crossings, such an optimized memory organization was very efficient. The optimization is especially effective on processors without tagged TLBs, since they must perform a full TLB flush on every address space switch [55].

	address range	entropy	possible slots	alignment
kernel image	0xffffffff81000000 - 0xffffffffbe000000	9	488	2MB
kernel module	0xffffffffc0001000 - 0xffffffffc0400000	10	1024	4KB
page_offset_base, vmalloc_base, vmemmap_base	0xffff888000000000 - 0xfffffe0000000000	15*	25600*	1GB

* depends on amount of physical memory and `CONFIG_RANDOMIZE_MEMORY_PHYSICAL_PADDING` configuration.

Table 2.1: KASLR entropy in Linux 4.19.4 for the kernel image, kernel modules and page_offset, vmmalloc and vmemmap. The number of possible slots for the kernel image are dependent on the size of the kernel image. The entropy and end address for page_offset depends on whether five page table levels are supported and how much physical memory is available.

As a result of a barrage of side-channel attacks [59, 71, 81, 109] that all abused the unified address space, modern operating systems recently abandoned it altogether and now provide the kernel with its own, completely separate address space. On Linux, this is known as KPTI (Kernel Page Table Isolation) [33], while Windows refers to it as KVA (Kernel Virtual Address) Shadow [84]. Fortunately, while the separation of address spaces is still quite expensive on older processors because of the TLB flushes, newer CPUs offer tagged TLBs where each entry contains an identifier (or “tag”) of the address that owns it. For instance, Intel’s x86_64 processors have supported tags in the form of 12-bit PCIDs (Process Context Identifiers) since the SandyBridge architecture [75]. When performing a lookup in the TLB, the entry’s tag must match that of the currently active address space. Thus, flushes are no longer needed, as there is never any confusion about the validity of a TLB entry—greatly improving performance.

Address Space Layout Randomization To prevent attackers from locating suitable targets to divert a program’s control flow in the presence of a vulnerability, all major operating systems today support Address Space Layout Randomization (ASLR). ASLR randomizes the locations of the code, heap and stack in memory and forms an effective first line of defense against memory error attacks. The kernel variant of ASLR, known as KASLR, is deployed on all major operating systems. This requires attackers to first break KASLR as a crucial step in kernel exploitation.

KASLR randomizes the location of the kernel and drivers running in kernel space either at boot time or at driver load time. Since brute forcing the randomization in the kernel is typically not an option due to the high rate of kernel panics, the randomization entropy in the kernel can be lower than in user processes. For instance, at the time of writing, the entropy for the kernel image in Linux is 9 bits, while the entropy for user-space code is as high as 30 bits [120].

More specifically, KASLR in Linux randomizes different kernel regions differently. At boot time the kernel image is unpacked and relocated to a random location. Regions for the identity map, vmalloc and vmemmap are randomized separately. Finally, kernel modules are randomized the first time a module is loaded. As shown in Table 2.1, the kernel image has 9 bits of entropy, and kernel modules have 10 bits. Identity map, vmalloc and vmemmap are randomized with a shared entropy depending on the size of physical memory.

TLB Tagging Switching address spaces on processors prior to TLB tagging is a costly operation due to the invalidation of all entries. Therefore, tagging TLB entries with an identifier for its current address spaces was introduced as an optimization. Intel processors use the first 12 bits of the CR3 register to store a so called PCID (process-context identifier) [75]. Entries in the TLB will only be taken into account if the current PCID in CR3 matches the PCID of the entry. This makes context switching more performant, since the TLB does not require any flushing.

Cache Attacks On address lookups, the TLB caches virtual to physical memory translations and as a shared resource between processes running on the same core, clearly introduces a side-channel risk that was exploited in the TLBleed attack by Gras et al. [53]. However, the TLB is just one of many shared resources that have been used for side-channel attacks. Modern Intel CPUs have multiple levels of caches to speed up memory accesses. Specifically, each core has its own L1 and L2 caches and shares the last level cache (LLC) with the other cores. Since attacker and victim don't even need to run on the same core, the LLC is a particularly interesting target for a side-channel attack [61, 111, 129, 173, 195]. Attackers can simply populate cache sets and then measure whether the victim process evicts their data. With this information the attacker can infer that the victim used addresses that map to the same cache set and researchers have shown that this is enough to leak sensitive information such as cryptographic keys [1, 12, 15, 137, 163].

AnC The AnC [54] attack measures the timing of accesses performed by the MMU during virtual address translation to break ASLR from within the browser. Whenever an address translation is not cached in the TLB, the MMU does a page table walk. It reads offsets within the multi-level page tables in order to first dereference the next page table and then the address of the physical page. In order to speed up expensive page table walks, accessed parts of the page tables are cached within the CPU's data caches. This makes consecutive translations faster even if the MMU has to do a page table walk. Inevitably, caching parts of the page table leaves traces in the CPU cache depending on the translated virtual address. By partially flushing the data caches, AnC can measure which cache lines within the page table pages have been used during a translation. This information already significantly compromises the ASLR entropy. To fully break ASLR, AnC needs to find out which page table entries within the target cache lines are accesses by the MMU. This is achieved by accessing a large virtual contiguous buffer, and observing the transitions between activated cache lines, known as *sliding*. The demonstrated attack requires large accessible virtually consecutive buffers (e.g., 8 GB). While this is possible in the browsers, it becomes challenging when applied to the kernel.

2.3 Threat Model

We assume an attacker that can execute unprivileged code on the target system with a kernel that is hardened with all common mitigations, including KASLR and DEP [121]. For the hardware, we assume a modern processor with a tagged TLB. In this chapter we focus on Intel processors with PCID-based TLB tagging, ARM and AMD however also provides a similar feature through ASIDs. The attacker aims to elevate privileges to ring 0 by exploiting a memory corruption vulnerability in the kernel or a kernel module. To do so, the attacker first needs to break KASLR. In general, breaking KASLR is possible

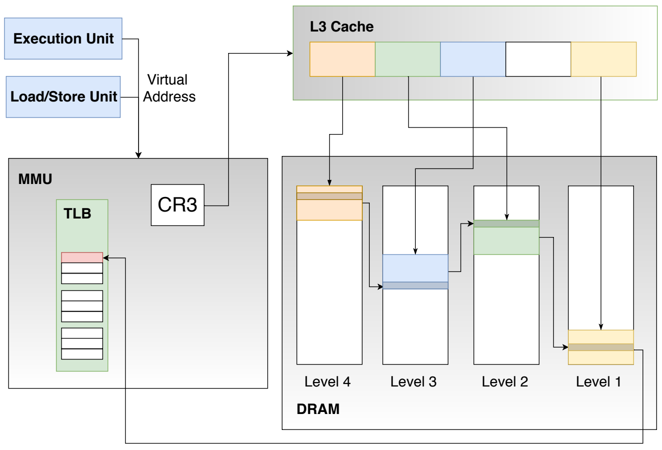


Figure 2.1: High level overview: On a memory access the MMU will translate a virtual address to a physical address using the page table structures. The result of the translation will be cached in the TLB and parts of the page tables in the L3 cache (LLC). By observing the state of the TLB and LLC we locate the position of kernel translations in the cache. Because the position is dependent on the translated virtual address we can successfully derandomize KASLR.

via a software-based information disclosure vulnerability or a side-channel attack. We assume that the kernel does not contain a known information disclosure vulnerability and that powerful mitigations against side-channel attacks on KASLR are turned on [47, 58]. The attacker's aim is to bypass these mitigations and successfully break KASLR with a side-channel attack. In this chapter, we mostly focus on the Linux kernel, but the techniques we develop will likely be applicable in other kernels as well.

2.4 Attack Overview

Existing side-channel attacks on KASLR rely on the kernel being mapped in the same address space as the user process [59, 71, 81]. Specifically, these attacks can detect whether the kernel is mapped at a given address without the permission to *access* that address. By removing the kernel from the user address space and moving it to its own memory space these side channels are no longer possible [47, 58]. As kernel address space isolation ensures that kernel memory is only mapped while executing in kernel space, an attacker needs to perform a confused deputy attack [64] on the kernel, tricking it into performing the attack on itself.

Unfortunately for the attacker, the confused deputy attack is not possible with any of the existing techniques. For instance, the TSX attack [81] would require the kernel to access a user-controlled pointer in a hardware transaction. However, since the Linux kernel does not use transactions, this is not a feasible attack vector. The same is true for the `prefetch` instruction [59], while triggering invalid kernel page faults when performing the attacks described by Hund et al. [71] would crash the kernel. Therefore, our confused deputy attack should find other mechanisms. The simplest operation that we can force a kernel to perform an access to its own memory. Memory accesses occur on every single instruction as the processor loads text and often data from memory. An important research question is whether it is possible to break KASLR using uncontrolled valid memory accesses performed by the kernel—and as we will see, the answer is yes.

Figure 2.1 shows how the information that the kernel maintains for address translation leaves traces in various caches when performing a memory access. On each memory access, the MMU performs a virtual to physical address translation, first consulting the TLB to see whether the result is already cached there. If the translation is not in the TLB, the MMU performs a page table walk to translate the virtual address into a physical one. Accessing page table entries in the page table results in caching parts of the page table in the CPU's data caches. Bits of the virtual address determine the set in the TLB and which part of the page table is being cached.

Ironically, the introduction of tagged TLBs, so important for the performance of systems with an isolated kernel address space, undermines the very isolation it should be helping, since it is now possible for a user process to probe TLB sets in its own address space to detect kernel activity in these same sets. Moreover, the sets in which there is activity reveal information about the virtual address of kernel memory. However, breaking KASLR by observing such TLB activations presents several challenges:

C1 Since the kernel is mapped using huge or super pages, we need to understand the TLB architecture for these page types and their addressing function. The TLB architecture for 4 KB pages has only recently been reverse engineered by Gras et al. [53], but the TLB behavior for larger pages remains entirely unknown.

C2 Given the small number of TLB sets, the partial information retrieved from a tagged TLB is unlikely to be sufficient to fully derandomize KASLR. We therefore need an auxiliary side channel to combine it with the side channel over tagged TLB. This raises the challenge of combining these side channels.

C3 Timing a kernel memory access in a confused deputy-style attack requires forcing the kernel to perform a certain operation on behalf of a user process (e.g., a system call). Compared to existing attacks that simply time a single memory access by itself, the system call introduces significant additional noise. Building a practical side-channel attack in such a setting is challenging.

We first address **C1** by reverse engineering the TLB architecture for huge and super pages in Section 2.5. In Section 2.6, we then show how an EVICT+TIME attack on a tagged TLB combined with a constrained variant of the AnC attack [54] allows breaking KASLR (addressing **C2**). We further show how selective TLB flushing, made possible through our reverse engineering efforts, can significantly reduce noise to make our attack practical (addressing **C3**).

2.5 Reverse Engineering the TLB

For the TLBleed attack [53], the authors present their efforts in reverse engineering the TLB architecture to understand how virtual addresses map to different sets in the TLB. They use Intel Performance Counters (PMCs) to gather fine-grained information about TLB misses and their level. Intel provides performance events like `dtlb_load_misses.stlb_hit` and `dtlb_load_misses.miss_causes_a_walk` which allow differentiating between a L2 sTLB hit and a miss.

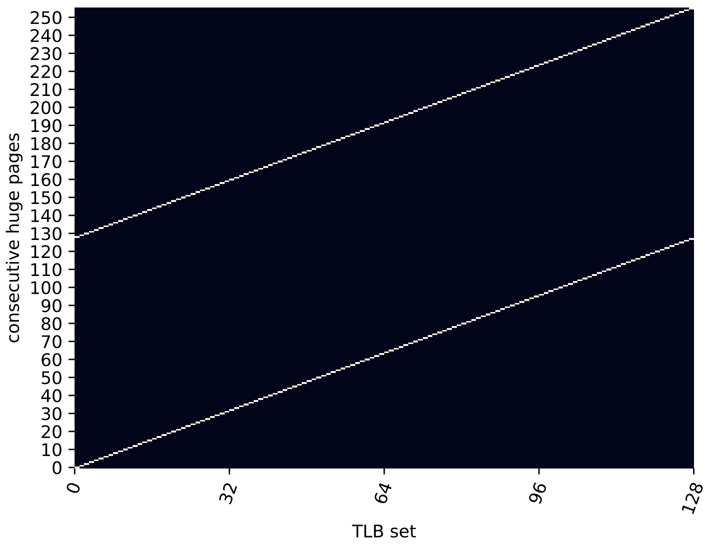


Figure 2.2: The graph shows the TLB set for 256 virtual consecutive huge pages starting at address `0x6ffe00000000`. The hash function for huge pages is linear, given by the 7 bits after the page offset.

With their findings they define the L1 TLB as linearly-mapped TLBs. This describes the hash function determining the TLB set of a virtual address. In a linearly-mapped TLB the hash function is given by $tlb\_set(va) = page_{va}\ mod\ s$. A TLBs architecture is defined by its sets *s*, ways *w* and its hash function *tlb_set(va)*. The L2 sTLB however is a complex-mapped TLB in recent Intel architectures. This means the hash function is not linear. In the Skylake sTLB, for example, the hash function is given by a *XOR-7* function which xors bits 19 to 13 with the bits 26 to 20 of the virtual address.

We first tested whether we can evict a TLB entry for a huge page (i.e., 2 MB) by flushing the whole TLB with 4 KB pages. Using performance counters we verified that we were able to evict the TLB entry of the 2 MB page from the L1 dTLB and the L2 sTLB. We conclude from this that both L1 dTLB and L2 sTLB are shared between 4 KB and 2 MB pages. This information already gives us the set and ways for the L2 sTLB since it is the same for normal pages. Contradictory to the information from `cpuid`, the L1 dTLB (2 MB pages) on Skylake therefore has 64 entries with 16 sets and 4 ways instead of the stated 32 entries and is shared with the L1 dTLB for 4KB page translations. Based on our findings the dTLB for 2MB pages, as well as the sTLB for 1GB pages are linearly mapped with the bits after the page offset. We, however, still need to reverse engineer the hash function that determines in which of the 128 sTLB sets a huge page is put in. We know that the sTLB is shared with 4KB pages where, on Skylake, the set is determined by a complex *XOR-7* hash function. To observe the addressing function for 2MB pages we evict one set in the sTLB at a time using 4KB pages. That allows us to observe which TLB set the huge page is mapped to. Figure 2.2 presents our results on Skylake for measuring the TLB set for 2MB pages. Based on our measurements, the sTLB set for 2 MB huge pages is determined by bits 27 to 21 of the virtual address (VA[27:21]). Contradictory to the complex *XOR-7* hash function for 4 KB pages, for 2 MB pages it is only addressed

Page size	L1 dTLB				L2 sTLB			
	sets	ways	indexing	shared	sets	ways	indexing	shared
4 KB [53]	16	4	VA[15:12]	✓	128	12	`XOR7`(VA[25:12])	✓
2 MB	16	4	VA[24:21]	✓	128	12	VA[27:21]	✓
1 GB	-	4	-*	✗	4	4	VA[31:30]	✗

* fully associative

Table 2.2: TLB architecture for Skylake architecture based on our reverse engineering efforts. The indexing column determines the bits used for the linear indexing function. Only the sTLB for 4 KB pages is indexed with the complex `XOR7` indexing function.

with a linear addressing function. An overview of our findings on the TLB architecture is summarized in Table Table 2.2.

2.6 TagBleed

In this section, we discuss the building blocks of our TagBleed attack. We first discuss how we can force the kernel to access its virtual memory to start off our confused deputy attack. We then describe how we utilized the knowledge we gained through our reverse engineering for leaking kernel virtual address information through the tagged TLB and to reduce noise during the attack. After that, we discuss how we utilized a constrained version of the AnC attack to break the remaining residual (2 bits) of entropy for the kernel text. We further extend our attack to derandomize the location of kernel modules and data.

2.6.1 Forcing Kernel Memory Access

Measuring the latency of a single memory translation in the kernel is challenging from user space. It is simply not possible to start the timer before that specific translation and stop it straight afterwards. We need to start the timer prior to entering the kernel and stop it once returned to user space. Measuring the entire length of the kernel operation introduces additional noise caused by other instructions being executed. System calls are an easy way for a user process to communicate with the kernel. To minimize noise of other code executed during the system call, we took the shortest possible system call by providing an invalid system call number. Early in the system call handler the kernel will look for a system call with the provided number and abort because of the invalid argument. With calling such an invalid system call, we minimize the time spent in the kernel, minimizing the amount of unrelated instructions being executed within the timing window. We note that the `SYSCALL` instruction is also considerably faster than `INT 0x80`, in order to shorten the execution path.

2.6.2 Leaking Through Tagged TLB

As mentioned in Section 2.2, KASLR support in Linux aligns the kernel text to 2 MB and randomizes bit 21 to 29 of the virtual address, in other words, the slot in second page table level (i.e., PTL2) as shown in Figure 2.3. We craft an EVICT+TIME attack, evicting one TLB set at a time and timing a target (kernel) memory access. If the TLB

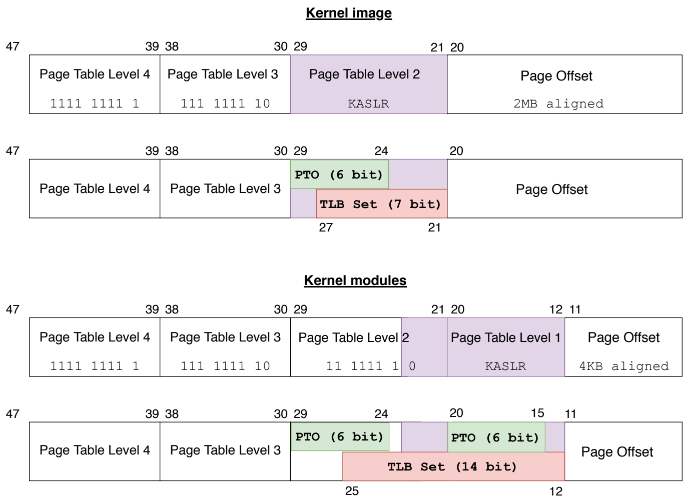


Figure 2.3: For the kernel image the whole second page table level (PTL) is randomized by KASLR. Using TLB sets we are able to randomize the lower 7 bits. Combining it with page table offset (PTO) information, which derandomizes the higher 6 bits, we can successfully break KASLR. For kernel modules PTL1 and the lowest bit of PTL2 are randomized. Using the TLB set, given by the `XOR-7` hash function, together with the offset within the page table is enough to derandomize those 10 bits of entropy.

set with the desired entry is evicted, the MMU is forced to perform a page table walk. A page table walk is considerably slower than just using the cached translation from the TLB. Note that due to KPTI the TLB would be flushed entirely on a context switch if no TLB tagging would be in-place. With the tagged TLB, we now have the capability to selectively evict parts of the TLB and observe the effects across context switches. As already presented, the L2 sTLB set for huge pages is determined by VA[27:21], which allows derandomizing the lower 7 bits of the second page table level. The last 2 bits of PTL2, which remain unknown, are derandomized by combining this attack with information obtained through page table walker discussed in the next section.

Reducing noise Selectively flushing one TLB set at a time also reduces the noise. Not evicting the entire TLB massively reduces the amount of unwanted page table walks which create a large amount of false positives.

2.6.3 Confused Deputy Attack with AnC

The AnC attack on the MMU can break ASLR when the attacker can freely *slide* in the virtual address space [54]. AnC relies on the MMU caching parts of the page table page on a page table walk. Depending on the virtual address, different parts of the page table end up being cached in the LLC. Using EVICT+TIME, AnC locates the cache lines containing the accessed page table entries by the MMU. However, this will not reveal the complete virtual address since multiple 8-byte page table entries are stored in

```
for each tlb_set do
    for each cache_line do
        $evict\_l1\_tlb()$
        $evict\_l2\_tlb\_set(tlb\_set)$
        $evict\_cache\_line(page\_table\_cache\_line)$
        $past \leftarrow rdtscp()$
        $syscall$
        $now \leftarrow rdtscp()$
        $timing[tlb\_set][cache\_line] \leftarrow now - past$
    end for
end for
```

Listing 1: Timing a system call by only evicting one TLB set and one page table cache line at a time.

the same 64-byte cache line. AnC addressed this problem by accessing large virtually contiguous memory addresses, i.e., sliding. Accessing subsequent virtual addresses cause subsequent cache lines to be accessed by the MMU. The point at which a new cache line is accessed reveals the offset of page table entries in a cache line – fully derandomizing ASLR. While powerful, it is not trivial to apply the AnC attack to the kernel due to two reasons. First, we cannot make the kernel slide its address space, and second, each step of the AnC attack causes up to four cache line activations due to four levels of the page table, introducing false positives. We present a variant of this attack integrated into our tagged TLB side channel to leak the remaining 2 bits of entropy and making TagBleed more noise-resistant.

Sliding As shown in Figure 2.3, the residual 2-bits of entropy left from our TLB attack are not related to the offset of page table entries within the cache lines. This means that we do not need to perform sliding to retrieve these two bits. As a result, *a single memory access* by the kernel is enough to break KASLR when combining these two side channels.

Other PTLs In order to speed up page table walks Intel not only caches complete virtual to physical translations in the TLB, but also partial translations in its *translation caches* [180]. These caches allow the MMU to skip page levels during the translation. We make use these translation caches to force the MMU to skip page tables that are not interesting for KASLR (see Figure 2.3). This allows us to avoid noise caused by other page table levels.

Combining the side channels Listing 1 shows the high level operation of TagBleed when combining the tagged TLB and AnC attacks. We generate a two-dimensional matrix with all combinations of evicting one TLB set and one cache line. Then we use a simple script to identify the best candidate. We first identify the best candidate for the TLB set. Since this derandomizes the lower 7 bits of PTL2, only the 2 highest bits are missing. Therefore, we only need to choose between 4 possible cache lines. Since the AnC attack gives us the upper 6 bits of PTL2, we can use the upper 4 bits of the best candidate from the previous step as a noise filter when selecting the final candidate.

Vendor	Microarchitecture	CPU model	Year	TLB tagging
Intel	Haswell	i7-4650U	2013	PCID
Intel	Skylake	i7-6700K	2015	PCID

Table 2.3: Microarchitectures used in evaluation.

2.6.4 Derandomizing Kernel Modules

Kernel modules are loaded with an offset randomized by KASLR when the first module is loaded in the system. In order to observe the signal, we need to force an access to a memory location within the kernel module. We achieve this by performing an *ioctl* call to a loaded kernel module. Kernel modules are mapped with 4 KB pages, contrary to 2 MB pages used for the kernel image. Therefore, as shown in Listing 1, bits 12 to 21 of the virtual address are randomized, since kernel modules are not 2 MB, but 4 KB aligned. This slightly changes our approach since the TLB indexing function for 4 KB pages is different than for 2 MB pages. For example, on Skylake the TLB set for 4 KB pages is determined by an XOR of bits 12 to 18 with the bits 19 to 25 of the virtual address. Hence, using TLB sets alone we cannot break any KASLR bits. The AnC attack, however, provides us with bit 15-20 through the offset of the activated PTL1 cacheline. Bits 19 and 20 allow us to find bits 12 and 13 as well, since bits 19 and 20 are XORed with bits 12 and 13 in the TLB's XOR-7 pattern [53]. The remaining entropy will be a single bit, since we cannot break KASLR at bit 14 and 21 while we know their XOR value.

2.6.5 Derandomizing Physmap

Derandomizing physmap is challenging because it is 1 GB aligned and randomized in PTL3 and PTL4. The TLB indexing function for 2 MB pages does not use those upper bits. Most of the physmap, however, is mapped with 1 GB pages with a separate TLB with its own indexing function. But since the sTLB for 1GB pages only has 4 sets, we can only use it to derandomize the lower 2 bits of PTL3. To derandomize the rest, we can make use of AnC to derandomize the higher 6 bits of both PTL3 and PTL4 reducing the entropy by another 10 bits. This still leaves us 4 bits of entropy (16 possible locations).

2.7 Evaluation

We evaluated TagBleed on a machine running Ubuntu 18.04 LTS (Linux kernel v4.19.4) with an Intel Core i7-4650U @1.70 GHz (Haswell) and 8 GB of RAM. In order to ensure portability across different architectures, we confirmed our evaluation results on another workstation running Ubuntu 18.04.1 LTS (Linux kernel v4.15.0) with an Intel Core i7-6700K @4.00 GHz (Skylake) and 16 GB of RAM. This also allowed us to confirm that a range of different TLB architectures is susceptible to our TagBleed attack. Table 2.3 details the CPUs and microarchitectures considered in our evaluation. In our evaluation, we targeted the derandomization of KASLR for the kernel image.

2.7.1 Side channel by TLB set eviction

We first evaluated our assumption on whether the partial TLB set eviction from user space can influence the MMU's virtual-to-physical memory address translation. Without

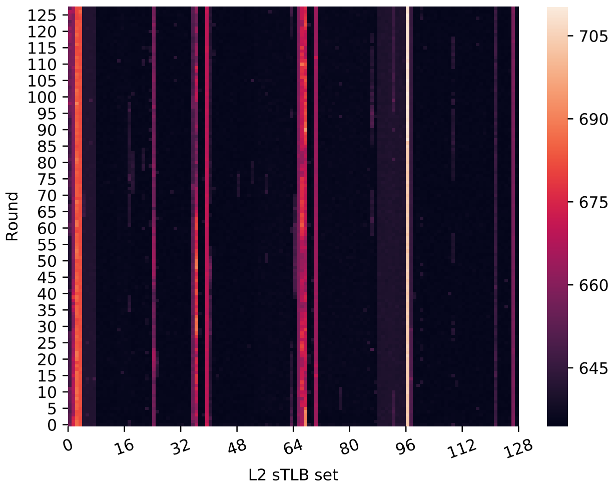


Figure 2.4: We can clearly see that evicting some TLB sets will slow down the time of an invalid system call. This can only happen when the (tagged) TLB is not fully flushed as the kernel switches address spaces with KPTI enabled.

KPTI, the (unified user/kernel) address space is not switched on kernel entry, so no TLB flushing is performed. With KPTI, however, the kernel updates the CR3 register to switch to the separate kernel address space. This operation does flush the TLB on legacy architectures, hindering our partial TLB set eviction strategy. However, on modern tagged TLB architectures, tagged entries are no longer flushed at mode switching time, and we should be able to surgically trigger a page table walk only when evicting the correct TLB set.

Figure 2.4 validates our assumption, depicting the impact of evicting different TLB sets on the execution time of a dummy (i.e., invalid) syscall. Note that using an invalid syscall, that is a syscall with an invalid syscall number, is a convenient way to trigger short-lived kernel activity, but using any other short-lived syscall (e.g., `reboot` without root privileges) would also serve our purposes.

As Figure 2.4 shows, the execution time increases only when evicting specific TLB sets (revealing the virtual memory activity of the syscall). The signal persists if we disable KPTI, since the address space is not changed on a context switch, keeping the current state of the TLB. KPTI on a legacy TLB architecture without a tagged TLB requires a full TLB flush on a context switch. This clears the state of the TLB cache and therefore stops TagBleed, but it comes at a high performance cost for each user to kernel transition.

2.7.2 Side channel by cache line eviction

Our second assumption is that we can observe the cache lines being accessed during a kernel page table walk. In order to test this assumption, we built a kernel module to perform an AnC-style cache attack to monitor the page table walks performed by the

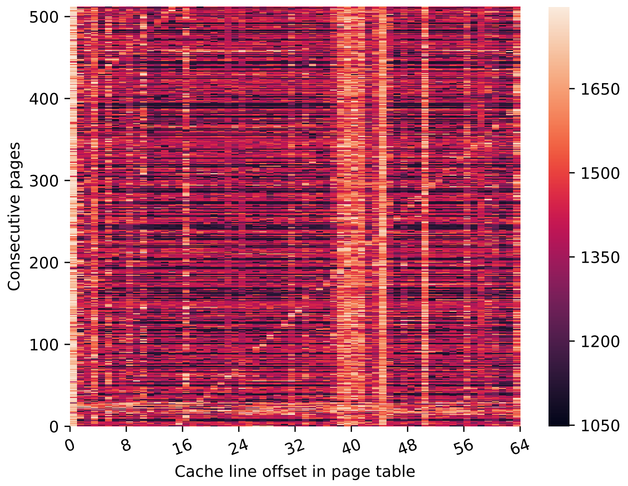


Figure 2.5: Although more noisy than a same-process page table walk signal, the sliding is still visible when timing the execution of an `ioctl` syscall to our kernel module. The offset within the buffer, which the kernel module accesses, is passed as an `ioctl` argument.

MMU. The kernel module gives us the ability to access a given offset within a kernel buffer to perform sliding on the virtual address space and make the signal more visible. By accessing virtually contiguous pages, which we define as sliding, the cache line of the page table entry will also be incremented. We then time the execution of an `ioctl` syscall that causes the kernel module to access a byte at a given offset. Figure 2.5 validates our assumption, depicting the signal for 512 contiguous virtual memory pages measured by timing the execution of the `ioctl` syscall.

However, when we repeat the experiment with an invalid syscall and without our kernel module (and therefore without sliding), the signal becomes very noisy, as depicted in Figure 2.6. This shows that, due to the entire cache activity of the kernel being impacted by cache evictions, without detecting the jump in between cache lines and absent other side channels, it is challenging to detect if an eviction induced a cache miss during a page table walk or in other kernel operations.

2.7.3 Combining the two side channels

Next, we combine the two side channels used by TagBleed to 1) derandomize all the required bits of entropy and 2) combine the available information for better TLB set and cache line detection. We first showcase our TagBleed attack sensing the second page table level for a huge page access in user space. This scenario demonstrates our attack in a low-noise scenario due to the ability to carefully time a single user-level memory access. Next, we show that TagBleed's signal is still detectable when measuring it through kernel activity triggered by a system call.

Figure 2.7 depicts the signal for our combined EVICT+TIME attack on user-level huge page access. As shown in the figure, the attack yields a fast access for all TLB sets

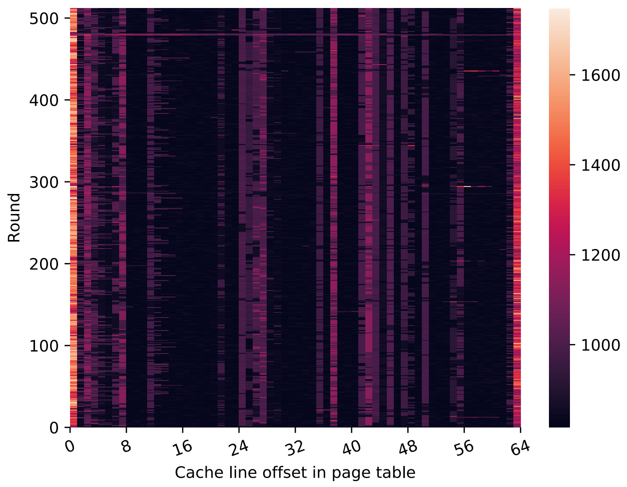


Figure 2.6: Cache evictions can slow down syscall execution in many unpredictable ways. As shown in the figure, not one but many different cache lines introduce cache misses in kernel execution even for an invalid syscall. The graph was created with the kernel target page being mapped using the sixth cache line, whose signal does not even stand out compared to other cache lines.

except the one triggering a page table walk. If, for that TLB set, we also evict the cache line of the second-level page table entry, the page table walk (and ultimately the access) is even slower. To only analyze the signal for the second-level page table, we keep the higher levels cached in the page table caches.

Figure 2.8 presents the same experiment but operated on an invalid syscall—a more useful but also more noisy scenario. As expected, the kernel activity yields several other memory access and results in several other cache misses. Nonetheless, we can identify the correct TLB set for kernel pages by finding the TLB sets that consistently keep getting evicted for all the cache lines. Another strategy to find the correct TLB set is finding several TLB sets close to each other. This requires accesses within different parts of the kernel image in consecutive TLB sets. However, the most reliable way to determine the correct TLB set is verifying if one of the four possible cache lines is considerably slower than the other cache lines.

2.7.4 Success rate

In order to evaluate the success rate of TagBleed, we ran 50 trials, restarting the system each time to trigger a rerandomization with KASLR. For each trial, we performed a total of 20 runs to minimize the risk of temporary noise. In 47 of 50 trials, we are able to recover the correct location of the kernel. In three other trials, we could not reliably disambiguate two cache lines. This translates to a 94% success rate while reducing the KASLR entropy down to 1 bit in the other cases.

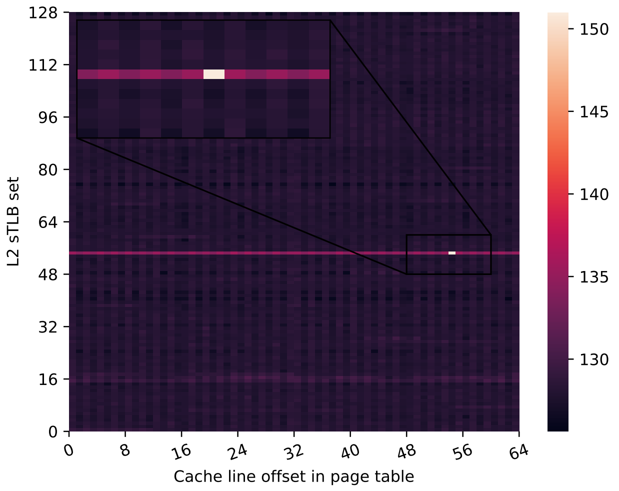


Figure 2.7: A combined EVICT+TIME attack on a user-level huge page access. Only when evicting the TLB set 55, the MMU performs an expensive page table walk. Moreover, when evicting cache line 54, the page table entries need to be loaded from memory which slows down the page table walk.

2.7.5 Attack time

Our TagBleed attack can be run in less than a second with satisfying results. Nonetheless, timing is usually not critical when building kernel exploits in a local exploitation scenario, where the attacker has already been granted (or achieved) unprivileged code execution. By default, we therefore increase the number of rounds to 10 for each TLB set to reduce the amount of false positives. This still allows us to run the attack in less 3 seconds including the time to run our solver script.

2.7.6 Noise

For the purpose of noise reduction, TagBleed combines two different side channels. Figure 2.6 shows that, without combining multiple side channels, TagBleed cannot easily battle spatial noise. For increased reliability against temporal noise, we repeat our measurements in several rounds. Temporal noise is especially critical when timing system calls. The kernel can potentially reschedule other processes after the system call instead of rescheduling the attacker-controlled process. By repeating the evictions in several rounds, we make sure that temporal noise does not negatively affect the measurements. As we lower the number of rounds below the default value of 10, we quickly observed degradation of the success rate due to temporal noise. As we increase the number of rounds above 10, we observed minor improvements to the previously reported success rate. Overall, we believe 10 is a good choice even for relatively noisy environments.

To assess noise in a realistic use case, we ran the experiment with additional workload on an Ubuntu Desktop with an open browser, running a YouTube video, and several

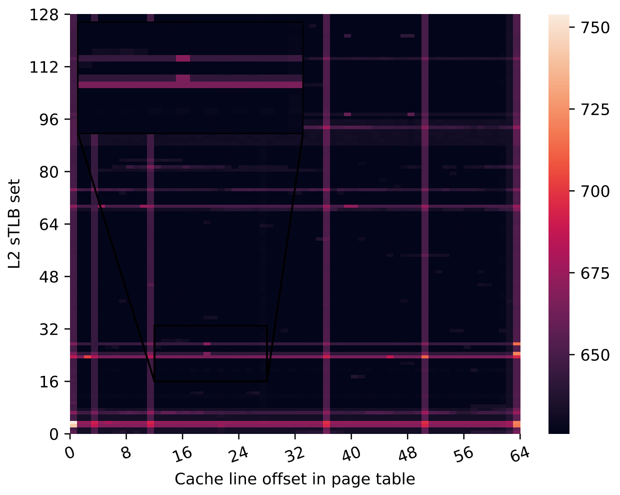


Figure 2.8: A combined EVICT+TIME attack on an invalid syscall. Based on our solver, we can see a signal for the TLB sets 25 and 27. These TLB sets have been selected based on their high slowdown for the last cacheline relative to all other cache lines. When testing for the four possible cache lines 3, 19, 35, 51 we can identify cache line 19 as the one containing the page table entries. Cache line 19 has been selected because it's not slow across all TLB sets but distinctively in TLB set 25 and 27.

applications running such as an email client. We could not measure any effect that suggests TagBleed is affected by such noise. The attack is also not required to run longer compared to the experiment without additional workload.

2.7.7 Comparison against other KASLR attacks

In Table 2.4, we compare our attack with existing attacks when considering existing state-of-the-art mitigations. TagBleed compromises KASLR regardless of either mitigation since it relies only on the shared tagged TLB and CPU data caches rather than shared branch state or shared address space. We verified our attack TagBleed against the latest Linux kernel with KPTI, which is based on the KAISER patches. All other existing attacks are mitigated by LAZARUS. Since TagBleed does not rely on a unified address space or the BTB, it is not mitigated by LAZARUS. Furthermore, KAISER protects against all existing attacks except BTB-based attacks (although the latter can be also mitigated using complementary mitigations such as eIBRS [73]).

2.8 Mitigations

We distinguish between defenses specific for the TLB side channels, defenses targeting cache side channels, and generic mitigations that target the timing primitives.

	KAISER [58]	LAZARUS [47]	time
Double PF [71]	✗	✗	< 1 min
prefetch [59]	✗	✗	12 s
TSX [81]	✗	✗	0.2 s
BTB [44]	✓	✗	60 ms
TagBleed	✓	✓	< 1 s

Table 2.4: KASLR attacks vs. defenses.

Stopping TLB side-channel attacks Completely removing the TLB side channels requires all shared state to be removed. There are two ways to do so: spatial and temporal partitioning.

Spatial partitioning removes the side channel by isolating user processes from the TLB sets associated with the kernel. With current architectures this is not practically possible, since a partitioned TLB makes it extremely hard to guarantee contiguous virtual memory. Changing the TLB indexing function in future architectures could work, although hardware changes are expensive, and it is not unlikely that doing so will introduce performance degradation. After all, the current function is specifically chosen for performance.

Instead of spatial partitioning, it is also possible to kill the side channel by partitioning the TLB in time—by performing a full TLB flush upon crossing security boundaries. Disabling TLB tagging/PCIDs to flush the TLB completely effectively mitigates our attack but at the cost of high performance overhead for all implementations of kernel address space isolation.

Stopping LLC side-channel attacks As temporal partitioning of the last level cache, although possible in theory, is not a feasible solution for obvious performance reasons, we consider only spatial isolation.

Cache coloring divides pages between kernel and user process in such a way that they do not share cache sets. Doing so will stop leaking through the LLC, but is far from trivial [172], has serious performance implications for both user processes and the kernel [91, 157], and needs to account for all memory used on behalf of the user process. Moreover, LLCs are not the only side-channel option for the determined attacker.

Instead of caching the content of the page table in the LLC, future processor generations could put it in a dedicated, isolated page table cache. Clearly, such a solution requires expensive hardware changes. Also, that page table cache would have to be designed carefully, lest it opens up a new potential side channel (e.g., if it is shared between user processes and kernel).

Stopping general kernel timing attacks At heart, our attack is possible because attackers can measure subtle timing differences in system call execution. Removing this ability would also stop the attack. To do so, we identify three approaches: detection of timing attacks, constant time system calls, and timer crippling.

HexPads [135] has shown that, in principle, performance counters can be used to detect ongoing side-channel attacks. However, it is hard to guarantee full mitigation and detection also introduces the risk of false positives and false negatives. While at first sight it may also seem possible to introduce a defense in the kernel to detect a high rate of failed system calls, say, such a solution would be naive. First, it is hard to be sure if the failed calls are really due to a side-channel attack. Second, attackers can easily make their attack more silent by extending its time and running it from separate processes. In order to mitigate the ability to time invalid system call the kernel could enforce a constant execution time for invalid system calls. Doing so would not influence performance during normal use, since invalid system call number are not typically used by normal applications. However, this is also not very effective because we could just find a short valid system call as an alternative to an invalid system call. The alternative, making all system calls constant time is not a practical solution. The most obvious mitigation is to cripple the timers, for instance by removing the availability of high resolution timers such as `rdtsc`. Again, this solution is not realistic since high resolution timers are vital to many applications. Moreover, crafting *ad-hoc* high-accuracy timers, e.g., using concurrent threads is always possible.

In summary, mitigating side channels for every single memory access is challenging and/or expensive. The simplest and most practical mitigation may be to simply use the higher bits of the virtual address for the operating system's implementation of KASLR. Since our attack fully derandomizes PTL2, extending the randomized bits into PTL3 could work as a possible mitigation, even though moving the location of the Linux kernel may (and probably will) introduce unforeseen performance issues. We point out that with our technique we would still be able to derandomize PTL2 which removes those bits from the actual entropy of both KASLR and user-space ASLR.

2.9 Related Work

Derandomizing (K)ASLR Derandomizing ASLR has been an active research topic as a fundamental primitive for code reuse attacks [145]. The simplest way to break ASLR is to leak code or data pointers with memory disclosure vulnerabilities [36, 159]. However, side-channel attacks showed that even without disclosures it is possible to derandomize the address space layout. These side-channel attacks use techniques such as control flow timing [43, 153], memory deduplication [9, 16], or CPU caches [44, 54, 59, 71, 81].

Hund et al. [71] showed three different scenarios for breaking KASLR by perfoming timing attacks on CPU caches and the TLB. Yang et al. [81] used Intel TSX to suppress exceptions that normally happen on faulty memory accesses. Since no page fault is raised but the transaction aborts and returns directly back to the user, the difference between invalid permissions or a missing mapping is measurable. Grus et al. [59] showed that the execution time of the prefetch instructions can be used to detect the existence of virtual mappings in the kernel region. Evtyushkin et al. [44] demonstrated that the BTB (Branch target buffer) leaks bits to break current KASLR implementations in Linux. Finally, in 2018 Meltdown [109] used a speculative execution vulnerability in Intel CPUs to read the entire virtual and physical address space. This of course also breaks both ASLR and KASLR.

Most of the presented KASLR attacks suggested better isolation between kernel and user space as a defense. The attacks are possible because the kernel is mapped into each user process address space. Gruss et al. [58] presented KAISER as a kernel page table isolation (now implemented as KPTI in Linux) with low performance impact. With KPTI the whole kernel is no longer mapped into each user address space which defends successfully against the presented attacks except Evtyushkin et al.'s BTB attack [45], which instead is mitigated by explicitly (as done by LAZARUS [47]) or implicitly (as done by eIBRS [73]) flushing the BTB on privilege switches. eIBRS [73] can additionally prevent cross-thread BTB attacks if SMT is enabled.

Hardware timing side channels Physical shared resource may easily give rise to side channels. For instance, as early as 1996 when Kocher [96] presented his timing side channel on crypto primitives, Kelsey et al. [89] mentioned the idea of using the cache hit ratio as a side channel on large S-box ciphers to break cryptographic keys. This theoretical idea was formalized by Page [131] in 2002. Just one year later the first successful cache-based attack against DES was presented by Tsunoo et al. [175].

EVICT+TIME [129, 173] attacks can only observe a single cache set per measurement and have been used to recover AES keys from the victim's process. Concurrently, two other papers presented cache attacks to leak cryptographic keys. Bernstein used a similar method to EVICT+TIME to break AES, which required reference measurements for a known key in an identical configuration to the victims [12]. The second paper, by Percival [137] presented a cache-based attack on RSA with SMT.

Meanwhile, the PRIME+PROBE [111, 129] and FLUSH+RELOAD [61, 195] attacks are able to observe the state of the whole cache which makes them popular due to its faster bandwidth compared to EVICT+TIME. PRIME+PROBE was utilized by several researchers to leak private keys [78, 199], keystrokes [144] and to read information from other processes or VMs on the same machine from JavaScript [128]. The FLUSH+RELOAD attack requires page sharing with the victim, for example by some sort of memory deduplication. However, it is more fine-grained than PRIME+PROBE by measuring the exact cache line. Memory deduplication was deployed in most major operating systems which resulted in several FLUSH+RELOAD based attacks exploiting shared memory [4, 79, 80, 198]. Several others used the CPU cache to extract private keys from AES implementations [1, 15, 163]. All previous attacks focused on reading secret information across boundaries, either from other processes or other VMs running on the same physical machine. Gras et al. [54] broke ASLR within the JavaScript sandbox with an EVICT+TIME technique.

However, not only CPU caches are an attractive target for side channels. TLBleed [53] showed that the TLB can also be used to leak sensitive information with all CPU cache defenses deployed. The BTB (Branch Target Buffer) has also been shown to leak sensitive information through side-channel attacks [44]. Pessl et al. [138] presented DRAMA a cross-CPU side-channel attack exploiting the DRAM row buffer.

SGX enclave attacks with TLB as an attack vector Previous work has shown that the TLB is shared between the SGX enclave and the process it is running in [186]. Since they run in the same address space this means classic TLB attacks apply on SGX environment [53], similarly to breaking KASLR without KPTI enabled. Therefore, tagged TLBs do not enable new leakage in such a case, but can be interesting if the address space is isolated like with AMD SEV [87].

Defenses against timing side channels on CPU caches As CPU caches became a target for side-channel attacks, several defenses were proposed. All defenses focus on scenarios where several untrusted entities share hardware (e.g., multiple tenants in the cloud). Cache isolation is intended to protect a tenant against an attacker running on the same physical machine [17, 57, 91, 110, 143, 157, 162, 202]. Some of the existing defenses focus on isolating critical code sections and disallow leaking information through the cache while executing in isolation [17, 57, 157]. Others protect areas in memory from leaking information to the cache [91, 110]. Some claim to provide full isolation between untrusted VMs running on a multi-tenant system [143, 162, 202]. None of these defenses isolate the kernel against the user or fully defeat CPU cache side channels. Especially traces left in the cache by the MMU, not by the user itself, will not be affected by any of these defenses. We rely on information in the cache that is cached on every single address translation by the MMU. Providing cache isolation is limited by resources and therefore can only be provided for a limited time to execute security sensitive sections. Address translations are always present, therefore not just a small portion of the code can be isolated.

Concurrent work In concurrent work, Data Bounce [22] and EchoLoad [23] present side-channel attacks to bypass KASLR in face of KPTI (and absent CPU bugs like RIDL [181]). In contrast to TagBleed, both attacks rely on the current KPTI implementation leaving a few kernel pages mapped in the user-visible address space with the same KASLR entropy used for all the other kernel pages. As such, similar to traditional address probing attacks against KASLR [59, 71, 81], such attacks can probe for user-mapped kernel pages and indirectly infer that of the other kernel pages. In contrast, TagBleed's confused deputy attack can directly drop the entropy of kernel pages mapped and used only by the kernel, showing that even a perfect implementation of KPTI as well as the most recent mitigations against address probing attacks such as FLARE [23] are insufficient.

2.10 Conclusions

In this chapter, we demonstrated that isolating the address space organization of the kernel from that of user processes is not enough to prevent attackers from breaking randomization in the kernel (KASLR). Ironically, the one feature that is commonly hailed as the performance savior for kernel address space isolation, the presence of address space tags in modern TLBs, turns out to break its isolation guarantees. Our attack makes use of the fact that tagged TLBs allow attackers to observe kernel memory accesses if they occur in the same TLB set. Moreover, by reverse engineering the TLB architecture, we were able to infer part of the kernel's virtual address from knowing the TLB set. Complementing our side channel with second one (based on cache activity as a result of page table walks), we completely broke KASLR in the Linux kernel, even in the presence of advanced defenses and kernel page table isolation. In conclusion, since we demonstrated that we need to reconsider the current designs and countermeasures are invariably expensive and typically complicated, we now know that the transition from a unified address space organization to one where the kernel gets its own address space will be more costly than we thought.

3 Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel

Due to the high cost of serializing instructions to mitigate Spectre-like attacks on mispredicted conditional branches (Spectre-PHT), developers of critical software such as the Linux kernel selectively apply such mitigations with annotations to code paths they assume to be dangerous under speculative execution. The approach leads to incomplete protection as it applies mitigations only to easy-to-spot gadgets. Still, until now, this was sufficient, because existing gadget scanners (and kernel developers) are pattern-driven: they look for known exploit signatures and cannot detect more generic gadgets.

In this chapter, we abandon pattern scanning for an approach that models the essential *steps* used in speculative execution attacks, allowing us to find more generic gadgets—well beyond the reach of existing scanners. In particular, we present KASPER, a speculative execution gadget scanner that uses taint analysis policies to model an attacker capable of exploiting arbitrary software/hardware vulnerabilities on a transient path to control data (e.g., through memory massaging or LVI), access secrets (e.g., through out-of-bounds or use-after-free accesses), and leak these secrets (e.g., through cache-based, MDS-based, or port contention-based covert channels).

Finally, where existing solutions target user programs, KASPER finds gadgets in the kernel, a higher-value attack target, but also more complicated to analyze. Even though the kernel is heavily hardened against transient execution attacks, KASPER finds 1379 gadgets that are not yet mitigated. We confirm our findings by demonstrating an end-to-end proof-of-concept exploit for one of the gadgets found by KASPER.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Listing 2: Spectre-PHT BCB pattern, where attacker data bypasses an array bounds check in order to leak secret data.

3.1 Introduction

Ever since Meltdown and Spectre burst onto the scene in January 2018 [69, 95, 109], transient execution vulnerabilities have had the security community scrambling for solutions. While some of the vulnerabilities, such as Meltdown, can be mitigated fairly easily [33], this is not the case for others. In particular, Spectre-PHT (commonly referred to as Spectre-V1)—which leaks secrets by abusing the transient execution that follows a mispredicted conditional branch—cannot be fully eradicated without crippling performance. One possible mitigation is to forbear all speculation after a conditional branch. However, since speculative execution in modern CPUs is essential to performance and conditional branches are everywhere, it is crucial that such an expensive mitigation be applied only where necessary. The attack itself requires specific Spectre-PHT *gadgets*: that is, vulnerable branches which can indeed leak secrets through the microarchitectural state. Therefore, to maintain acceptable performance, we should only serialize or otherwise instrument these gadgets.

Beyond pattern-matching However, current methods to identify gadgets are limited as they are entirely pattern-driven. By searching for specific features of well-known Bounds Check Bypass (BCB) patterns (Listing 2)—e.g., suspicious copies from user space [25], potential out-of-bounds accesses [126], or attacker-dependent memory accesses [140]—they only approximate the presence of BCB gadgets. Our experiments show that today's state-of-the-art scanners [126, 140] yield a false positive rate of 99% (Section 3.9.1). In other words, 99% of the code snippets identified as gadgets cannot actually lead to information disclosure. Adding expensive defenses to such code snippets incurs substantial and unnecessary overhead. More, the false negative rate is up to 33%. In other words, they miss many vulnerable branches that should be protected.

Furthermore, existing approaches are limited in scope. They all broadly assume the same primitives as the traditional BCB pattern: direct attacker input, an out-of-bounds secret access, and a cache-based covert channel. Such primitives are of little use in the hardened Linux kernel since the large majority of BCB gadgets have been mitigated. However, this does not mean the kernel is free of Spectre-PHT gadgets. Far from it: the problem goes much deeper than BCB patterns. In reality, attackers do not care about patterns; they just want to find *any* instructions in the wake of *any* conditional branch, controllable by *any* means, which access secrets in *arbitrary* ways, and leak the secrets through *any* covert channel.

A principled approach In this chapter, we propose a novel approach for finding vulnerable gadgets in the kernel, abstracting away all pattern-specific details and instead precisely modeling the *essential steps* of a Spectre-PHT attack shown in Figure 3.1.

Step 1: *Inject* controlled values into transient execution.
Step 2: Force the execution to *access* a secret.
Step 3: Force the execution to *leak* the secret.

Figure 3.1: Essential steps in a Spectre attack.

Around these steps, we use targeted taint analysis policies to generically model the effects of *arbitrary hardware/software vulnerabilities* on a transient path.

To evaluate the approach, we present Kasper, a Spectre-PHT gadget scanner. By modeling the effects of vulnerabilities on a transient path—e.g., memory errors [27, 95], load value injection [177], cache-based covert channels [95], MDS-based covert channels [22, 150, 181], and port contention-based covert channels [13, 46]—Kasper finds gadgets well beyond the scope of existing approaches.

Moreover, rather than focusing on user processes (to which our approach is easily applicable), we developed our techniques specifically for the Linux kernel—a high-value target like few other programs; since the kernel has access to all memory in the system, a kernel attacker can target data from any of the running processes in the system. Existing kernel gadget scanners however, are all based on static analysis [25, 93], which previous work observes is imprecise [126, 140]. Instead, we take a dynamic analysis-based approach, which simply requires us to build the kernel with Kasper support, fuzz the syscall interface (to simulate the possible coverage from a user-to-kernel attacker), then Kasper will report gadgets at runtime. Even though the kernel has undergone intensive scrutiny and mitigation efforts to neuter all gadgets, we still find 1379 gadgets in an automated fashion—including many gadgets which would be non-trivial to find statically.

Furthermore, we present a case study of a gadget found by Kasper which is pervasive throughout the codebase and non-trivial to mitigate. In total, we have found 218 instances of the demonstrated gadget sprinkled all over the kernel. We demonstrate the efficacy of our approach by presenting a proof-of-concept exploit of this gadget.

Contributions We make the following contributions:

- We present taint-assisted generic gadget scanning, a new approach to identify pattern-agnostic transient execution gadgets that stem from arbitrary software/hardware vulnerabilities on a transient execution path.
- We present Kasper, an implementation[1] of our approach for the Linux kernel.
- We evaluate Kasper on a recent version of the Linux kernel to identify 1379 previously unknown transient execution gadgets and present a proof-of-concept exploit for one of the gadgets. The Linux kernel developers assessed mitigations for the disclosed gadgets and requested access to Kasper for mainline kernel regression testing moving forward.

The rest of this chapter is organized as follows. Section 3.2 presents background on the attack primitives modeled by Kasper. Section 3.3 describes the threat model. Section 3.4 defines the problem scope for Kasper. Section 3.5 describes the design of Kasper at a high level. Section 3.6 explains speculative emulation, including unique implementation challenges posed by the kernel. Section 3.7 explains Kasper's vulnerability detectors and the taint policies which model their effects. Section 3.8 briefly

[1]Kasper is available at `https://www.vusec.net/projects/kasper`.

Speculation	Attacker input	Secret output
	ARCH:[a]	
PHT	**USER**, FILE, NET,	**CACHE**
BTB	DEV, **MASSAGE**	**MDS**
RSB		**PORT**
STL	**LVI**	AVX
	FPVI	

[a] We only list the set of ARCH inputs that are relevant to the kernel.

Table 3.1: Overview of the possible primitives of a Spectre gadget. Kasper models the primitives in bold.

describes implementation details. Section 3.9 evaluates the efficacy of KASPER compared to existing approaches and KASPER's gadgets found in the kernel. Section 3.10 discusses the limitations of our approach. Section 3.11 presents a case study of a gadget found by KASPER. Finally, Section 3.13 concludes.

3.2 Background

In this section, we will first provide a background on the components involved in a Spectre attack and the defenses which combat them, motivating the need for Spectre-PHT gadget scanners. Then, we will provide background on previous gadget-scanning tools, highlighting the need for a pattern-agnostic gadget scanner.

3.2.1 Speculative execution attacks and defenses

Spectre attacks exploit the fact that modern processors predict the outcome of operations such as conditional branches, and speculatively continue executing as if its prediction is correct. If it turns out the prediction was incorrect, the processor reverts the results of any speculative operations and restarts from the correct state. The modifications made by the incorrect path to the microarchitectural state, however, can be examined by an attacker using a covert channel to leak sensitive information.

We propose that the underlying primitives used by a Spectre gadget—i.e., the type of speculation it abuses, the type of attacker data it depends on, and the type of leakage it exploits—define a Spectre variant. The interactions between the different primitives affect whether the variant is indeed exploitable and whether it is easily mitigated. We summarize these primitives in Table 3.1.

Speculation type Spectre variants based on speculation from anything other than the *Pattern History Table*—that is, Spectre-*BTB* [95], Spectre-*RSB* [99, 118], and Spectre-*STL* [70]—are easily mitigated with relatively low overhead using microcode updates [77] or software updates such as retpolines [76] and static calls [203]. Unfortunately, this is not so for Spectre-PHT. Spectre-PHT gadgets can be mitigated by adding an `lfence` instruction after conditional branches; this approach does not scale, however, as adding an `lfence` after every conditional branch would incur up to a massive 440% overhead [125]. Hence, rather than avoiding speculation altogether via `lfence`, Spectre-PHT is better addressed by mitigating specific speculative operations—that is, either at the point where attacker data is used or where secret data is leaked.

Attacker input type Attacker data may reach a kernel gadget in a variety of ways—either via architectural or microarchitectural means. First, an attacker may pass data to the kernel via well-defined interfaces such as *userspace*, *files*, the *network*, or malicious *devices*. Second, data can come from such places as normal, but then an attacker may inject it into a victim kernel thread via targeted *memory massaging* [27, 28, 193, 206]—wherein the attacker lands the data into a specific place on the kernel's stack or heap, in the hopes that later on, a bug such as an out-of-bounds read or an uninitialized read (bugs that are relatively common on a transient path) will eventually use the malicious data. Third, using thread that shares a simultaneous multithreaded (SMT) core with a kernel thread (i.e., where the core executes instructions from both the attacker thread and the victim thread at the same time), the attacker may inject data into the victim's transient path via *load value injection (LVI)* [177]—wherein the attacker issues a sequence of faulting stores, filling the CPU's load port with unresolved data dependencies; meanwhile, if the kernel simultaneously loads from the same faulting address, the CPU will inadvertently serve malicious data to the kernel. Concurrent to our work, *floating point value injection (FPVI)* [141] similarly demonstrates this issue for floating point values. Note that when we will discuss gadget exploitability, we will group together variants using *architectural* input, since a Spectre gadget is agnostic to architectural-level semantics, and will execute the same regardless of whether the input comes from e.g., user space or memory massaging.

A widely-used mitigation to pacify attacker input is to prevent certain transient array accesses from accessing secret data out-of-bounds by forcing the access to stay in-bounds via a masking operation. The Linux kernel uses user pointer sanitization and the macro `array_index_nospec` for this purpose. This approach however, does not generalize well to non-array transient accesses.

Secret output type Secrets may leak from a kernel gadget in a variety of ways. First, a gadget may rely on a *cache*-based channel [95], wherein the victim dereferences a secret, thereby leaving a trace on the data cache (or TLB); an attacker can then recover this secret through methods such as FLUSH+RELOAD [195]. Second, a gadget may rely on a *microarchitectural data sampling (MDS)*-based channel [22, 150, 181], wherein the victim simply accesses a secret, causing the CPU to copy the secret into its load buffer (or line fill buffer, store buffer, etc.); meanwhile, an attacker can co-locate a thread on an SMT core and issue a conflicting load. As a result, the CPU will inadvertently serve the secret data to the attacker, who can then use it to leave a trace on their own FLUSH+RELOAD buffer for recovery. Third, a gadget may rely on a *port contention*-based channel, wherein the victim—depending on a secret—either executes one set of instructions or another; meanwhile, an attacker can co-locate a thread on an SMT core and issue instructions that compete for the same execution units (i.e., ports) as the victim's instructions. Then, the attacker can use timing information to infer which instructions the victim executed, and hence, learn a bit of the secret [13, 46]. Finally, a gadget may rely on an *AVX*-based channel, which exploits the timing of AVX2 instructions [152].

The kernel flushes CPU buffers to mitigate same-thread MDS channels. Hardware updates for the most recent generation of CPUs mitigate cross-thread MDS (and LVI) channels. For the same protections, older CPUs must disable hyper-threading, resulting in massive performance penalties (not the default on Linux). The reality that these defenses are only on by default on the newest CPUs, and that PHT gadgets cannot be

```
x = *ptr ;
if (x < size) {
  y = arr1[x];
  z = arr2[y]; }
```

(a) Gadget that is difficult to detect statically because it is unclear whether `*ptr` is attacker-controllable.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y & MASK ]; }
```

(b) Gadget that is undetectable by SpecFuzz [126] because its in-bound *leak* eludes code sanitizers.

```
x = 1000 ;
if (x < size) {
  y = arr1[x];
  z = arr2[y]; }
```

(c) Gadget that is not controllable by an attacker, yet falsely reported by SpecFuzz [126] because it yields an out-of-bounds access.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x & MASK ];
  z = arr2[y]; }
```

(d) Gadget that is mitigated by the masking operation, yet falsely reported by SpecTaint [140] because there is a direct dataflow from `x` to `y` to `arr2[y]`.

```
if ( addr_is_mapped(ptr) ) {
  x = *ptr;
  y = arr1[x];
  z = arr2[y]; }
```

(e) Gadget that existing approaches cannot detect. `*ptr` gives a transient page fault, so an attacker can *inject* data for `x` via LVI [177].

Listing 3: Gadgets that differ slightly from the basic BCB pattern in Listing 2, and therefore foil existing approaches.

systemically mitigated via `lfence`, `array_index_nospec`, and user pointer sanitization, highlights the pressing need for Spectre-PHT gadget scanners.

3.2.2 Gadget scanning

Existing gadgets scanners, however, cannot accurately identify gadgets that stray even slightly from the basic BCB pattern in Listing 2.

Static analyses Existing static analyses find gadgets through pattern-matching of source code [25, 56], pattern-matching of binaries [30], static taint analysis [185], and symbolic execution [60, 184]. Concurrent work [93]—which goes beyond the BCB pattern, and instead targets a type-confusion pattern—similarly uses such approaches. These approaches, however, all suffer from the fundamental limitation of static analysis: assumptions to compensate for unknowns at compile time will severely hamper soundness and/or completeness. For example, consider the gadget in Listing 3a, which loads its input from a source that is unknown at compile time. In theory, an advanced analysis such as symbolic execution could deduce whether `*ptr` is indeed attacker-controllable by verifying the controllability of *every possible* value in *every possible* location of `*ptr`; however, this kind of analysis cannot scale to the complex and massive kernel codebase [206], so it instead must ultimately either assume that `*ptr` is attacker-controllable (and risk false positives) or that it is not (and risk false negatives). Such scalability issues are inherent to any sufficiently complex static analysis, leading to imprecise points-to analyses, inaccurate call graph extractions, and ultimately, imprecise gadget identifications. In Section 3.D, we describe a gadget found by our dynamic analysis which relies on an indirect call, thereby thwarting static call graph extraction, and hence, identification by static analysis altogether.

Dynamic analyses To overcome the limitations of static analysis, recent work instead opts for dynamic analysis. Their approaches however, fall short since they are pattern-driven and do not model the underlying semantics of gadgets.

For example, since the BCB pattern has an out-of-bounds access, SpecFuzz [126] uses code sanitizers to report as gadgets any speculative out-of-bounds accesses. By targeting this single behavior however, it incurs false negatives for gadgets such as the one in Listing 3b, whose *leak* instruction is in-bounds. Furthermore, it incurs false positives for any unrelated out-of-bounds accesses, such as the one in Listing 3c, even though it is entirely uncontrollable by an attacker.

Another property about the BCB pattern is that there is a direct dataflow from an attacker-controllable value, to a secret, and finally to a *leak* instruction. Targeting this single property, SpecTaint [140] taints attacker-controllable values (`x`), then taints as a secret any attacker-dependent loads (`y`), then reports as a gadget any secret-dependent accesses (`arr2[y]`). This policy however, assumes very specific patterns and also incurs false positives since not every attacker-dependent load accesses secret data. For example, consider the attacker-dependent load in Listing 3d: even though the masking operation prevents it from accessing secret data, SpecTaint falsely reports this mitigated gadget as an exploitable gadget.

3.3 Threat Model

We consider a local unprivileged attacker with the ability to issue arbitrary system calls to a target kernel free of (exploitable) software bugs. The attacker aims to gain a memory leak primitive by exploiting a transient execution gadget in the kernel. As an operating system kernel, we focus on a recent Linux kernel (in our case 5.12-rc2) with default configurations, including all the mitigations against transient execution attacks enabled, such as user pointer sanitization, `lfences`, `array_index_nospec`, retpolines, static calls, etc. Additionally—although overlooked by the Linux kernel's transient execution threat model [170, 171], which only considers attacker input from user space and MDS/cache-based covert channels—we consider an attacker able to: (1) inject data via memory massaging, (2) inject data via LVI, and (3) exfiltrate data via port contention-based covert channels. Note that on the most recent generation of CPUs, LVI and MDS are mitigated in-silicon to a large extent, so our results for LVI and MDS do not apply to the full extent to such CPUs.

3.4 Problem Analysis

Not only do existing approaches fail to identify gadgets which stray slightly from the basic BCB pattern (Section 3.2.2), they do not even attempt to model primitives beyond those it uses—i.e., where an attacker directly passes data to the victim, causing a cache-based leak. For example, consider the gadget in Listing 3e, where an attacker can inject data via LVI [177]. No existing approach attempts to model LVI, and hence, gadgets such as these are missed. In reality, attackers have many such primitives at their disposal (Table 3.1)—such as memory massaging, LVI, MDS, and port contention—all of which may yield gadgets that fall outside the scope of existing approaches.

To reason about the various combinations of primitives which are possible in a Spectre gadget, we express Spectre variants as a triple in terms of these primitives. For example, we consider the original Spectre-PHT attack [95] which exploited the BCB

Spectre variant	Described in:	Verified signal?	Existing scanning techniques?
PHT-ARCH-CACHE	[95]	✓	✓
PHT-ARCH-MDS	[22]	✓	-
PHT-ARCH-PORT	[46]	✓	-
PHT-LVI-CACHE	This chapter[a]	✓	-
PHT-LVI-MDS	None[a]	-[b]	-
PHT-LVI-PORT	None[a]	-	-

[a] The demonstrated LVI attack [177] exploits an LVI-CACHE gadget.
[b] We verified a signal for LVI-MDS, but not for PHT-LVI-MDS.

Table 3.2: Exploitability of the Spectre variants that are composed of the primitives which Kasper models. Despite the signal on 4 variants, existing scanning techniques target only PHT-Arch-Cache.

pattern to be a PHT-ARCH-CACHE gadget because it: (1) exploits prediction from the PHT, (2) depends on architecturally-defined attacker input, and (3) leaks the secret via the cache. Similarly, Fallout [22] uses a PHT-ARCH-MDS gadget and SpectreRewind [46] uses a PHT-ARCH-PORT gadget. Other Spectre attacks, including those using other speculation types, can be expressed as a triple in this way; e.g., SMoTherSpectre [13] exploits a BTB-ARCH-PORT gadget. Furthermore, even non-Spectre attacks can be expressed as a tuple of the last two primitives; e.g., a standard cache attack [137] exploits an ARCH-CACHE gadget.

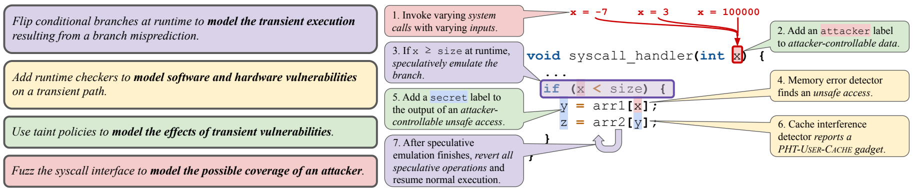
Finally, assuming a gadget scanner can model all such primitives, we would first need to verify that all the possible variant combinations are indeed exploitable. In Table 3.2, we summarize the exploitability of the Spectre variants made up of the primitives modeled by KASPER. The first three rows, as described above, are based on attacks from previous work. Beyond existing work, we were able to verify a signal for a PHT-LVI-CACHE gadget, but not for a PHT-LVI-MDS gadget or a PHT-LVI-PORT gadget. Hence, KASPER models the first four variants of the table.

3.5 Overview

The key insight to our approach is that by generically modeling the vulnerabilities that an attacker can use in each step of a Spectre attack, we can precisely identify gadgets. In particular, we use: (1) *speculative emulation* to model transient execution, (2) *vulnerability detectors* to model various software and hardware vulnerabilities, (3) *dynamic taint analysis* to model the architectural and microarchitectural effects of such vulnerabilities, and (4) *fuzzing* to model the possible coverage of an attacker. Figure 3.2 presents an example of how these components interact to identify a gadget in a system call handler.

Modeling transient execution To model branch mispredictions, we invert conditional branches at runtime and emulate the corresponding transient execution by taking a checkpoint at the point of 'misprediction' and executing the code that would be executed speculatively. We roll back to resume normal execution when the speculative window closes.

Speculative emulation is not trivial in general and this is especially true for speculative emulation in the kernel, where complexities such as device interactions, exceptions,


Flip conditional branches at runtime to ***model the transient execution*** *resulting from a branch misprediction.*
Add runtime checkers to ***model software and hardware vulnerabilities*** *on a transient path.*
Use taint policies to ***model the effects of transient vulnerabilities.***
Fuzz the syscall interface to ***model the possible coverage of an attacker.***
1. Invoke varying *system calls* with varying *inputs*.
x = -7
x = 3
x = 100000
2. Add an attacker label to *attacker-controllable data*.
3. If x ≥ size at runtime, *speculatively emulate the branch*.
4. Memory error detector finds an *unsafe access*.
5. Add a secret label to the output of an *attacker-controllable unsafe access*.
6. Cache interference detector *reports a PHT-USER-CACHE gadget*.
7. After speculative emulation finishes, *revert all speculative operations* and resume normal execution.

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Figure 3.2: Components used by our approach and how they interact to detect a PHT-User-Cache gadget in a system call handler.

and inline assembly all pose challenges. We explain how our approach overcomes these challenges in Section 3.6.

Modeling software and hardware vulnerabilities By modeling transient execution at runtime, we expose transient code paths to runtime checkers to generically identify software and hardware vulnerabilities which would otherwise remain undetectable.

Our current prototype uses: (1) a memory error detector to target *software vulnerabilities* in the shape of transient out-of-bounds and use-after-free accesses, (2) an LVI detector to target *hardware vulnerabilities* via transient faulting accesses, and (3) a covert channel detector for *hardware vulnerabilities* via cache-based, MDS-based, or port contention-based channels. We explain our detectors in more detail in Section 3.7.1.

Modeling the effects of transient vulnerabilities By detecting software and hardware vulnerabilities, we can design taint policies around the essential steps of a Spectre attack (Figure 3.1) to reason about the effects of such issues.

For example, our policies may handle a transient invalid load in different ways: (1) our LVI detector may taint the load with an `attacker` label to indicate that the attacker may *inject* the value via LVI; (2) our memory error detector may taint the load with a `secret` label to indicate that it may *access* arbitrary memory; or (3) our covert channel detector may report the load as an MDS-LP [181] covert channel to indicate that it may *leak* `secret` data. We explain how our taint policies reason about the interactions between different vulnerabilities on a transient path in Sections 3.7.2–3.7.4, including a summary of the policies in Figure 3.3.

Modeling the possible coverage of an attacker By modeling the effects of transient vulnerabilities in the kernel, we can use a user-to-kernel fuzzer to only model the vulnerabilities that are reachable by a user-space attacker issuing arbitrary syscalls. We describe the fuzzer and implementations details in Section 3.8, including a summary of the end-to-end pipeline in Figure 3.4.

3.6 Speculative Emulation

To emulate speculative execution, we need to be able to execute possible execution paths that would otherwise not be executed architecturally. Specifically, on a branch misprediction, the processor first executes the *wrong* code path until the speculation is eventually squashed. The processor then continues executing the correct side of the branch. To model such branch mispredictions, we invert conditional branches at

```
  x = get_user(ptr);
  // Sets in_checkpoint
  if (!in_checkpoint) new_checkpoint();
L1:
  if ((x < size)
      XOR in_checkpoint) {
    y = arr1[x];
    z = arr2[y];
  }
  // Unsets in_checkpoint
  if (in_checkpoint) {rollback(); goto L1;}
```

Listing 4: Conditional branch instrumented to enable speculative emulation.

runtime in software and to be able to squash the incorrect execution path, we rely on software-based memory checkpointing. Thus, at runtime we will first execute the wrong code path to emulate speculation and when speculation is squashed, we execute the correct code path.

Listing 4 shows how we instrument the example of Listing 2 to support speculative emulation. If `x` $\geq$ `size` at runtime, we:

1. Start a checkpoint immediately before the branch via the call to `new_checkpoint`.
2. Simulate a branch misprediction by flipping the branch via `XOR in_checkpoint`.
3. Emulate speculative execution along the *taken* code path.
4. Simulate a speculative squash by rolling back to the saved checkpoint via `rollback(); goto L1;`
5. Finally, continue normal execution along the correct *not-taken* code path.

3.6.1 Transactions and rollbacks

To ensure correct execution, speculative execution is inherently transactional from an architectural point of view; either all the instructions after a predicted branch commit (i.e., correct prediction) or none does. To provide this semantic, we opted for a compiler-based memory checkpoint-restore mechanism to build a notion of transactional execution. We implemented our solution with a combination of LLVM compiler passes and a runtime component. On a rollback, we need to restore the original state before the misprediction. KASPER does this by saving all relevant registers when starting a checkpoint and tracking all memory changes in an undo log in preparation for a replay on rollback, similar to prior work [126]. However, doing so in the kernel presents unique challenges which we address in Section 3.6.2.

Stopping Speculative Emulation One question we need to answer is when to roll back speculative emulation. Speculative execution is limited to a certain number of *micro-ops* executed depending on the ReOrder Buffer (ROB). At compile time we approximate this behavior with the number of executed LLVM instructions. At the start of every basic block, the number of executed LLVM instructions from the beginning of the checkpoint is compared against a configurable threshold to decide when to abort speculative emulation.

```
x = get_user(ptr);
if (x < size) {
  foo = *bar; // Page fault if bar is invalid
  y = arr1[x]; // Would not execute
  z = arr2[y]; // Would not execute
}
```

Listing 5: Executing past a potential page fault in speculative emulation.

While this does not map to the exact behavior of hardware, a reasonable approximation is good enough, and we can easily configure the threshold to be conservative to avoid false positives or more permissive to decrease the likelihood of false negatives. Similarly, we define an upper limit of call depth, simulating the behavior of the Return Stack Buffer (RSB).

Exception handling Exceptions do not necessarily stop speculative execution, as instructions past an exception can still be executed out-of-order [109, 181]. Our initial evaluation showed that the majority of exceptions raised during kernel speculative emulation are page faults, due to a corrupted memory address within speculation. We hence designed a *page fault suppression* for speculative emulation to execute past raised page faults—and simply stop emulation in the exception handler in the other cases. To avoid page faults, KASPER validates the pointer before dereferencing and replaces it with a valid dummy pointer if it is invalid. Listing 5 shows an example gadget that we would miss without this improvement. Another advantage of dedicated page fault handling is the ability to model common hardware vulnerabilities, as discussed later.

3.6.2 Challenges unique to the kernel

Implementing speculative emulation for the kernel introduces new challenges compared to userland. For example, a user program only operates on its own accessible memory while the kernel is able to access the entire range of memory. The kernel is therefore responsible for ensuring full memory integrity while user processes are only aware of their own address space, prompting special treatment, as discussed next. As shown in Section 3.A.2, these strategies only contribute to a negligible amount of rollbacks.

Non-conventional memory accesses The kernel uses device or I/O mapped memory to communicate with external devices. Memory writes to such memory cannot be rolled back by simply writing back the original value. Rollback after writes to such memory ranges would require also rolling back the dedicated device. Since I/O communication does not happen often in most syscall handlers, we gracefully stop emulation in those cases without a big loss in speculative code coverage.

Low-level code and mitigations The Linux kernel codebase is sprinkled with low-level assembly and transient mitigation code. To ensure speculative emulation correctness, it is important to handle such snippets properly. To this end, KASPER conservatively treats assembly code as a speculative emulation barrier (i.e., forcing rollback) by default and implements dedicated handling for the common assembly snippets to allow their use in speculative emulation. This is to avoid unnecessary rollbacks and maximize

```
static void arch_atomic64_inc(atomic64_t *v) {
  kasper_track_store((void*)&v->counter);
  asm volatile(LOCK_PREFIX "incq %0"
    : "=m" (v->counter)
    : "m" (v->counter) : "memory");
}
```

Listing 6: Enabling a common assembly sequence in speculative emulation.

speculative code coverage. Listing 6 illustrates an example, with custom handling for atomic increment instructions. Since the assembly is not instrumented, we manually preserve the changed memory location. As for mitigations, we observe the relevant ones (`lfence`, `stac`, etc.) are all implemented in assembly snippets, thus our default assembly handler (i.e., stop speculative emulation) already models them correctly with no special treatment needed.

Concurrency Unlike many common user programs, the Linux kernel is a highly concurrent piece of software, with multiple threads running in parallel on different CPU cores and hardware (e.g., timer) interrupts causing asynchronous event handling. Taking correct checkpoints in face of concurrency poses a fundamental challenge for the correctness of checkpointing [158] (and speculative emulation in our case). To address this challenge, we disable SMP support in the kernel and force single-core execution. To handle hardware interrupts, we instrument the interrupt handler to: (i) record interrupt information; (ii) stop speculative emulation and resume execution at the last checkpoint; (iii) replay the interrupt as though it was delivered before entering speculative emulation. These steps are all crucial to ensure correct checkpointing *and* kernel execution (i.e., no interrupts are lost).

3.7 Taint Policies

To model each of the three steps in Figure 3.1, we can draw upon previous work that uses information flow policies to detect sensitive data leakage [38, 42, 124, 147, 197], and use a basic framework consisting of three types of policies:

1. Taint any data *injected* by an attacker with an `attacker` label.
2. Taint secret data (i.e., data *accessed* via an `attacker` pointer) with a `secret` label.
3. Report gadgets that *leak* such `secret` values.

However, to detect leakage via generic *transient* execution gadgets, we must refine our framework with taint policies that account for arbitrary vulnerabilities exploited on a transient path that enrich the attacker's capabilities. In particular, with detectors capturing the effects of vulnerabilities in software (i.e., memory errors in the current implementation) or hardware (i.e., LVI, MDS, port contention, and cache covert channels in the current implementation), we are able to model increasingly complex gadgets. In the following, we first detail our current detectors and then discuss how our *taint manager* enforces the detection-based taint policies (for *injection*, *access*, and *leakage*). Many more detectors can be deployed in the future for similar types of vulnerabilities, such as TLB, branch target buffer or DRAM bank side channels. With KASPER, we currently

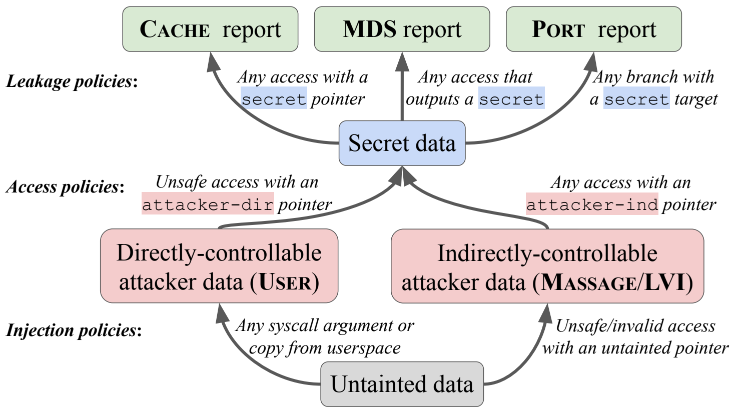

CACHE report
MDS report
PORT report
Leakage policies:
Any access with a secret pointer
Any access that outputs a secret
Any branch with a secret target
Secret data
Access policies:
Unsafe access with an attacker-dir pointer
Any access with an attacker-ind pointer
Directly-controllable attacker data (USER)
Indirectly-controllable attacker data (MASSAGE/LVI)
Injection policies:
Any syscall argument or copy from userspace
Unsafe/invalid access with an untainted pointer
Untainted data

Figure 3.3: *Injection*, *access*, and *leakage* taint policies which describe how data rises from an untainted label, to an `attacker` label, to a `secret` label, and finally to a gadget report.

focus on the presented detectors to demonstrate its effectiveness. We summarize our policies in Figure 3.3.

3.7.1 Vulnerability detectors

Memory error detector Our current memory error detector supports the detection of *unsafe* (i.e., out-of-bounds or use-after-free) accesses on a transient execution path. This is done by running sanitizers during speculative emulation and reporting information about unsafe load/store operations and addresses to the taint manager. Information about unsafe accesses is useful to model a degree of attacker's control on a given address. For instance, if an out-of-bounds detection is taken as evidence that an attacker can make a load address go out-of-bounds at will, that address is a *controlled pointer* that could be used for *injection* (via memory massaging), *access* (via unauthorized reads), and *leakage* (via vulnerabilities such as MDS).

LVI detector Our current LVI [177] detector supports the detection of *invalid* loads able to trigger LVI on a transient execution path. We tested the (known) LVI triggering conditions and could reproduce only two cases of such invalid loads on a mispredicted branch, also observed in prior work [22]: (i) loads incurring an SMAP fault (i.e., loading a user address with SMAP on—default); (ii) loads incurring a noncanonical address fault (i.e., loading an address outside the canonical 48-bit address space). We also verified these loads can be poisoned by attackers for *injection* of transient values via MSBDS exploitation [22]. To identify such *invalid* loads, we target loads with user/noncanonical addresses (a subset of unsafe loads). However, by default we omit NULL pointer dereferences. Allowing them leads to plenty of usable gadgets, however those require an attacker to map the 0x0 page (i.e., to cause an SMAP fault). The latter is forbidden by default on Linux, but possible on systems with `mmap_min_addr=0`.

Cache interference detector Our current cache interference channel detector supports the detection of secret-dependent loads/stores on a transient execution path.

For all such memory accesses, an attacker can achieve *leakage* of the target (secret-dependent) address with a classic cache [50] or TLB [115] attack. Hence, to identify these loads/stores, we can simply report those that use a pointer tainted with the `secret` label during speculative emulation.

MDS detector Our current MDS detector supports the detection of secret-accessing loads/stores on a transient execution path whose data can be leaked by an MDS exploit (i.e., sampling data via various CPU internal buffers, such as LFB [181], SB [22], etc.). We tested the (known) MDS triggering conditions and verified that an attacker can achieve *leakage* of data from arbitrary loads/stores on a mispredicted branch. Hence, to identify these loads/stores, we report cases where the pointer is under the control of the attacker during speculative emulation. These pointers are tainted with the `attacker` label and/or leading to unsafe loads/stores.

Port contention detector Our current port contention detector supports the detection of secret-dependent branches on a transient path. For such branches, an attacker can *leak* a bit of the secret by issuing instructions that use the same execution units (i.e., ports) as the branch targets' instructions. Hence, to identify these branches, we report those that use a target tainted with a `secret` during speculative emulation. Note that future work could refine the set of reported port contention gadgets by using static analysis to determine whether a secret-tainted branch's possible targets are indeed SMoTHER-differentiable [13]—i.e., whether there is a sufficiently large enough timing difference generated by the port contention of one branch target to tell it apart from the port contention of another branch target.

3.7.2 Injection policies

Our unified *injection policies* combine our basic policy of tainting directly-controllable data injected via external input (e.g., syscall arguments) with our detection-based policies of tainting data injected via hardware/software vulnerabilities (e.g., memory massaging or LVI) which we refer to as indirectly-controllable attacker data. We distinguish between directly-controllable attacker data and indirectly-controllable attacker data because our *access policies* make use of this distinction, as explained later (Section 3.7.3). We refer to these labels as `attacker-dir` and `attacker-ind` (and `attacker` to refer to either one).

Injection Policy I: Directly-controllable data. *We taint all data which is directly-controllable from user space—that is, syscall arguments and the output of functions such as `get_user`, `copy_from_user`, and `strncpy_from_user`—with the `attacker-dir` label.*

Injection Policy II: Indirectly-controllable data. *We taint all data which is indirectly-controllable from an attacker, as modeled by our memory error and LVI detectors, with the `attacker-ind` label.*

Data injected via memory errors We use our memory error detector to identify unsafe loads during speculative emulation; by default, upon detection and if the pointer is untainted, we add the `attacker-ind` label to the loaded value. Note that if the pointer is already tainted with an `attacker` label, such label is automatically propagated (pointer tainting enabled for loads). These policies are to model an attacker massaging controlled data in the target memory location. Since our memory error detector operates on heap

```
if (a < size) {
   b  = arr1[a]; // Transient out-of-bounds
↪  access loading memory massaged data
   c  = arr2[ b ];
   d  = arr3[ c ];
}
```

(a) Gadget where an attacker can inject data via memory massaging by taking advantage of a transient out-of-bounds read.

```
if (addr_is_mapped(ptr)) {
   x  = *ptr; // Transient faulting accessing
↪  loading LVI-injected data
   y  = arr2[ x ];
   z  = arr3[ y ];
}
```

(b) Gadget where an attacker can inject data with LVI by taking advantage of a transient faulting load (see Listing 3e).

Listing 7: Gadgets that demonstrate the injection policies.

and stack, these policies provide the attacker with plenty of exploitation strategies [27, 28, 193]. As an example, consider the gadget in Listing 7a which loads the attacker input by reading a heap object out-of-bounds, thereby possibly reading data from another object placed by the attacker using memory massaging. Attackers able to target heap memory massaging may gain indirect control over the data at `arr1[a]` and inject it into the otherwise-benign gadget.

Data injected via LVI Similarly, we use our LVI detector to detect invalid loads during speculative emulation; by default, upon detection and if the pointer is untainted, we add the `attacker-ind` label to the loaded value (again existing `attacker` labels are propagated via pointer tainting). Consider the gadget in Listing 7b, which loads attacker-controlled data via a transitive faulty access, thereby retrieving a value from the store buffer. Attackers capable of LVI may gain indirect control over the data at `a->bar` and inject it into the otherwise-benign gadget.

In addition to tainting unsafe (including invalid) loads as `attacker-ind` data, we could also taint them as `attacker-ind` data loads whose pointer is tainted with an `at tacker-dir` label. This is because if an attacker controls a pointer, then the attacker could hypothetically force it to load data through LVI or memory massaging. However, as explained later (Section 3.7.3), such loads are already tainted with the `secret` label and such modeling would be redundant (and lead to gadget over-reporting). We could also avoid using our detectors and only rely on pointer tainting, but this strategy leads to unnecessary false negatives (i.e., noncontrollable pointers still able to read memory massaged data or LVI data). Finally, we could disable pointer tainting on loads, but this strategy still misses cases of (safe/valid) loads where the attacker can control the loaded value.

3.7.3 Access policies

The raising of `attacker` labels to `secret` labels in access instructions is dependent on (1) the *type of control* the attacker has on the pointer, as determined by its taint sources, and (2) the *degree of control* the attacker has on the pointer, as approximated by the memory error detector.

Access Policy I: Raising directly-controllable attacker data. *If a load is unsafe and has a pointer with an `attacker-dir` label, then we add the `secret` label to the loaded value.*

We use both taint information and memory error detection to identify secret accesses. We observed that using only memory error detection leads to many false pos-

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x & MASK]; // In-bounds access
  z = arr2[y];
}
```

Listing 8: Gadget that is mitigated via a masking operation will not be falsely report as a gadget (see Listing 3d).

itives. This is because the attacker may not have enough control over the pointer to leak arbitrary data. Using only taint information, on the other hand, still leads to false positives, especially in the kernel. Indeed, the kernel contains many pointer masking operations (for input sanitization), which have the effect of keeping many controlled pointers always safe even in transient execution. Both strategies are an approximation of controllability: using only memory error detection but then flagging cases with fixed addresses as an indication for the lack of the attacker's control, or using only tainting but then flagging cases with limited controllability, as an indication that an attacker will never be able to promote the access to go out-of-bounds. While state-of-the-art solutions [126, 140] have focused on these two extremes, we will later show there is no one-size-fits-all strategy and that different conditions require different treatments.

For example, consider the gadget in Listing 8, which is benign due to the masking operation which keeps the access in-bounds. In the gadget, we do not raise the `attacker-dir` label to a `secret` label (i.e., use only taint information) because it could never lead to an out-of-bounds access. In doing so, we avoid false positives which would arise from reporting harmless gadgets.

Access Policy II: Raising indirectly-controllable attacker data. *If a load has a pointer with an `attacker-ind` label, then we always add the `secret` to its output.*

Unlike the taint policies for raising directly-controllable data, we do not use memory error detectors in this case because indirectly-controllable data is generated (barring actual architectural bugs) within the same speculation window as the access instruction. Code paths using such transiently-injected data typically break global code invariants and we observed them to be rarely subject to pointer masking operations. As such, these paths offer almost unrestricted control to the attacker—unlike, say, syscall arguments and data loaded from usercopy functions, which kernel developers take efforts to sanitize against traditional exploits.

Note that the indirectly-controllable data which is loaded during the analysis is not generated by the user-space attacker. It is either loaded from a code sanitizer's redzone or a similar dummy region (as described in Section 3.6). Hence, its possible values during analysis are limited, so any restriction based on a memory error detector would simply test against the values incidentally loaded at runtime, rather than any meaningful values injected by a user-space attacker.

With this policy, we avoid false negatives which would arise from failing to identify an indirectly-controllable gadget because it incidentally did not lead to an unsafe *access* instruction.

```
x = get_user(ptr);
if ( x < size) {
   y = arr1[ x ];
   z = arr2[ y ]; // Leaks via cache
}
```

(a) Gadget leaking a secret via a cache-based covert channel.

```
x = get_user(ptr);
if ( x < size) {
   y = arr1[ x ]; // Leaks via MDS
}
```

(b) Gadget leaking a secret via an MDS-based covert channel.

```
x = get_user(ptr);
if ( x < size) {
   y = arr1[ x ];
   if ( y ) {
     ... // Leaks via port contention
} }
```

(c) Gadget leaking a secret via a port contention-based covert channel.

Listing 9: Gadgets that demonstrate the leakage policies.

3.7.4 Leakage policies

We use our hardware vulnerability detectors to identify gadgets that use cache-based, MDS-based, or port contention-based covert channels to leak `secret` information.

Leakage Policy I: Identifying cache-based gadgets. *If a memory access has a pointer with a `secret` value, then we report it as a cache-based gadget.*

Hence, by propagating taint from `attacker`-controlled sources to dependent `secret` accesses, we find the simple Spectre-BCB gadget from Listing 2 by propagating taint as in Listing 9a. Similarly, if `*ptr` in Listing 3a is `attacker`-controlled, the policy also identifies the gadget.

Unlike *access* instructions, *leak* instructions require no memory error detector—only our simple cache interference detector. Such instructions will leave a trace on the cache (and TLB) regardless of whether it is, say, in-bounds or out-of-bounds. Hence, even if the leak instruction contains a masking operation to keep the index in-bounds—as in Listing 3b—we would still report it as a cache-based gadget, as it leaks at least *some* information. In doing so, we avoid false negatives which would arise from failing to identify gadgets that leave traces in the cache.

Leakage Policy II: Identifying MDS-based gadgets. *To identify MDS-based gadgets, we use the same policies as our access policies (which raise `attacker` data to `secret` data) with one minor difference (similar to our LVI policy): we do not report directly-controllable null-memory accesses because exploitation of such accesses is more difficult.*

For example, consider the MDS-based gadget in Listing 9b, where the errant access loads secret data into internal CPU buffers. Our policy reports an MDS gadget since the memory access outputs secret data, thereby potentially leaking the secret data to an attacker thread sharing an SMT core with the kernel thread. Note that for MDS-based gadgets, the *access* and *leak* steps occur in a single instruction.

Just as we avoid over-tainting harmless (e.g., in-bounds) *access* instructions, we avoid over-reporting MDS-based gadgets which are similarly benign, thereby avoiding a source of false positives.

Leakage Policy III: Identifying port contention-based gadgets. *If a `secret` affects the flow of execution—that is, if a branch's condition, switch's condition, indirect call's target,*

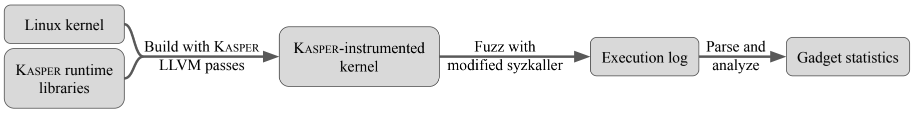

Linux kernel
Kasper runtime libraries
Build with Kasper LLVM passes
Kasper-instrumented kernel
Fuzz with modified syzkaller
Execution log
Parse and analyze
Gadget statistics

Figure 3.4: The workflow of Kasper, which takes a vulnerable Linux kernel, identifies gadgets, and finally presents statistics to aid developers in applying mitigations.

or indirect branch's targets is a `secret`—then we report it as a port contention-based covert channel.

For example, consider the port contention-based gadget in Listing 9c. Our policy identifies the gadget because the secret determines whether the branch is taken—and in effect, determines the resulting port contention, which can leak a bit of the secret to an attacker that shares an SMT core with the kernel.

3.8 Implementation

Figure 3.4 presents the workflow of Kasper, our generalized transient execution gadget scanner for the Linux kernel. First, we build the kernel with Kasper support by using three components that each consist of a runtime library and an LLVM pass[2]: (1) Kernel Speculative Emulation Unit (KSpecEm), which emulates speculative execution due to a branch misprediction, (2) Kernel DataFlow Sanitizer (KDFSan), which performs the taint analysis, and (3) Taint Manager (TManager), which manages the vulnerability-specific taint policies. Next, we use a modified version of syzkaller [52] to fuzz the instrumented kernel and generate gadget reports. Finally, we calculate aggregate gadget statistics that aid developers in applying mitigations. Table 3.3 presents the total lines of code (LOC) in each of the main components of Kasper.

Kernel Speculative Emulation Unit To simulate branch mispredictions, KSpecEm replaces branch conditions with a `SelectInst` that, depending on a runtime variable, will either take the original branch target or the inverse branch target. To simulate the resulting speculative execution, KSpecEm hooks store instructions so that any speculative memory write is reverted after speculative emulation finishes.

Kernel DataFlow Sanitizer For our taint analysis engine, we ported the user-space DataFlowSanitizer (DFSan [113]) from the LLVM compiler to the kernel. KDFSan is, to our knowledge, the first general compiler-based dynamic taint tracking system for the Linux kernel. In contrast to the original DFSan implementation, we: (1) modified its shadow memory implementation to work in the kernel, (2) fixed its flawed label union operation (similar to existing work [26, 166]), (3) modified it to conservatively wash taint on the outputs of inline assembly, and (4) created custom taint wrappers to emulate the semantics of uninstrumentable code.

[2]We opted for an LLVM-based implementation due to its low complexity and better performance compared to e.g., a full-system emulator. Future work may see improvements by complementing Kasper with a binary-based approach since it does not require special care of low-level code.

Component	LLVM pass	Runtime library
KSpecEm	964	2102
KDFSan	251 (diff)	1975
TManager	57	65
syzkaller	–	355 (diff)

Table 3.3: Total lines of code (LOC) in the various components of Kasper. If a component is based on an existing one, the LOC given is the diff with respect to the original.

Taint Manager TManager implements the taint policies described in Section 3.7. It receives callbacks from the kernel and the KDFSan runtime library and will e.g., apply taint to syscall arguments, or determine if a memory operation is an *inject*, *access*, or *leak* instruction. We e.g., apply taint on system call arguments to mark them as attacker controlled by patching the system call routine added the necessary calls to the KDFSan runtime library. It re-uses checks from KernelAddressSanitizer (KASAN) [98] to determine whether a memory operation is an unsafe or an invalid access.

syzkaller We used a customized version of syzkaller [52], an unsupervised coverage-guided fuzzer for the kernel, to maximize speculative code coverage. syzkaller's strategy of maximizing regular code coverage will inevitability maximize speculative code coverage since we start a speculative emulation window at every regularly-executed branch. We utilized qemu snapshots to begin every syzkaller testcase (a series of syscalls) from a fresh snapshot, avoiding leftover taint from previous testcases.

Gadget aggregation After fuzzing, Kasper parses the execution log for gadget reports. Then, it filters out duplicate reports, categorizes them by type, and prioritizes them based on a set of exploitability metrics (see Section 3.C for a description of these metrics). Finally, it stores the resulting aggregate gadget statistics into a database that is used by a web interface to present the results. In Section 3.E, we present the interface that allows kernel developers to easily process the found gadgets.

3.9 Evaluation

We evaluate Kasper's efficacy compared to existing solutions and Kasper's gadgets found in the Linux kernel. We perform our evaluation on an AMD Ryzen 9 3950X CPU with 128GB of RAM, where the Kasper-instrumented kernel (v5.12-rc2) runs as a guest VM on a host running Ubuntu 20.04.2 LTS (kernel v5.8.0).

3.9.1 Comparison with previous solutions

We compare Kasper against previous approaches in a variety of ways. First, as a micro-benchmark, we evaluate Kasper and all other approaches on the Kocher gadget dataset [94]. Then, as a macro-benchmark, we evaluate Kasper and other dynamic approaches on the syscalls invoked by UNIX's `ls` command. Finally, in Section 3.B, we evaluate the performance of Kasper and compare it to existing approaches, where applicable.

		Accesses detected	*Leaks* detected
Static	MSVC [134]	–	7
	RH Scanner [30]	–	12
	oo7 [185]	–	15
	Spectector [60]	–	15
Dynamic	SpecFuzz [126]	15	0[a]
	SpecTaint [140]	15	14[b]
	Kasper (User/Cache-only)	15	14
	Kasper (User/Cache/Port-only)	15	15

[a] SpecFuzz's eval. reports 15 leaks since it assumes all *accesses* are *leaks*.
[b] SpecTaint's eval. reports 15 leaks since it assumes arbitrary ptrs. are secret.

Table 3.4: Gadgets detected in the 15 Spectre Samples Dataset [94] by various solutions.

Micro-benchmark We evaluate Kasper and previous work against Paul Kocher's 15 Spectre examples [94], which were originally designed to evaluate the effectiveness of the Spectre mitigation in MSVC (Microsoft Visual C++). Although the examples are simple—as gadgets in real-world programs are often much more complex—they represent one of the few well-defined microbenchmarks available for direct comparison with different solutions.

Table 3.4 shows the results for tools that are based on static and dynamic analysis. Static analysis tools specifically model the Spectre-PHT gadgets, combining the access to the secret and its leakage through a covert channel. While these tools scale to such small code snippets, they often miss many cases with more complex gadgets combining multiple attack vectors. Furthermore, it is difficult to run solutions based on symbolic execution [60] on large codebases such as the Linux kernel without losing soundness.

In the dynamic analysis category, SpecFuzz [126] depends on address sanitizers to detect invalid accesses. Without data flow analysis, tracking the data dependency between accessing and leaking of a secret is not possible. Assuming the second access encoding the secret in the reload buffer is inbounds, SpecFuzz is not able to detect it. SpecTaint [140] makes use of dynamic taint analysis, allowing it to detect dependencies between the access and leakage. While the authors report that they detect all 15 variants, it is unclear how they detect variant v10, that leaks the secret through the outcome of the branch. In direct communication with the authors, they mentioned that SpecTaint assumes that the leaking pointer is tainted as a secret. However, this makes it independent of the presented variant.

Without implicit flow tracking, Kasper detects the access of the secret for all 15 variants and the transmission of 14 of those. For the undetected variant (v10), Kasper's port contention policies detect the attacker-controlled branch giving the attacker the ability to leak the secret by controlling the outcome of the branch.

Macro-benchmark To provide a more realistic scenario, we evaluate Kasper and previous work by running the `ls` UNIX command. Since SpecFuzz [126] and SpecTaint [140] are only implemented for user-space programs, we adapted Kasper to model their functionality for the kernel. To model SpecFuzz, we report any address sanitizer violation within speculative emulation as a Cache gadget. To model SpecTaint[3], we taint syscall arguments and user copies as `attacker` data, taint the output of any `attacker`-

[3]We base our implementation of SpecTaint on the information provided in the paper, because while the paper states the authors' intention to release SpecTaint as open source, the code is not yet available.

	Total CACHE gadgets reported	FP rate	FN rate (USER-only)	FN rate
SpecFuzz [126]	662	99%	33%	60%
SpecTaint [140]	688	99%	0%	40%
KASPER (USER-only)	8	25%	0%	40%
KASPER	14	29%	0%	0%

Table 3.5: Cache gadgets reported in the kernel by various solutions when running the `ls` UNIX command.

dependent speculative load as `secret` data, and report any `secret`-dependent access as a CACHE gadget.

For each solution, Table 3.5 presents: (1) the total number of CACHE gadgets reported, (2) the false positive rate, (3) the false negative rate when scanning for gadgets controlled only by USER input, and (4) the false negative rate when scanning for gadgets controlled by USER, MASSAGE, and LVI input. In the scenario where an attacker can only *inject* USER input, we define a true positive as a gadget that: (1) *injects* directly-controllable attacker data (i.e., syscall arguments or user copies), (2) *accesses* a secret by dereferencing a pointer that is both unsafe (i.e., out-of-bounds or use-after-free) and directly-controllable (i.e., flowing from the *injected* data), and (3) *leaks* the secret by dereferencing a pointer that is secret (i.e., flowing from the *accessed* data). In the scenario where an attacker can also *inject* MASSAGE and LVI data, we define a true positive as a gadget that may additionally: (4) *inject* indirectly-controllable attacker data by dereferencing a pointer that is either invalid (i.e., a non-canonical address or a user address with SMAP on) or unsafe, and (5) *access* a secret by dereferencing a pointer that is indirectly-controllable.

As shown in the table, existing solutions' pattern-based approaches incur a high FP rate (99%) and a substantial FN rate (up to 33%). In contrast, KASPER (USER-only)'s more principled approach drastically decreases the FP rate (25%) and matches the best case FN rate (0%). However, when considering gadget primitives beyond those in the BCB pattern—i.e., MASSAGE and LVI attacker inputs—FN rates for existing approaches (and for the limited version of KASPER) increase significantly (to 40%–60%). Since the full version of KASPER models these primitives (and more), it has no FNs and maintains an acceptable FP rate (29%). We observed similar results on the GNU core utilities other than `ls`.

We verified our FP/FN rates by checking whether each reported gadget satisfies our definition of a TP. First, we verified that KASPER's FPs are due to overtainting; moreover, these FPs are also nondeterministic (i.e., they only appear in specific runs). Second, we verified that all of SpecFuzz's FPs are due to its inability to track attacker/secret data, causing it to report *leaks* of non-secret data (as in Listing 3c). Third, we verified that all of SpecFuzz's (USER-only) FNs are due to its inability to identify in-bound *leaks* (as in Listing 3b). Fourth, we verified that all of SpecTaint's FPs are due to its inability to filter out safe *accesses* (as in Listing 3d). Finally, we verified that all remaining FNs are due to existing approaches (and the limited version of KASPER) not modeling MASSAGE or LVI attacker input.

From these results, we can conclude that previous approaches are insufficient for a variety of reasons. First, the high FP rates for existing techniques are problematic because hardening every reported gadget would lead to a substantial slowdown, yet attempting to verify all the reported gadgets is impractical. Second, SpecFuzz's substantial FN rate

Gadget type	Number of reports
PHT-USER-CACHE	147
PHT-MASSAGE-CACHE	47
PHT-LVI-CACHE	12
Total PHT-*-CACHE	183
PHT-USER-MDS	600
PHT-MASSAGE-MDS	193
Total PHT-*-MDS	722
PHT-USER-PORT	407
PHT-MASSAGE-PORT	123
Total PHT-*-PORT	474
Total PHT-*-*	1379

Table 3.6: Different gadgets discovered by Kasper.

is problematic because it leaves many unidentified gadgets vulnerable. Finally, the high FN rates when considering MASSAGE and LVI primitives—which would be far worse if also considering MDS and PORT primitives—are problematic because they highlight how previous approaches leave generic gadgets completely vulnerable.

3.9.2 Gadgets found in the kernel

We fuzzed the kernel for 18 days with 16 virtual machines running in parallel and report the number of gadgets found in Table 3.6. KASPER currently taints data copied from user space (USER), data vulnerable to memory massaging (MASSAGE) and data vulnerable to LVI-injection (LVI) as `attacker` input. Furthermore, the prototype assumes that `secret` data can leak either through microarchitectual buffers (MDS), port contention (PORT) or cache-based covert channels (CACHE). Although KASPER can model PHT-LVI-MDS and PHT-LVI-PORT gadgets, we do not report such variants since we were unable to exploit them in practice (see Section 3.4).

Most of the gadgets found by KASPER allow information to leak via MDS after a PHT misprediction. The input for these gadgets mostly comes from syscall arguments or values that the attacker can indirectly control in memory via massaging techniques. We present a case study of one such gadget that is difficult to mitigate in Section 3.11. KASPER also finds 147 gadgets with the same capabilities as Spectre-BCB, which are still missed by existing mitigations due to limitations in the kernel's static analysis-based gadget scanning tools, which only look for specific patterns.

3.10 Limitations

Although we have shown that KASPER outperforms state-of-the-art gadget scanners, it is still limited relative to the *ideal* gadget scanner—i.e., a scanner that detects *practically exploitable* gadgets with *perfect precision*.

Practical exploitability We do not evaluate the practical exploitability of every gadget found. Such an evaluation is infeasible in bounded time: it would require testing each gadget against a myriad of possible microarchitectures, attack patterns to facilitate the exploit (e.g., concurrent cache evictions [50]), etc. Instead, like existing kernel Spectre mitigations and state-of-the-art gadget scanners [126, 140], we aim to provide a more comprehensive security-by-design, since even gadgets that are seemingly nonexploitable now may become practically exploitable due to seemingly-innocuous changes—e.g., microcode updates, code refactors, or different compiler versions [93]. Nonetheless, in Section 3.11, we present a proof-of-concept exploit of a gadget found by KASPER. Furthermore, in Section 3.C, we discuss heuristics that developers can use to prioritize gadgets that are more likely to be exploitable in practice.

Completeness KASPER's results may be incomplete for a couple of reasons. First, similar to existing dynamic gadget scanners [126, 140], we inherit dynamic analysis's inherent limitation of incomplete coverage. In Section 3.A, we evaluate this limitation and conclude that as fuzzing progresses, FNs due to incomplete coverage become more and more rare. Second, since (K)DFSan does not track attacker-controllable implicit data flows, KASPER will fail to identify gadgets that rely on them. However, these FNs may be less useful to an attacker, because implicit flows normally propagate only one bit of data.

Soundness Similar to existing scanners based on dynamic taint analysis (DTA) [140], we inherit DTA's inherent soundness limitations. Specifically, since DTA cannot model data-flow constraints (e.g., the effect of arbitrary arithmetic or bitwise operations on data), KASPER may report gadgets that are not entirely controllable by an attacker. For example, even though it might only be possible for an `attacker`-controllable *access* to go one byte out-of-bounds—rather than *arbitrarily* out-of-bounds—KASPER will overlook this, and still taint the output as a `secret`. To mitigate these FPs, KASPER washes taint for common data masking operations that are part of mitigations in the Linux kernel (i.e., `array_index_nospec`), but a general solution remains out of the scope of DTA. A generic way to address this problem is to rely on symbolic (or concolic) execution, but state-of-the-art techniques can only scale to a limited number of basic blocks of kernel execution [206]. In contrast, KASPER can scalably find gadgets with attacker-controlled data propagating even across multiple syscalls.

Fidelity Since our implementation does not faithfully model every aspect of a natively-run kernel, its results may be imprecise. First, we cannot precisely model nondeterminism—e.g., from timer interrupts occurring at different program states in the KASPER kernel compared to the native kernel. We can mitigate any resulting FNs by increasing coverage. Second, we do not precisely model all microarchitectural details—e.g., exact speculative window sizes, pipeline stalls caused by memory aliasing, etc. We can mitigate any resulting FNs by extending KASPER to model even more low-level microarchitectural details. We consider any fidelity-related FPs to be acceptable, as described above (i.e., seemingly-innocuous changes may turn a FP into a TP).

3.11 Case Study

We have shown that KASPER finds a wide range of gadgets in the hardened Linux kernel codebase. To demonstrate that the presence of such gadgets is serious, we now present a case study for one of the (unmitigated) gadgets found by KASPER. It shows that focusing

```
#define list_for_each_entry(pos, head, member)
  for (pos = list_first_entry(head,
         typeof(*pos), member);
     &pos->member != (head);
     pos = list_next_entry(pos, member))
```

Listing 10: Linux implementation for generic list iterations. `pos` is the current list element, `head` is the head of the list, and `pos->member` is the next list element.

on simple gadgets is insufficient and mitigation can be far from trivial. Finally, we analyze the exploitability of the found gadget. We refer the interested reader to Section 3.D for an additional case study illustrating the need for a dynamic analysis tool for the Linux kernel.

3.11.1 List implementation of the kernel

Our case study is fundamental to the (double linked) list implementation that is used pervasively in the Linux kernel. The kernel's list implementation is cyclic and consists of a head element (of type `list_head`) which is typically a field in another data structure, and list elements containing the data, where the last element points back to the head element. A data structure can become a list element, if it also embeds the `list_head` struct as one of its fields. The `list_head` simply contains pointers to the `list_head` fields in the previous and next list elements (or the head).

To iterate over a list, the kernel provides macros such as `list_for_each_entry`, shown in Listing 10. Here `pos` points to the data structure and `pos->member` to the embedded `list_head`. The list iteration terminates when it reaches the head element in the cyclic list. List iterators are also used pervasively in the kernel codebase with more than 2600 uses.

3.11.2 A `list_for_each_entry` gadget in `keyring.c`

The security implications of the `list_for_each_entry` implementation become clear when we consider the gadget found by Kasper in the `find_keyring_by_name` function in `keyring.c`. Listing 11 shows the relevant code snippet. Simulating a branch misprediction, Kasper flips the terminating condition of the list iterator (`&pos->member != head`), resulting in an additional iteration with `&pos->member == head`. Note that when the list iteration is speculatively executed for the `head` element, there is no associated `key` data structure. In other words, `keyring` and `&keyring->user` point to out-of-bounds memory, in this case belonging to the data structure containing the head element. The type confusion results in the access to `keyring->user->uid` dereferencing an out-of-bounds read pointer.

Kasper not only detects the KASAN violation, but further reports the gadget as capable of leaking secrets trough MDS, as it verifies through its dynamic taint analysis that the pointer is attacker controlled.

```
struct key *find_keyring_by_name(const char *name, bool uid_keyring) {
  struct user_namespace *ns = current_user_ns();
  struct key *keyring;
  ...
  list_for_each_entry( keyring , &ns->keyring_name_list, name_link) {
    if (!kuid_has_mapping(ns,  keyring->user->uid ))
      continue;
  ...
} }
```

Listing 11: PHT-Massage-MDS gadget in `find_keyring_by_name`.

3.11.3 Exploitation

First, we evaluate the controllability of the out-of-bounds read pointer. The type-confusion assumes that `&ns->keyring_name_list` is within a `key` struct, but in reality the head element is in a `user_namespace` struct. To compute the location of `&keyring->user` in the speculative iteration, we first compute the offset of `name_link` within the `key` struct. Next we compute the offset of `user` in the struct and apply those offsets to `&ns->keyring_name_list` as shown in Figure 3.5. The out-of-bounds read pointer `keyring->user` is read from `ns->projid_map` which can be easily controlled from user space through the proc interface.

We evaluated the gadget on an i7-7700K machine with a recent v5.12 Linux kernel. First, we verified whether the branch can be mistrained. Since the attacker can control the length of the `keyring_name_list` list by adding keys through the `add_key`, mistraining the branch condition in-place is not difficult. Moreover, for easier exploitation, the gadget can be reached close to the system call entry of `keyctl` with the `KEYCTL_JOIN_SESSION_KEYRING` operation.

We build a simple Flush+Reload [195] proof of concept to verify that the attacker-controlled pointer is actually used in the speculative load operation. For verification purposes, we disabled SMAP and set the pointer pointing directly into the reload buffer and observed the signal by executing the `keyctl` system call between flushing and reloading. We confirm that the attacker-controlled pointer is dereferenced during speculative execution.

In principle, any value loaded by a speculative load can be leaked cross-thread with MDS [22, 150, 181] if SMT is enabled. Since SMT is not disabled by default in the latest version of Linux, it is still vulnerable to MDS when leaking from the sibling hyperthread. We verify this with a simple proof of concept where one thread repeatedly executes the `keyctl` system call and the other thread reads in-flight data and encodes it within a Flush+Reload buffer. We use `madvice` to force a page fault across a page boundary which is required to leak the in-flight data. Listing 12 presents a simplified version of the proof of concept. Without synchronization between the two threads (which was not the focus of our research), the signal strength depends on the leaking load in the system call occurring roughly at the same time as the load from CPU-internal buffers in the reading thread. We verified that a signal exists if the loads are happening approximately at the same time in both threads, leaking the secret from a kernel buffer to user space.

Mitigating the presented gadget is far from trivial. Adding a spot mitigation in `keyring.c` leaves all other uses of `list_for_each_entry` vulnerable. Mitigating the

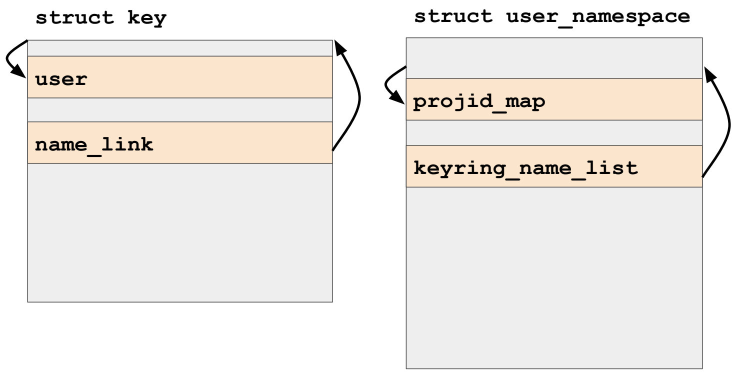


Figure 3.5: Through the type confusion `head` is assumed to be within a `key` struct instead of the `user_namespace` struct.

list iterators, that are frequently used throughout the kernel, may cripple performance when using `lfence` in every iteration of the loop. Other kernel mitigations such as `array_index_nospec` are not applicable with the current list implementation.

3.12 Related Work

Checkpointing for transactional execution While checkpointing was used to recover from software faults [107, 182], we require such high frequency checkpointing (on every conditional branch) that we built a checkpointing mechanism for the Linux kernel from scratch. However, the solution is general and applicable to other scenarios (e.g., crash recovery).

Taint tracking in the kernel For KASPER, we built KDFSAN, the first generic compiler-based dynamic taint tracking system for the kernel. Other non-generic, non-compiler-based, and static taint tracking systems have also found bugs in the kernel. For example, KMSAN is a widely-used compiler-based dynamic taint tracking system targeted at detecting uninitialized memory usage; it has found hundreds of bugs in the Linux kernel [139]. Furthermore, Bochspwn Reloaded is an emulation-based dynamic taint tracking system targeted at detecting uninitialized memory disclosures to user-mode; it has found 70 bugs in the Windows kernel and 10 bugs in the Linux kernel [86]. Finally, a wide range of static taint tracking systems have found numerous bugs in the kernel [85, 117, 187].

Speculative gadget scanners Shortly after the publication of Spectre [95], the Red Hat Spectre V1 scanner [30] and Microsoft's Visual C/C++ Compiler [134] tried finding and mitigating Spectre V1 gadgets using well-defined code patterns. More advanced static analysis gadget scanners followed. oo7 [185] uses binary-level static taint analysis and has analysis times of 74 hours on average on medium-sized user applications, scaling it to large codebases such as the Linux kernel is impractical. SPECTECTOR [60] and KLEESPECTRE [184] use symbolic execution to find Spectre based information leaks.

```
while(1)
  syscall(__NR_keyctl,
      KEYCTL_JOIN_SESSION_KEYRING,
      "kasper", 0, 0, 0);
```

(a) Attacker thread that repeatedly invokes the vulnerable `keyctl` system call.

```
while(1) {
  madvise(leak+4096, 4096, MADV_DONTNEED);
  flush(reloadbuffer);
  asm volatile(
    "movdqu (%0), %%xmm0    \n"
    "movq %%xmm0, %%rax     \n"
    "andq $0xff, %%rax      \n"
    "shl $0xa, %%rax        \n"
    "prefetcht0 (%%rax, %1) \n"
    "mfence                 \n" ::
    "r"(leak + 4096 - 14),
    "r"(reloadbuffer):"rax","rbx","rcx");
  reload(reloadbuffer, results);
}
```

(b) Attacker thread that repeatedly encodes in-flight secret data into a `reloadbuffer`.

Listing 12: Simplified proof of concept exploit consisting of two simultaneously executing threads.

Without sacrificing soundness and completeness, symbolic execution becomes a bottleneck when scaling to large real-world codebases like the kernel [206]. The Linux kernel currently relies on manual analysis and static analysis tools such as smatch [25] to identify potential Spectre gadgets. However, this still involves manual inspection because found code locations are patched at the source code level and it suffers from a high rate of false positives. SpecFuzz [126], a dynamic analysis tool, combines fuzzing with the use of sanitizers to detect and mitigate potential speculative memory leaks within user applications. SpecTaint [140] uses dynamic taint analysis to track attacker controllability and leaking of the secret. Similar to SpecFuzz, it targets user-space programs which rarely have mitigations applied.

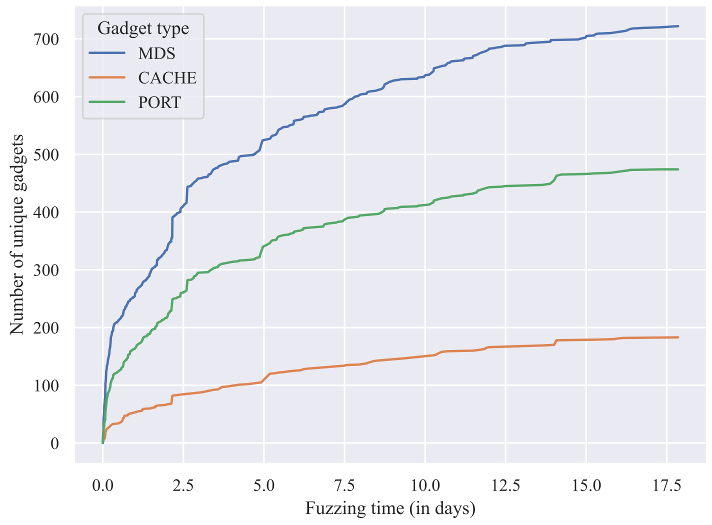


Figure 3.6: Gadgets found by Kasper over time, separated by gadget type.

3.13 Conclusion

We presented KASPER, a solution for finding generalized transient execution gadgets in the Linux kernel. Where existing gadget scanners limit themselves to specific Spectre patterns (in user programs), KASPER abstracts away pattern-specific details and instead models the essential steps of an attack: injecting controllable data, accessing a secret, and then leaking the secret. Moreover, where existing scanners target only the primitives used in the BCB pattern, KASPER models the wide variety of primitives that are at an attacker's disposal. As a result, KASPER finds gadgets that are well out of reach of existing techniques. We conclude that current Spectre mitigations in the Linux kernel are wholly insufficient.

Appendix 3.A Coverage Evaluation

We evaluate the completeness of KASPER's fuzzing results (Section 3.9.2) based on its coverage in both normal execution and speculative emulation.

3.A.1 Normal execution coverage

First, we find that KASPER's total gadgets found (Figure 3.6) begins to flatten at around 18 days; this is not surprising, since we fuzzed the kernel until KASPER found only 3 new gadgets per day (out of 1379 gadgets found in total). Next, we find that KASPER's code coverage (Figure 3.7) also flattens, confirming that further fuzzing will most likely execute already-executed code, and as a result, uncover fewer and fewer gadgets.

To evaluate how long it would take for KASPER's code coverage to *completely* flatten (and in effect, uncover almost *all* gadgets), we compare KASPER's code coverage to an

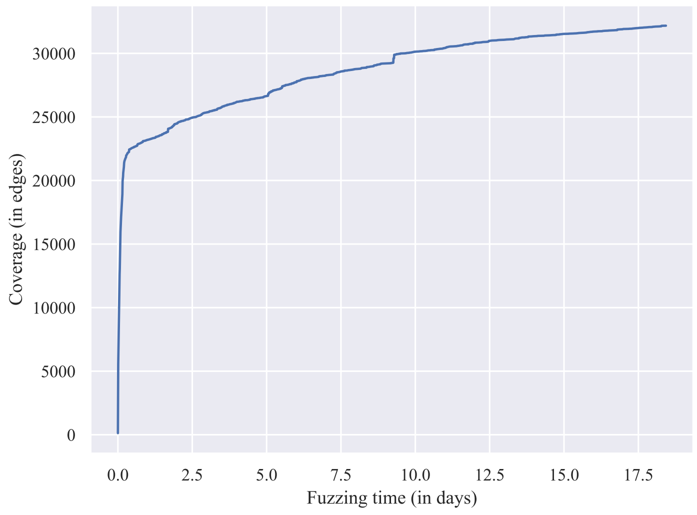


Figure 3.7: Covered edges in the Linux kernel reported by syzkaller.

uninstrumented kernel's code coverage. We fuzzed an uninstrumented kernel for 18 days and found that in the final 24 hours, the baseline's coverage increased by only 0.15%, and in the same span, KASPER's coverage similarly increased by only 0.61%. In other words, the baseline's coverage—like KASPER's coverage—flattens, but not completely. Since even the baseline does not completely flatten in bounded time[4], we cannot expect KASPER to completely flatten in bounded time. Hence, future work on improving syzkaller's coverage would not only improve the completeness of KASPER's results—it would also improve *all* kernel fuzzing results.

3.A.2 Speculative emulation coverage

Apart from *normal* code coverage, KSPECEM specifically targets increasing the code coverage in the *speculative window*. Simply executing basic blocks within speculative emulation is not sufficient. Ideally, KSPECEM should execute a basic block speculatively in every speculative window that may cover it. In other words, we should exhaust speculative execution windows as much as possible and eliminate premature rollbacks.

Lightweight checkpointing in the kernel is more challenging than in well-defined user-space programs, due to the many complex operations, frequent use of inline assembly, interactions with device memory, and complex control flow resulting from interrupts. As described in Section 3.6.2, to handle such cases we allow speculative emulation to stop gracefully to ensure memory integrity on a rollback. Stopping speculative emulation in the above cases limits the length of the emulation window.

Table 3.7 presents the different causes of rollbacks in three scenarios: the basic implementation of KSPECEM and two improvements that we later made to improve speculative code coverage. These numbers report the causes for rollbacks averaged over

[4]Indeed, Google runs syzkaller continuously on many different kernel versions and configurations, and even its longest-running fuzzing campaigns continue to gain code coverage [51].

Rollback cause	Basic	+ Page fault suppression	+ Inline assembly patches
Max spec length	20.8%	22.9%	56.9%
Return	32.9%	33.8%	39.7%
Inline asm	38.4%	41%	0.2%
Indirect calls	0.8%	1.2%	1.3%
Interrupts	6.5%	0.5%	1.2%
Blacklisted function	0.6%	0.6%	0.7%

Table 3.7: Causes for rollback in speculative emulation.

all instrumented functions when executing the `ls` command. A negligible amount of rollbacks (3.4%) are due to limitations introduced by unique challenges of the Linux kernel, as explained earlier in Section 3.6.2. We set the maximum of speculative instructions for all evaluations to 250 executed LLVM IR instructions, which is in line with the size of the ROB in modern microarchitectures [74] and what has been used in recent work (for x86 instructions) [126, 140]. We have analyzed the ratio of the number of LLVM IR instructions to x86 instructions within the Linux kernel per function using LLVM passes. The median of the ratio is 0.92, concluding that counting LLVM IR instructions is a reasonable approximation for the number of machine code instructions. The first row shows the percentage of cases where KSpecEm rolls back because we reach this maximum speculation length. Ideally, the percentage of rollbacks due to reaching the maximum speculation length should be as high as possible.

We see that a small percentage of rollbacks can be attributed to indirect calls where the target address cannot be verified at compile time, interrupts, and blacklisted functions such as those that interact with I/O memory. This proves that, even for drivers code, we have isolated the small locations interacting with I/O mapped memory, still reaching high speculative code coverage within those kernel components. The low number of rollbacks due to interrupts show that these interrupts, caused by exceptions creating an unrecoverable state, are not a significant limitation of Kasper. A major source of rollbacks consists of returns as speculative emulation tries to go up in the call trace from the start of the checkpoint. In contrast, returning in functions that have been called within speculative emulation is safe since it will return back into a function that was already executed within emulation. Future work can support additional returns by keeping track of safe functions within the call trace, and thus increase speculative code coverage even further.

In the basic implementation (column 2), we stop on all interrupts (including exceptions and timers) and every occurrence of inline assembly. In this case, speculative emulation reaches the maximum speculation length in a modest 20.8% of all cases. However, by adding Page Fault suppression (column 3), we find new classes of attack (e.g., LVI) and reduce interrupts from 6.5% to a mere 0.5%, to arrive at 22.9% of the cases that exhaust the maximum speculation window. Moreover, by modeling the most frequently-used inline assembly fragments, we increase the percentage of rollbacks due to reaching the speculation length to 56.9%. As shown by our case studies, these improvements enabled Kasper to find a wide range of gadgets.

Appendix 3.B Performance Evaluation

We evaluate the performance of KASPER relative to an uninstrumented kernel and where possible, relative to previous approaches.

Analysis time Similar to previous work [126], we evaluate the time overhead of our approach based on the fuzzing throughput (i.e., the number of testcases over time). First, we compare the fuzzing throughput of KASPER against the uninstrumented kernel. Next—since we observed that our modifications to syzkaller, which use qemu's snapshot feature to revert taint between testcases (see Section 3.8), introduced a major overhead—we also compare the fuzzing throughput of KASPER against the uninstrumented kernel running with our modified version of syzkaller.

We ran each setup for 36 hours on the same machine used for our fuzzing evaluation (Section 3.9.2). We found that on average, KASPER executes 136 testcases per hour, compared to the uninstrumented kernel executing 45,933 per hour; however, when running with our modified version of syzkaller, the uninstrumented kernel executes a mere 252 testcases per hour. Hence, we can attribute a *1.8x slowdown due to KASPER's instrumentation* and a *183x slowdown due to our syzkaller modifications*. Since our modifications to syzkaller were not the focus of this work, we consider this an acceptable overhead; orthogonal (and concurrent) efforts to optimize qemu's snapshot feature can be used to improve the throughput [148]. Furthermore, note that reverting taint between testcases is not strictly necessary for KASPER. Nonetheless, we opted for this strategy because we prefer to have reproducible results (by starting each testcase from a clean snapshot) over quick results (by fuzzing without snapshotting).

For comparison, SpecFuzz reports a *23x slowdown in the best case* (on libHTP) and a *560x slowdown in the worst case* (on OpenSSL). Hence, relative to the 1.8x overhead of KASPER's instrumentation, SpecFuzz's instrumentation incurs a sizable overhead. We attribute this difference to SpecFuzz's use of nested speculation to improve speculative code coverage: i.e., SpecFuzz inverts many conditional branches within a single speculative emulation window, whereas KASPER only inverts one. Although SpecFuzz observes that most gadgets found are within the lower orders of nested speculation (i.e., only requiring one or two inverted branches), nested speculation (as well as other forms of speculation) can be integrated in KASPER in future work.

Unfortunately, we cannot meaningfully compare against SpecTaint for a couple of reasons. First, its performance numbers are not relative to a tangible baseline: e.g., it does not present baseline times for programs when *not* run with SpecTaint; also, it does not define when an analysis is "finished", even though the analysis times it presents use this metric. Second, the code is not yet available, so we cannot reproduce its performance numbers. However, we estimate that, because it uses a full-system emulation-based approach, it likely incurs a more significant overhead compared to KASPER's and SpecFuzz's LLVM-based approaches.

Memory consumption KASPER consumes just over 4x memory relative to a baseline system. Of this overhead, 2x is required by syzkaller for its snapshot feature. Another 2x is required by KDFSAN's shadow memory, which allocates a page of shadow memory for every page of kernel memory. Beyond that, a constant, negligible amount of memory is required for KSPECEM's tracking of speculative memory writes and other internal data structures. Unfortunately, neither SpecFuzz nor SpecTaint evaluate memory consumption, so we cannot compare against them.

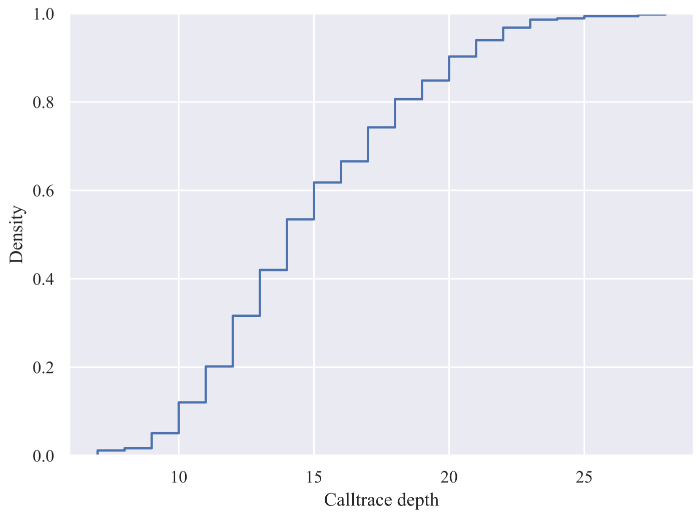


Figure 3.8: Distribution of the smallest calltrace for the found gadgets.

Appendix 3.C Large-Scale Exploitability Evaluation

We evaluate the found gadgets across different metrics to provide a more detailed analysis of exploitability. Exploitability of a gadget depends on e.g., how close it is to the syscall entry to avoid unrelated noise or how long is the speculative window until the gadget is reached.

Required call depth We present the distance to the syscall entry by the depth of the calltrace in Figure 3.8. The first four to five functions called on a syscall entry are usually small and just direct execution towards the correct syscall handler, explaining the low number of unique gadgets found with a call depth below five. 50% of the gadgets are found with a call depth of a maximum of 9, suggesting that they are more likely to be exploitable by this metric. There was no significant difference in the call depth between the different gadget types.

Required speculation window length In Figure 3.9, we present the length of the speculative window from the mispredicted branch until the leakage. More than 60% of MDS gadgets have a speculative window length of less than 100 LLVM instructions. Similarly, 50% of the Cache and Port gadgets have a speculative window length of less than 130 LLVM instructions. MDS gadgets, on average, require a smaller window since these gadgets leak through the access and do not require an additional step to encode the secret in the cache. Such short speculative windows allow for easier exploitation and make the gadgets also applicable to processors with smaller speculative windows.

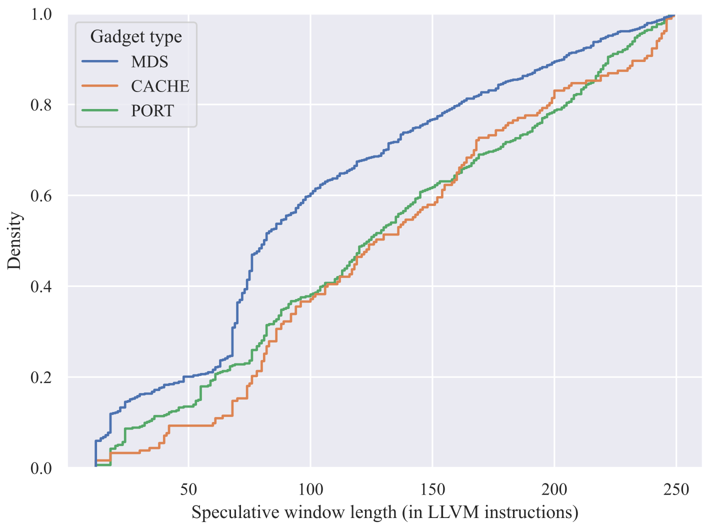


Figure 3.9: Distribution of the smallest speculative length in LLVM instructions for the found gadgets.

Appendix 3.D Additional Case Study

We present a case study found by KASPER that highlights the need for a dynamic analysis-based approach and the need for automated transient execution patch verification.

The gadget Listing 13 shows an MDS-based gadget in the `vc_allocate` function, which is called by `vt_ioctl`. In this gadget, the attacker first supplies `arg` through an ioctl syscall argument, which KASPER marks as tainted. Then, KASPER emulates the mispredicted bounds check at line 1 and executes the *else* statement with an out-of-bounds value for `arg`. After a second mispredicted bounds check at line 8, KASPER detects an out-of-bounds memory access at line 11. Since `curcons` is a 32-bit value under full control of the attacker, this gadget allows an attacker to leak a large range of kernel memory. Note that a previous version of KASPER that used a basic implementation of nested speculation found this gadget, as it relies on two inverted branches (lines 1 and 8).

Drawback of static analysis We identified an almost identical code snippet in `vc_setallocate`—which is very close to the other gadget, and also called from `vt_ioctl`—however, it was already (partially) mitigated through speculative array index masking. It is unclear why the kernel developers applied the mitigation to this gadget, but not to the other. At first glance, the only noticeable difference between the two gadgets is that this gadget receives user data from a `copy_from_user` call, whereas the other gadget receives user data from a syscall argument. However, upon closer inspection, it becomes clear why the kernel's static analysis tool [25] may not have identified it. The dataflow from the `copy_from_user` to the (partially) mitigated gadget is only separated by a *direct* call,

```
 1 if ( arg == 0 || arg > MAX_NR_CONSOLES )
 2   ret = -ENXIO;
 3 else {
 4   arg--;
 5   currcons = arg ;
 6   console_lock();
 7   WARN_CONSOLE_UNLOCKED();
 8   if ( currcons >= MAX_NR_CONSOLES )
 9       return -ENXIO;
10
11   if ( vc_cons[ currcons ].d )
12      ...
13 }
```

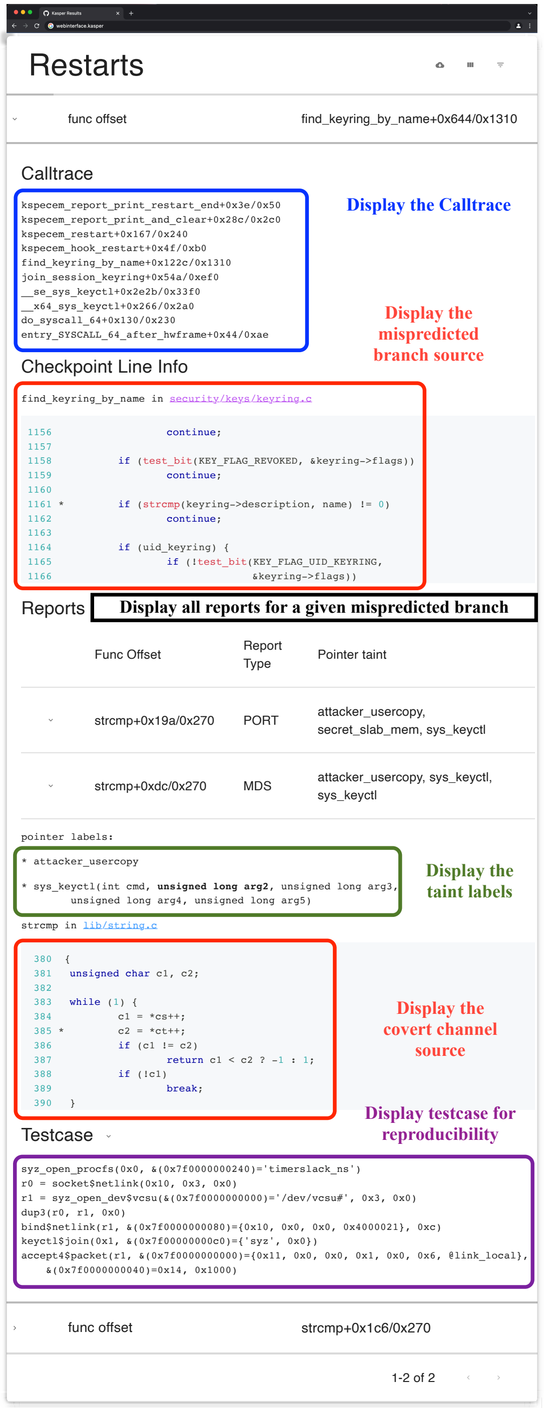
Listing 13: MDS-based gadget in the `vc_allocate` function used within the `vt_ioctl` function.

whereas the dataflow from the syscall argument to the unmitigated gadget is separated by an *indirect* call. Since indirect calls are notoriously difficult to resolve statically [116], it is no surprise that the gadget was left unmitigated. This highlights the importance of a dynamic analysis-based approach, which can uncover gadgets beyond the reach of static analysis.

Drawback of manual mitigation verification Upon further inspection of the mitigation in `vc_setallocate`, we found that it was applied incorrectly. Indeed, KASPER verifies that the mitigation is only partial. Namely, while the speculative array index masking ensures that the index becomes zero if it goes out-of-bounds, the decrement that follows (similar to line 4 in Listing 13) causes an integer underflow in transient execution. This demonstrates the importance of an automated tool such as KASPER, which can verify that manually-applied security patches work as intended.

Appendix 3.E Developer Interface

We have built a web interface to visualize all the necessary information retrieved from the database, so that kernel developers can easily analyze and patch the reported KASPER gadgets. Figure 3.10 presents a screenshot of the interface giving the kernel developer important information such as the calltrace, source code of the mispredicted branch and the leaking operation, taint information, and a testcase to reproduce the report.


Kasper Results
webinterface.kasper
Display the Calltrace
Display the mispredicted branch source
Display all reports for a given mispredicted branch
Display the taint labels
Display the covert channel source
Display testcase for reproducibility

Restarts

func offset find_keyring_by_name+0x644/0x1310

Calltrace

```
kspecem_report_print_restart_end+0x3e/0x50
kspecem_report_print_and_clear+0x28c/0x2c0
kspecem_restart+0x167/0x240
kspecem_hook_restart+0x4f/0xb0
find_keyring_by_name+0x122c/0x1310
join_session_keyring+0x54a/0xef0
__se_sys_keyctl+0x2e2b/0x33f0
__x64_sys_keyctl+0x266/0x2a0
do_syscall_64+0x130/0x230
entry_SYSCALL_64_after_hwframe+0x44/0xae
```

Checkpoint Line Info

find_keyring_by_name in security/keys/keyring.c

```
                        continue;

                if (test_bit(KEY_FLAG_REVOKED, &keyring->flags))
                        continue;

*               if (strcmp(keyring->description, name) != 0)
                        continue;

                if (uid_keyring) {
                        if (!test_bit(KEY_FLAG_UID_KEYRING,
                                      &keyring->flags))
```

Reports

	Func Offset	Report Type	Pointer taint
˅	strcmp+0x19a/0x270	PORT	attacker_usercopy, secret_slab_mem, sys_keyctl
˅	strcmp+0xdc/0x270	MDS	attacker_usercopy, sys_keyctl, sys_keyctl

```
pointer labels:
* attacker_usercopy

* sys_keyctl(int cmd, unsigned long arg2, unsigned long arg3,
        unsigned long arg4, unsigned long arg5)
strcmp in lib/string.c
```

```
{
    unsigned char c1, c2;

    while (1) {
            c1 = *cs++;
*           c2 = *ct++;
            if (c1 != c2)
                    return c1 < c2 ? -1 : 1;
            if (!c1)
                    break;
    }
```

Testcase

```
syz_open_procfs(0x0, &(0x7f0000000240)='timerslack_ns')
r0 = socket$netlink(0x10, 0x3, 0x0)
r1 = syz_open_dev$vcsu(&(0x7f0000000000)='/dev/vcsu#', 0x3, 0x0)
dup3(r0, r1, 0x0)
bind$netlink(r1, &(0x7f0000000080)={0x10, 0x0, 0x0, 0x4000021}, 0xc)
keyctl$join(0x1, &(0x7f00000000c0)={'syz', 0x0})
accept4$packet(r1, &(0x7f0000000000)={0x11, 0x0, 0x0, 0x1, 0x0, 0x6, @link_local},
    &(0x7f0000000040)=0x14, 0x1000)
```

func offset strcmp+0x1c6/0x270

1-2 of 2

Figure 3.10: Screenshot of Kasper's web interface.

4 uncontained: Uncovering Container Confusion in the Linux Kernel

Type confusion bugs are a common source of security problems whenever software makes use of type hierarchies, as an inadvertent downcast to an incompatible type is hard to detect at compile time and easily leads to memory corruption at runtime. Where existing research mostly studies type confusion in the context of object-oriented languages such as C++, we analyze how similar bugs affect complex C projects such as the Linux kernel. In particular, structure embedding emulates type inheritance between typed structures. Downcasting in such cases consists of determining the containing structure from the embedded one, and, like its C++ counterpart, may well lead to *bad casting* to an incompatible type.

In this chapter, we present UNCONTAINED, a systematic, two-pronged solution to discover type confusion vulnerabilities resulting from incorrect downcasting on structure embeddings—which we call *container confusion*. First, we design a novel sanitizer to dynamically detect such issues and evaluate it on the Linux kernel, where we find as many as 11 container confusion bugs. Using the patterns in the bugs detected by the sanitizer, we then develop a static analyzer to find similar bugs in code that dynamic analysis fails to reach and detect another 78 bugs. We reported and proposed patches for all the bugs (with 102 patches already merged and 6 CVEs assigned), cooperating with the Linux kernel maintainers towards safer design choices for container manipulation.

4.1 Introduction

Complex software often makes use of class and type hierarchies to achieve modularity in the design and favor code reuse for operations meant to work on similar objects. Interestingly, this phenomenon is not exclusive to software written in object-oriented languages. One compelling case involves the C language, as implementers of kernels and large userland applications commonly resort to custom means, namely *structure embedding*, to model inheritance between typed structures. In the lack of explicit language provisions, the validity of casting operations becomes an implicit assumption from code semantics (i.e., on implementation correctness).

Structure embedding operates by declaring an instance of a more general typed structure (the parent) as a field of a more specific one (the child). A well-known example is the `list_head` structure in the Linux kernel. In this chapter, we will sometimes refer to such structures as *objects*. Code that needs to access the more general representation of an object, thus realizing an *upcast*, will simply use the member field for the parent in the object. This operation is intuitively safe. Code that needs to access a more specialized representation of an object, thus realizing a *downcast*, will (unsafely) manipulate the parent pointer to recover the address of the child.

In more detail, an object downcast subtracts the offset of the parent field in the child object from the address available for the parent, yielding the address of its *container* structure (i.e., the child). The term container follows from the popular `container_of` macro pioneered by the Linux kernel. Issuing a downcast is not only always unsafe, but even not conforming to any C language standard [127]. Thus, the correctness and safety burden is on the shoulder of the developers, who have to guarantee through program semantics that the requested child type is correct. Failing to meet this requirement would cause a type confusion, which may have possibly disastrous consequences, such as a memory corruption vulnerability [106].

For object-oriented languages, runtime type information (RTTI) enables straightforward validation of downcasting operations. For example, current solutions that look for type confusion in C++ code rely on forms of RTTI tracking [6, 39, 62, 88]. Solutions with provisions for C code can detect (some) cases of type confusion by intercepting heap allocations of objects and binding them with their top-level allocation type [39, 88] in userland code. Automatic type identification is difficult in C programs due explicit/implicit unions, pointer casting, allocation wrappers, and other factors as shown in previous work [48, 178]. For kernels, current type-based solutions resort to manually annotating allocation sites with the necessary type information [6].

In this chapter, we take a systematic approach to discover type confusion vulnerabilities resulting from incorrect downcasting on structure embeddings, which we call *container confusion*. We design a new sanitizer that does away with runtime type tracking of objects and uses instead information on object allocation boundaries, which we obtain using an off-the-shelf solution. In more detail, we rely on redzones from memory sanitization literature [154] to augment allocation sites for out-of-bound access detection. Our sanitizer checks type compatibility for a downcasting operation by checking the relative position of the embedded parent structure, the outer child structure, and the redzones. This scheme transforms a type check in multiple straightforward structure bound checks, with low runtime overhead and no manual code changes.

We apply our sanitizer to the Linux kernel, one of the most complex and security-sensitive program instances. An initial study of its code base, which we conducted to gauge the potential bug surface, reveals more than $50,000$ occurrences of `container_of` involving nearly $4,000$ structure types. The type graph is also highly connected, with extreme cases such as `list_head` used as parent for over 1, 800 child types.

We fuzzed a sanitized build of the kernel for one week and uncovered 11 cases of container confusion, including long-standing container confusion bugs present in its code base since 18 years. As the kernel is continuously fuzzed under multiple sanitizers and configurations, these findings lead us to argue that our approach can find bugs that current state-of-the-art testing practices fail to capture.

By analyzing the nature of such bugs, we identify five container confusion patterns of general interest. We use such patterns to develop a static code analyzer that can process the whole kernel in only a few seconds, allowing us to reach also code compartments that fuzzers may not cover. The static analyzer identifies 366 potential cases of confusion: by manual analysis, we identify 78 other bugs along with 179 anti-patterns where code correctness hinges only on implicit assumptions on program semantics.

We reported our findings to the Linux kernel maintainers, who acknowledged them, and proposed patches for all the bugs we found. At the time of writing, 102 patches have been merged in the kernel, and 6 CVE identifiers have been assigned for bugs whose security implications were immediately apparent. Our reports sparked valuable discussions which, among others, resulted in upgrading the C standard (to mitigate recurrent issues that we found) and in an attempt to change the list iterator integral to the kernel.

In sum, this chapter proposes the following contributions:

- We systematize a class of type confusion bugs, showing how C programs are affected by incorrect downcasting on structure embeddings. We dub it *container confusion*.
- We design a sanitizer for them that does away with type tracking and show its applicability to the Linux kernel.
- We derive 5 general patterns of container confusion from bugs we found in the kernel and design a static analyzer around them to make our approach scale in coverage.
- We evaluate our approach on a recent Linux kernel version, identifying 11 bugs with dynamic analysis (e.g., fuzzing) and another 78 bugs through our static analyzer.

Our sanitizer and static analyzer together form a framework, termed UNCONTAINED, which is open source and available at: `https://vusec.net/projects/uncontained`.

4.2 Background

In this section, we will provide the relevant background to understand the remainder of the chapter.

4.2.1 Type Confusion Bugs in C++... *and in C*

Casting an object to an incompatible type violating casting rules (i.e., bad-casting) causes *type confusion*. For instance, a static downcast in C++ checks only if the source and destination types are in the same type hierarchy, but not if the runtime destination type is the expected one. As a result, large C++ projects, such as the major browsers, parts of Windows, and the Oracle JVM [62], are rife with type confusion bugs.

Downcasting in C. The problem is not limited to object-oriented languages such as C++ but also extends to large programs written in C. Since C is not an object-oriented programming language, it does not support classes like C++. However, developers use *structure embedding* to benefit from an approximation of classes and inheritance. In particular, properties shared by multiple types are defined as a struct *embedded* in all the relevant types. In such a way, all the child types inherit the struct members declared in the parent type that is embedded. We show a simplified example of such use in Listing 14. Since the child type includes the parent type in this design, it is called a *container*.

Analogous to C++, we require primitives to go from the child type to its parent ("upcasting") and from the parent to its child type ("downcasting"). Upcasting is implemented by obtaining a pointer to the embedded parent structure from the child structure and is guaranteed safe. Downcasting is not defined in the C standard since it would require using a pointer to the parent structure to obtain a pointer outside of the memory defined by the type of the parent structure itself [127]. Still, many projects, including the Linux kernel, do exactly that. Given a pointer to the parent in a type hierarchy based on structure embedding, they implement their own version of downcasting, often in the form of a macro, that uses pointer arithmetic to calculate a pointer to the child type.

Such a macro is often named `container_of`. The reference implementation in the Linux kernel is shown in Listing 15. The `container_of` macro is not exclusive to the Linux kernel but present in many large C projects such as QEMU, Node.js, Xorg, the Windows kernel, git, FreeBSD, and XNU.

List Iterators. As an example, consider the popular `list_head` structure that programmers embed in their data structures in the Linux kernel to create a double-linked circular list, with `next` and `prev` pointers pointing to the next and previous `list_head` element of the list. Iterating over a list, we know we have reached the end when we encounter the same pointer a second time. An empty list has its `next` and `prev` pointers pointing to itself. Issuing a `container_of` on a `list_head` allows access to the derived type, i.e., the element of the entry.

While there are different ways to use `list_head`, adding a linked list to a structure in the Linux kernel is a matter of embedding a `list_head` whose `next` field points to the first entry of the list, while that of the last entry points back to the `list_head` in the "owning" data structure. In this way, all list entries have the same type, except the owning structure that anchors the *head* of the circular list. Similarly, it is safe to issue a `container_of` from any list entry, except for the `list_head` in the owning structure, where it would lead to container/type confusion. The owning structure need not even be a struct, as it could also be a single `list_head` variable.

To iterate over a list, the kernel uses macros such as `list_for_each_entry`. It repeatedly follows the `next` pointer to find the next `list_head` and then uses `container_of` to set the iterator to the base of the entry that embeds it. For instance, we can iterate over all inodes of a superblock as shown in Listing 16.

This is safe if the possibly invalid list iterator, upon loop exiting, is not used afterwards. While the most common, `list_head` is not the only iterator in the Linux kernel but most work in a similar way. Well-known further examples include single-linked lists (`hlist_node`) and red-black trees (`rb_node`).

This chapter will highlight several cases where iterator invariants are violated, resulting in buggy code.

```
// parent struct
struct usb_request {
    void *buf;
    unsigned length;
    dma_addr_t dma;
    ...
}
// child struct
struct gr_request {
    struct usb_request req;    // member field
    ...
    struct gr_dma_desc *first_desc;
    ...
};
// child struct
struct goku_request {
    struct usb_request req;    // member field
    ...
    unsigned mapped:1;
};
```

Listing 14: Structure embedding example, where `gr_request` and `goku_request` "inherit" from `usb_request`.

4.2.2 Sanitizers

Sanitizers are runtime tools to detect undefined behavior in programs, typically through compiler-based instrumentation that checks undefined behavior. The best-known example is AddressSanitizer (ASan) [154], which detects memory errors such as buffer overflows and use-after-frees. ASan instruments every memory access with a check that consults a shadow memory to see if the memory access is valid. In particular, to detect buffer overflows, ASan pads memory allocations with *redzones* and poisons the memory in the shadow memory (setting it to a nonzero value) so that any future access results in an ASan error. In this chapter, we will repurpose ASan redzones to detect object boundaries.

4.3 Container Confusion in the Linux Kernel

In this section, we discuss security risks that can arise from container confusion, examine a real-world bug as a running example, and show to what extent the Linux kernel resorts to structure embedding.

4.3.1 Security Implications

Like C++'s `static_cast`, the `container_of` macro does not perform runtime checks to verify whether the structure is actually contained within the expected outer structure. When this is not the case, container confusion leads the program to access memory under wrong assumptions on its layout. Two base scenarios are possible: a) the structure

```c
#define container_of(ptr, type, member) ({
  void *__mptr = (void *)(ptr);
  ((type *)(__mptr - offsetof(type, member)));
})
```

Listing 15: `container_of` implementation in the Linux kernel.

is embedded in a different container, leading to member access over memory contents typed for another layout; or b) the structure is not embedded in a container, leading to a pointer that is out-of-bounds by the relative offset assumed within the container.

The security implications of bad casting have been well-researched for C++ (e.g., in the CaVeR paper [106]) and similarly apply here, being `container_of` equivalent to C++'s static downcasting. Such effects can range from subtle state corruptions to controlled out-of-bounds accesses that attackers can evolve for exploit construction. The security risk is mainly dependent on structure layouts, for example when memory containing function pointers can be overwritten. To probabilistically mitigate these and other issues, the Linux kernel can randomize the layout of some structures at compile time [72]. While this can make exploitation less reliable, in some cases it may also turn an unexploitable bug into a security vulnerability. In fact, as the offset for the embedded structure changes, also does the memory pointed by the type-confused pointer, directly affecting the bug exploitability (for example, when further memory corruption becomes possible under some randomized layouts). At the time of writing, only a few structure types (65 in the entire kernel) can undergo randomization: enabling it globally (as done in research operating systems [49]) can be difficult as code may assume a specific layout for some structures, while others have layouts that are tuned for better performance [37].

We will show concrete examples of security risks uncovered by the dynamic and static analyses of UNCONTAINED in Sections 4.6 and 4.7.3, where we outline, among others, a vulnerability that breaks Kernel Address Space Randomization (KASLR) and a controlled out-of-bound write. We will also discuss examples of bugs that may affect execution semantics.

4.3.2 Running Example

We discuss next our running example (Listing 17) involving the kernel USB stack to better illustrate container confusion.

The function `gr_dequeue()` iterates over a list of requests to find and remove the one matching the supplied `_req` argument. Under correct operation, `container_of(&ep->queue.next, struct gr_request, queue)` in the macro at line 6 takes the address of field `queue` in a `gr_request` list entry and subtracts a quantity χ=`offsetof(struct gr_request, queue)` to make it point to the entry itself.

However, if the list is empty or does not contain it, the execution leaves the list iterator variable `gr_req` with a container-confused pointer. As mentioned in Section 4.2.1, the list iterator would incorrectly reference the owning structure (i.e., the list head), which has `gr_ep` type. The confused `container_of` subtracts χ from the pointer to the field `queue` in this other structure: the result will point somewhere within structure `*ep`.

The exploitability of the bug depends on the position of field `req`, used at line 10, within `gr_request` structures. Listing 14 shows the partial structure layout. Had the position been "deeper", the resulting pointer could have reached and surpassed the

```
// owning data structure ->
//    struct superblock embeds 'struct list_head s_inodes'
struct superblock *sb;
// iterator ->
//    struct inode embeds 'struct list_head i_sb_list'
struct inode *inode;
  ...
list_for_each_entry(inode, &sb->s_inodes, i_sb_list) {
  spin_lock(&inode->i_lock);
  ... // do more with inode
}
```

Listing 16: Kernel code to use `list_for_each_entry` to iterate over inodes.

outer `gr_ep` structure, referencing the adjacent heap storage. Were `_req` to match such an out-of-bound pointer, the code attempts to remove a list entry that is not present, possibly causing further memory corruption.

Rich discussions followed our disclosure of the bug to the Linux kernel mailing list. As a result, the maintainers opted to migrate to the C11 standard, which would allow them to define the iterator variable with a scope limited to specific loops, preventing its usage afterwards. In the next section, we will examine the potential surface for container confusion cases in the Linux kernel.

4.3.3 Type Graph Complexity

To examine the use of structure embedding in the Linux kernel, we analyze the prevalence of `container_of` and its derivatives, as `container_of` takes part in several macros and inline functions. Depending on the selected kernel configuration, we note that the build system of the kernel can choose between different function implementations and even type definitions. Hence, we study the Linux kernel v.5.17 with the configuration in use to Google's syzbot [51] for continuous fuzzing.

We write an LLVM compiler pass to spot all the uses of `container_of` in the source code as lowered during compilation and track the parent and child types at each such use. This allows us to build a *type graph* that captures the possible containment relationships between different structure types. We count over $56,000$ downcast instances (as `container_of` or any of its derivatives) under our kernel configuration.

As the chapter will detail, the type graph is a foundational element of our approach to container confusion detection. Figure 4.1 shows the one being discussed here, highlighting the relationships between the embedded types. Each node represents a type involved in a downcast. We have a (directed) edge between two types if we find a downcast instance that derives a child of the destination node type from a parent of the source node type. We also compute edge weights based on the number of such instances.

While we count as many as 18323 types in all the code for the build, we find 4275 of them to be involved in downcast operations: 506 can occur as parent and 4033 as child object. To our surprise, this implies that almost one-fourth (23.3%) of all types are involved in structure embedding.

For example, the `usb_request` structure shown in Listing 14 can be embedded in 17

```
static int gr_dequeue(struct usb_ep *_ep,
                      struct usb_request *_req) {
  struct gr_request *gr_req; // renamed: was `req`
  ...
  struct gr_ep *ep = ...; // derived from `_ep`
  list_for_each_entry(gr_req, &ep->queue, queue) {
    if (&gr_req->req == _req)
      break;
  }
  if (&gr_req->req != _req) {
    ret = -EINVAL;
    goto out;
  }
  ...
}
```

Listing 17: Using the list iterator `gr_req` past its validity causes container confusion.

different child structures in use to different USB drivers. Generally speaking, a variety of destination types may favor cases of invalid runtime downcasts.

By looking at topological properties of the type graph, we find that 3486 of the 4033 possible destination types are not contained in any other type, meaning no other type “inherits” from them. 419 of the 506 possible source types have an out-degree greater than one, meaning that they can have multiple child types; 221 have more than 10 possible child types.

In the figure, we also highlighted the top-5 structure types by highest number of child types: `list_head` (1857), `work_struct` (611), `hlist_node` (244), `timer_list` (235), and `qspinlock` (223). Each colored cluster shows the possible destination types for such a source type during downcasting.

Looking at edge weights, the structure types most often used as parent when downcasting are `list_head` (22033), `inode` (7669), `device` (4130), `hlist_node` (3221), and `rb_node` (2272). Several of them are involved in iterators.

We also note that `list_head` emerges as the type with most child types that inherit from it and as the most used parent type across the whole kernel code base.

As the main takeaway of this study, we argue that the prevalence of `container_of` and derivatives, combined with the notable complexity of the type graph they induce, makes a compelling case for seeking container confusion bugs.

4.4 uncontained Overview

In this chapter, we design and implement UNCONTAINED to detect container confusion bugs in the Linux kernel.

In Section 4.5, we present a novel container confusion sanitizer that uses object boundaries to detect invalid downcasts during dynamic analysis. After describing the design and implementation, we evaluate effectiveness and performance of the sanitizer by combining it with the well-known syzkaller [52] kernel fuzzer and other benchmarks. Finally, we use the sanitizer to analyze the occurrence of container confusion in the

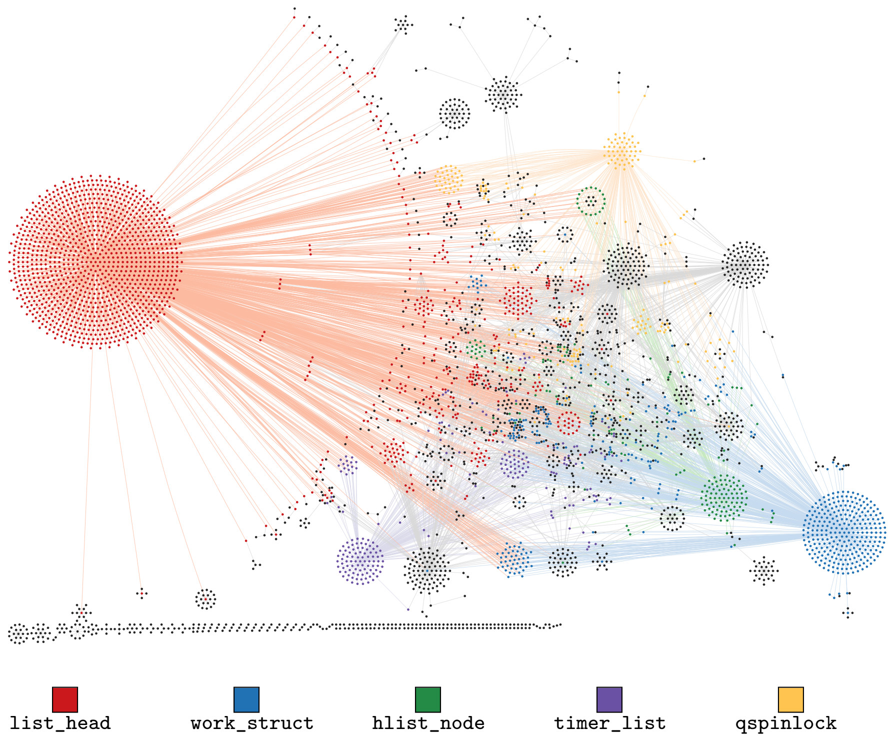


Figure 4.1: Type graph for `container_of` (and alike) instances.

Linux kernel.

Achieving code coverage with dynamic analysis on the Linux kernel can be challenging due to the amount of complex code. In Section 4.6, we therefore analyze the bugs we detect through fuzzing and identify common bug patterns that result in invalid `container_of` usage. Based on these patterns, we develop a static analyzer to search for additional bugs without suffering from the lack of code coverage inherent to dynamic analysis in Section 4.7. In particular, we design and implement a configurable LLVM forward and backward dataflow analysis to identify potentially buggy code patterns. We then analyze any additional bugs found by the static analysis, including a worrying out-of-bounds write, and demonstrate an acceptable rate of false positives. Although static analysis has lower accuracy than dynamic analysis, it acts as an effective complement for code that dynamic analysis fails to reach.

4.5 Container Confusion Sanitizer

This section introduces the sanitizer component of UNCONTAINED meant to detect cases of container confusion at runtime. We explain its design and implementation in Section 4.5.1 and Section 4.5.2, respectively, and evaluate it in Section 4.5.3.

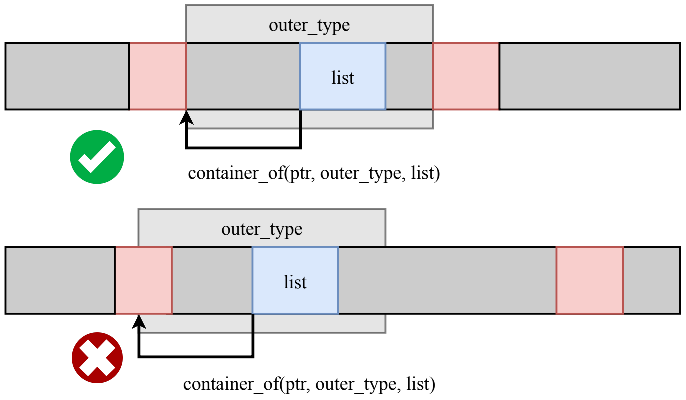

outer_type
list
container_of(ptr, outer_type, list)
outer_type
list
container_of(ptr, outer_type, list)

Figure 4.2: Redzone layout for a valid downcast (top) and for an invalid one (bottom). Here, *list* is the member field name.

4.5.1 Design

Our sanitizer aims to expose downcasts, represented by uses of the `container_of` macro in kernel, where an incorrect destination (i.e., child) type causes a container confusion. As we anticipated in Section 4.1, detecting such errors with existing approaches to type confusion detection would require maintaining a form of RTTI for each allocated object.

Our design aims instead for a general solution that does not incur code modifications and/or pointer tracking costs while achieving broad compatibility. The key idea is to turn a downcasting validity check into multiple bound checks relative to the current embedded object (the parent) and the requested container object (the child) of a `container_of` operation. Parent and child here are synonyms for *inner* and *outer* structure.

We analyze structure definitions and use the relative distances of an embedded structure from the start and the end of its container structure as the discriminating factor for violations. When the container object is of the requested type, its allocation boundaries will align perfectly with those that one can infer starting from the parent pointer. A violation occurs instead when the object enclosing the parent turns out to be larger or smaller than expected on either side.

To insert sanitization checks, inferring the expected boundaries of a child object is straightforward, as both its size and the displacement of the parent field from its start are known at compile time. However, even at runtime, the actual boundaries of an object are normally not available in C programs.

Object Boundaries. For reliable boundary identification, we rely on standard runtime means in use to sanitizers that target spatial memory safety violations. Namely, we pad object allocations with redzones (Section 4.2.2) and use them to recover object boundaries. The addresses immediately preceding and following an object will appear as invalid in the shadow memory, while those at the boundaries will be valid.

For a `container_of` operation, we can thus check for the validity of memory at the expected start and end addresses of the requested container, and the invalidity of the memory right before and after them, respectively. This will readily expose mismatches

```c
static int gr_dequeue(struct usb_ep *_ep,
                      struct usb_request *_req) {
  struct gr_request *gr_req; // renamed: was `req`
  ...
  struct gr_ep *ep = ...; // derived from `_ep`
  list_for_each_entry(gr_req, &ep->queue, queue) {
    if (&gr_req->req == _req)
      break;
  }
  if (!check_redzone(gr_req, sizeof(struct gr_request))) {
    uncontained_report(gr_req);
  }
  if (&gr_req->req != _req) {
    ret = -EINVAL;
    goto out;
  }
  ...
}
```

Listing 18: Running example with our bound checks added.

between expected and actual boundaries.

Figure 4.2 shows an example of valid and bad downcasting, highlighting the differences in their object redzone layouts.

We chose a redzone-based approach over other bounds-tracking designs due to its efficiency, practicality, and compatibility with complex code bases: mainly, inspections have O(1) cost and we can build on an existing, well-tested infrastructure from memory sanitizers for kernels. Alternative design points such as low-fat pointers [103] remain a possibility.

Container Nesting. The bounds-checking policy we just presented may mishandle containers that are embedded in another container. For those cases, we cannot expect the presence of redzones for the inner container, being it a structure field. However, we can still do the validation through the outer container. In the Linux kernel, only 547 of its 4033 container types may incur such a scenario, whereas for 3486 no nesting is possible. Therefore, when the desired child type of a `container_of` instance is one of those 547, we apply the following scheme if the normal bound checks fail.

We note that a `container_of` operation carries the expected type for the innermost container only. Moving to an outer container, we can check if its boundaries (i.e., the redzones around it) align with the layout expected for any of the container types that have a field of the expected inner container type. This information is available in the type graph (Section 4.3.3) at compile time and we compute it recursively for multi-nesting cases. If the redzones of the outermost container do not match any feasible layout, we report a container confusion error.

This strategy effectively allows us to avoid false positives from container nesting. The attentive reader may notice that, by accepting more redzone layouts as valid, we open the door to more false negatives: however, as we will show later in this section, the probability of such layout collisions is very low. Ideally it would be possible to inject

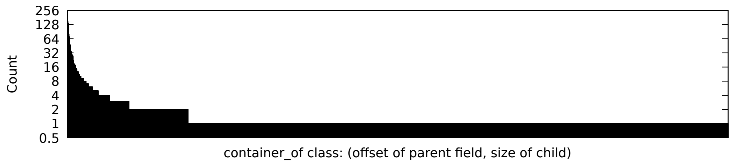


Figure 4.3: Distribution of `container_of` invocations according to offset of parent field and container size. Logarithmic scale.

redzones between struct members to both make the analysis more accurate and reduce the complexity of analyzing outer structs. Unfortunately, many structs in the kernel are carefully optimized to fit within certain boundaries or represent a specific amount of memory that make inserting such redzones non-trivial and would require a kernel-wide effort to support.

Time-of-use Checking. In the Linux kernel code, we found several cases where a `container_of` instance sees at runtime also objects of an incompatible type but the following code is never affected by the confusion. For example, with list iterators, the obtained child pointer was used only to access the parent again through the child field corresponding to it. These cases in the programming practice are not strictly bugs. Therefore, in our design, we opted to validate a `container_of` instance at the time of use for its output pointer rather than immediately when downcasting. Listing 18 shows our running example augmented with bound checks around redzones.

To identify uses of the output pointer, we run a standard intra-procedural def-use [65] analysis. As the program may modify it before dereferencing it (e.g., to access a child field), we analyze pointer arithmetic operations and, when the modification can be determined statically, we forward the check to the next use of the pointer. When the program dereferences it or we can no longer follow it statically, we emit bound checks and have them account for the modified offset, if any.

Discussion. The sanitization scheme we propose can detect container confusion by relying solely on structure layout knowledge (known at compile time) and object boundaries (obtainable with off-the-shelf lightweight techniques). When both sources are accurate, no false positives are possible.

Compared to an ideal design that tracks pointer types, the price we may pay for our efficiency and compatibility relates to false negatives when an invalid downcast involves an object whose layout coincides with the one of a valid child type[1].

To look into this dimension, we identify a domain and a codomain for it. As domain, we study how many unique `container_of` instances are present in the Linux kernel as we consider the pair (parent field, child type) for a downcast operation. We include the field as one child may embed multiple parents. As codomain, we identify pairs of the form (offset of parent field, size of child) for such operations, since these are the two quantities that we use—independently from one another—for bound checking. We

[1]We deem a container confused if not immediately preceded (resp., followed) by a redzone byte and if its first (resp., last) byte is valid memory. With a false negative, the former check lands on invalid memory and the latter on valid memory. Note also that this property is not affected by the redzone size.

count 6,526 unique instances mapping to unique 3,262 pairs. A collision occurs when two distinct instances map to the same pair.

The distribution in Figure 4.3 shows that 40.8% of the unique `container_of` instances map to one pair exclusively, 16.9% to 2-4 pairs, 21.1% to 4-32 pairs, and only 5 of them to 100 or more pairs. Hence, we expect collisions to be infrequent. We then analyze them under the realistic hypothesis that incorrect downcasts happen only over objects of related types. When counting all the siblings and descendants in the type hierarchy for the expected downcast type of a unique `container_of` instance, we measure the probability of a collision to be 0.0283, which decreases to 0.0088 when considering siblings only.

Note also that one may avoid false negatives almost entirely by adding padding bytes to structures mapped to the same codomain point(s). We leave this investigation to future work.

4.5.2 Implementation

The sanitizer of UNCONTAINED consists of two components. The first one is a coccinelle [130] script to intercept occurrences of `container_of` at the source level, which the C preprocessor would otherwise expand before we may instrument them.

The second one is a pass for the intermediate representation (IR) of the LLVM compiler (v.12.0.1) implemented in 1640 lines of C++ code. The pass is responsible for building the type graph of the code base, expanding the intercepted `container_of` instances, and adding sanitization machinery.

We also develop a framework[2] of potentially independent interest to apply custom LLVM passes during kernel compilation and run VMs for testing (e.g., with syzkaller) and debugging, automatically spawning one with a breakpoint attached to the found crash site for manual inspection in gdb.

To have full visibility on type information, we run our pass as a link-time optimization. We then leverage the existing redzone insertion and shadow memory mechanisms of Kernel Address Sanitizer (KASAN) [98] to support object boundary identification for stack, global, and heap-allocated variables. While our sanitizer can coexist with KASAN's machinery to sanitize memory accesses for safety violations, we disable its generation as these checks are unnecessary for our purposes.

As mentioned in the previous section, correct object boundary identification is essential for precision. This aspect is not influenced by the redzone size (for which we use KASAN's defaults), as the shadow memory has always 1-byte granularity. However, even state-of-the-art techniques for redzones fail to handle the edge cases we discuss next. As they may lead to false positives, we disable confusion checks for them.

We find two object allocation schemes that require special handling. One involves a known limitation of redzones with arrays: in these cases, redzones cannot be inserted around their individual elements, unless one modifies the type definition. With a coccinelle script, we identify in the code base all the types that take part in array allocations and disable the validation of `container_of` instances using them as a child type. For future work, we are considering the addition of machinery to test all possible array cells when their number is known statically, whereas for dynamic sizes the recent proposal of bounded flexible C arrays [31] may be of help.

The second scheme involves the allocation of multiple, differently typed structures (e.g., `kalloc(sizeof(A) + sizeof(B), ...)`) followed by pointer extraction for each

[2]Available at `https://github.com/Jakob-Koschel/kernel-tools`.

structure. This coding choice brings performance benefits, as it optimizes the use of the allocator, but complicates memory sanitization schemes. To avoid false positives for objects involved in such allocations, we devise a coccinelle script to disable the involved types from validation. However, for a few recurring cases and if code semantics allowed doing so safely, we manually split allocations and enable container confusion detection for types like `io_buffer` used in `io_uring` code or `net_device` private data in networking code.

Overall, for the two schemes, we disable validation for 13, 926 out of 56, 468 downcasts. We also highlight that the shadow memory and redzones of KASAN operate only after the early boot phase of the kernel. Heap objects allocated by the boot memory allocator `memblock` have no redzones: we identify and skip them using address range checks at runtime.

While we test and evaluate our sanitizer around the Linux kernel, the adaptations needed for other subjects would be limited. Redzone management for userland software is available in LLVM with AddressSanitizer [154], while kernels like FreeBSD and XNU have their own KASAN implementation.

4.5.3 Evaluation

We run our sanitizer on the Linux kernel v.5.17 (commit `c269497d248e`). For the fuzzing experiments, we use syzkaller (commit `9e8eaa75a18a`) and build two images compiled, respectively, with the default kernel configuration and the one in use to Google's syzbot [51], as it enables additional features. The choice is an attempt to slightly balance the exploration of code between pervasiveness and breadth.

To stress specific/additional components, we also run typical userland workloads such as installing programs with the `aptitude` package manager, executing `binutils` utilities, code for SGX enclaves, and the Linux Test Project [104].

As experimental setup, we ran syzkaller for one week on two Ubuntu 22.04.1 (Linux kernel v.5.15) host machines with 16 cores @2.3GHz (AMD EPYC 7643), using a total of 16 QEMU-KVM virtual machines with 4GB RAM and even distribution of the default and the syzbot-configured builds.

Discovered Cases of Container Confusion

Our fuzzing campaign revealed 37 cases of container confusion. After manual analysis of the crash sites, we identified 11 unique bugs and 10 *anti-patterns* (see below). The remaining 16 are false positives deriving from missing redzones in mixed-type allocations that our coccinelle scripts miss (Section 4.5.2). Adding them to our filtering logic is a one-time effort that would prevent such false positives from occurring in future campaigns.

We consider *anti-patterns* type confusion cases where the use of a confused pointer is a "controlled" case of undefined behavior as the code does not incur a corruption only thanks to implicit assumptions on program semantics (which may silently change over time) and/or compiler behavior. Such anti-patterns might silently turn into bugs in future releases.

The 11 bugs affect the following kernel subsystems: `drivers/net`, `net/{ipv4&6, sctp}`, `fs/f2fs`, and `sgx`. We disclosed and proposed patches to the maintainers for all the bugs: at the time of writing, all patches have been or are being merged. We present five of these bugs in Section 4.6. The 11 bugs had not emerged, e.g., in the continuous fuzzing efforts from Google's syzbot, which uses state-of-the-art sanitizers like KASAN and tests several configurations.

The 10 anti-patterns relate to places where a container confusion occurred but developers manage it explicitly later. As examples, we briefly describe two of the anti-patterns that our sanitizer found.

The first involves the function `crypto_alg_lookup()` of the Kernel Crypto API. The function can return a pointer to a synchronous-hash structure (`shash_alg`) confused as if it were an asynchronous (`ahash_alg`) one. However, all the users of the function eventually check the requested instance type through additional fields to differentiate them and correctly cast the confused pointer before use.

The second involves the `inet_lookup_established()` networking function, which can return a pointer to a `struct inet_timewait_sock` confused as a `struct sock`. Similar to above, all the users of the function check the socket state to differentiate them.

Runtime Overhead

We conduct two sets of experiments to measure the overhead introduced by the sanitizer component of UNCONTAINED: the bare sanitization costs with LMbench [119] and their impact on the end-to-end throughput when fuzzing with syzkaller.

We run the LMbench programs on a single QEMU-KVM instance with 8 GB of RAM executing on an i7-10700K CPU host machine with minimal background activity and identical software to the previous experiments. We repeat each experiment 10 times, taking the median value for every program. Our sanitizer introduces a geomean overhead of 74%. As a reference, KASAN introduces a 126% overhead (with 33% coming from redzone management, which we use too). We list figures for the individual programs in Section 4.B.

For fuzzing throughput, we measure how many test cases one syzkaller VM executes within the first hour of fuzzing. We take the median value of 10 experiment repetitions, starting from an empty fuzzing corpus. The syzkaller baseline with no sanitizers enabled executed 80348 test cases, whereas with UNCONTAINED 69734 with a net reduction of the fuzzing throughput of around 13%. As a reference, KASAN introduced a 55% net reduction of the throughput. We find our approach to induce an overhead[3] acceptable for fuzzing.

4.6 Retrospective Analysis and Bug Patterns

The cases of container confusion that our sanitizer detected when fuzzing revealed several lingering bugs and anti-patterns in the Linux kernel. Their analysis brought out two key reflections we present next, as they motivate and form the basis of the research from the remainder of the chapter.

Unexplored Code. In spite of the widespread use of containers, the issues found were located in a fairly limited, yet relevant, subset of the Linux kernel code base. Prolonging the fuzzing campaign by a few days did not uncover new bugs.

We find this to stem directly from the inherent coverage problem of dynamic tools. Much code may be locked under specific kernel states [63, 201], require emulation for

[3]One opportunity to reduce it would be to follow [156] by disabling stack walking upon memory (de)allocation events, as it helps only for crash debugging/deduplication but is expensively frequent. Each crash may be analyzed offline by re-running the test case in an unmodified KASAN.

crossing the hardware/software barrier with device drivers [136], or need complex input generation logic (e.g., with protocols). Special-purpose fuzzers [35, 132, 136, 149, 156, 160, 161, 167], which one may run naturally on our instrumented kernels, currently exist only for a fraction of such components.

This led to us eventually to investigate container confusion detection through static approaches that could cover the whole code base, even if with a diminished precision/recall.

Dynamics of Bugs. We noted a few distinctive traits in the nature of the bugs spotted with the experiments of Section 4.5.3. These may make some bugs harder to reason about, especially for static analysis. However, as we show in Section 4.7, domain knowledge (e.g., on list operations) can come to the rescue.

For example, one trait relates to whether, for a `container_of` instance that sees objects incoming from a given program path, confusion occurs on all or only a few of them (e.g., only on a list's owning element). Another relates to whether, on the path(s) from the container allocation to its confused use, pointer upcasts and downcasts involve indirection (e.g., the address is stored in a field of another object).

In the following, we present five bug patterns that encompass all the issues of Section 4.5.3 and represent general forms of container confusion. These patterns are distinct, albeit not exhaustive in terms of possible types of confusion (other than those we encountered). Most importantly, the descriptions we give are actionable for program analysis (Section 4.7).

Pattern ❶: Statically Incompatible Containers. This pattern describes the most generic and shallow container confusion that we identified. It involves using a type (or member field) that is always incorrect when downcasting object pointers incoming from a certain program path.

Listing 19 reports an exemplary bug found when fuzzing in the `sock_init_data()` function while manipulating a `socket` struct. The function assumes that its `struct socket* sock` parameter is embedded in a `socket_alloc` container. This assumption is correct for most sockets in the kernel, except for TUN and TAP ones. Hence, when a program path from function `tun_chr_open()` reaches the buggy function, its argument is embedded in a `tun_file` container instead.

When the function assigns the socket with the owner's UID, the confused bytes are always set to zero in the kernel configuration that we tested. Any TUN or TAP socket thus appears as owned by the root user, nullifying user-based firewall/routing rules possibly in place. The severity of the bug may be even amplified by the effects of structure randomization (Section 4.3.1). At the time of disclosure, the bug had been present in the Linux kernel for more than 6 years.

Pattern ❷: Empty-list Confusion. As we anticipated in Section 4.2.1, a confusion can originate when issuing a `container_of` operation on the owning structure of a circular list. When such a list is empty, the owning structure sees the `next` and `prev` fields of its embedded `list_head` point to itself. Accessing list members in a `list_entry`[4], `list_first_entry`, or `list_last_entry` operation causes container confusion.

Listing 20 reports an exemplary bug found in the kernel networking stack when fuzzing. Since the `inet_diag_msg_sctpasoc_fill()` function assumes that the `asoc->base.bind_addr.address_list` list is populated without checking for it, `laddr` points to a container-confused object when the `list_entry()` operates on an empty list. The

```c
static int tun_chr_open(struct inode *inode, struct file *file) {
  struct tun_file *tfile;
  ...
  sock_init_data(&tfile->socket, &tfile->sk);
  ...
}

struct inode *SOCK_INODE(struct socket *socket) {
  return &container_of(socket,
    struct socket_alloc, socket)->vfs_inode;
}

void sock_init_data(struct socket *sock, struct sock *sk) {
  if (sock) {
    ...
    sk->sk_uid = SOCK_INODE(sock)->i_uid;
  } else {
    ...
  }
  ...
}
```

Listing 19: The first argument to `sock_init_data()` is contained within `tfile` when called from `tun_chr_open()`. `SOCK_INODE()` incorrectly assumes `sock` to be contained within a `socket_alloc` struct.

code at line 11 copies some of its fields into memory provided to user space. As these confused fields contain kernel heap pointers, this results in a KASLR leak that deterministically breaks the address randomization of the kernel, which often represents one of the first steps in kernel exploitation [59, 71, 81, 101]. At the time of disclosure, the bug had been present in the Linux kernel for almost 7 years.

Pattern ❸: Mismatch on Data Structure Operators. Insertion, deletion, selection, and other operations on objects taking part in container-based data structures (e.g., lists, trees) should see the use of consistent types and member fields.

Listing 21 shows an exemplary bug found when fuzzing involving the `sock` structure. A `struct sock` can be inserted into multiple lists and therefore embeds multiple list structures. Among others, it contains two single-linked lists using the fields `sk_bind_node` and `sk_node`. With a list, its elements must always be accessed via the field used to insert them into it. The socket code manages the `&tb->owners` list, which holds sockets using their `sk_bind_node` member. But `__inet_hash_connect()` accesses the same objects using the `sk_node` member. In this case, the two members are located at different offsets, thus the downcast on the access adjusts the pointer incorrectly, causing container confusion.

As a result, the condition at line 17, which controls a fast path for the function, never evaluates to true. At the time of disclosure, the bug had been present in the Linux kernel for 18+ years (i.e., the extent of its git history).

[4]We recall that `list_entry` is simply an alias for `container_of`.

```c
static void inet_diag_msg_sctpasoc_fill(
        struct inet_diag_msg *r,
        struct sock *sk,
        struct sctp_association *asoc) {
  union sctp_addr laddr;
  ...
  laddr = list_entry(asoc->base.bind_addr.address_list.next,
    struct sctp_sockaddr_entry, list)->a;
  ...
  if (sk->sk_family == AF_INET6) {
    *(struct in6_addr *)r->id.idiag_src = laddr.v6.sin6_addr;
    ...
  }
  ...
}
```

Listing 20: `list_entry()` assumes the presence of at least one entry within `asoc->base.bind_addr.address_list`, causing a container confusion in `inet_diag_msg_sctpasoc_fill` due to the missing check for whether the list is empty.

Pattern ❹: Past-the-end Iterator. Developers often rely on a break-like logic when searching for an element in a data structure using iterators. Program semantics may sometimes deceive them into believing that a search will always succeed, so they may use an iterator without checking for its validity, which would not hold if the loop completes.

This container confusion characterized our running example (cf. Section 4.3.2). Listing 22 shows another exemplary bug that we found in SGX code when running an enclave in our instrumented kernel build using `qemu-sgx`. As the function processes an empty `&encl_mm->encl->mm_list` list, the `tmp` iterator is never assigned a valid entry, holding a confused pointer after the loop. At the time of disclosure, the bug had been present in the Linux kernel for more than 2 years.

Pattern ❺: Containers with Contracts. An object embedded in a data structure may come with additional metadata (e.g., custom RTTIs [106]) that program semantics uses as an implicit *contract* to control what operations can be done on it.

This is the case with the *sysfs* subsystem of the kernel, which lets user-space programs inspect and control several kernel features. Listing 23 shows a container confusion that we found in an inspection function when fuzzing. Here, the `kobject` that `kobject_init_and_add()` registers is not embedded in another structure, but the buggy `f2fs_attr_show()` function treats it as if embedded in a `f2fs_sb_info` structure.

This plays out as a "controlled" confusion, as the contract (i.e., the companion object of type `ktype` at line 3) carries a pointer, retrieved at line 11, to a function that does not access the confused `sbi` supplied at line 12. We classify this as an *anti-pattern*, as an imperfect knowledge of program semantics or changes to it would open up the possibility for bugs.

Bug Counts. With our sanitizer (Section 4.5.3), we discovered 6 mismatches on data structure operators, 2 cases of empty-list confusion, and 1 case for each of the other patterns.

```
void inet_bind_hash(struct sock *sk,
        struct inet_bind_bucket *tb,
        const unsigned short snum) {
  ...
  hlist_add_head(&sk->sk_bind_node, &tb->owners);
  ...
}

int __inet_hash_connect(..., struct sock *sk, ...) {
  ...
  struct inet_bind_bucket *tb;
  ...
  if (port) {
    ...
    tb = inet_csk(sk)->icsk_bind_hash;
    ...
    if (hlist_entry((&tb->owners)->first,
        struct sock, sk_node) == sk &&
        !sk->sk_bind_node.next) {
      inet_ehash_nolisten(sk, NULL, NULL);
      spin_unlock_bh(&head->lock);
      return 0;
    }
    ...
  }
  ...
}
```

Listing 21: `inet_bind_hash()` inserts list elements using the `sk_bind_node` member, whereas `__inet_hash_connect()` accesses them incorrectly using the `sk_node` member.

4.7 Static Analyzer

This section introduces the static analyzer component of UNCONTAINED, which aims to identify the container confusion patterns presented in the previous section. We illustrate the design of our static analyses in Section 4.7.1, their implementation in Section 4.7.2, and the experimental results in Section 4.7.3.

4.7.1 Design

As anticipated in Section 4.6, our static analyzer aims for the code regions that are not within easy reach of current dynamic testing solutions. We note, though, that the reflections and bug patterns we presented involve phenomena, like indirection via memory, that may be expensive to reason about statically. Also, most of the bugs found involved inter-procedural flows.

For our analysis to scale to a code base as huge as the Linux kernel while maintaining satisfying accuracy, we make the following design choices. We cast bug pattern search to a static *information flow analysis* problem, relying on def-use information to track value

```
void sgx_mmu_notifier_release(struct mmu_notifier *mn,
                              struct mm_struct *mm) {
  struct sgx_encl_mm *encl_mm = ...;
  struct sgx_encl_mm *tmp = NULL;
  ...
  list_for_each_entry(tmp, &encl_mm->encl->mm_list, list) {
    if (tmp == encl_mm) {
      list_del_rcu(&encl_mm->list);
      break;
    }
  }
  ...
  if (tmp == encl_mm) {
    synchronize_srcu(&encl_mm->encl->srcu);
    mmu_notifier_put(mn);
  }
}
```

Listing 22: Incorrect use of the list iterator variable `tmp` after the loop in `sgx_mmu_notifier_release()`.

propagation. The five bug patterns become rules for an on-demand backward or forward analysis where `container_of` instances act as sources or sinks depending on the pattern. We extend def-use chains through procedure boundaries (as a simplified form of [65]) and model memory as a single, coarse-grained symbolic location for scalability. We use semantic knowledge of common data structure manipulations (e.g., list iterators) to model several flows that involve indirection, enabling static reasoning.

We provide descriptions below for how we encode the five bug patterns as rules for the information flow analysis. Section 4.C contains more rigorous definitions of what we use as (and do at) sources, sinks, and path-discarding filters.

Pattern ❶. To spot statically incompatible containers, we run a backward analysis from the pointer supplied to a `container_of` instance to every operation, if any, that obtains a pointer to an embedded structure starting from a pointer typed as a container. If the type (or member field) is incompatible with what `container_of` is asked for, we report a confusion.

Static reasoning is limited to instances for which we can infer the container type, i.e., cases where the code computes the parent structure pointer flowing into `container_of` by referencing the member field of the child structure—e.g., with a `&(child.member)` pattern. Our static reasoning gives up instead if the code reads the parent pointer value directly from memory: in these cases, even complex pointer analyses may be inconclusive due to aliasing, indirection, and other factors.

Pattern ❷. To spot potential accesses on empty lists, checking only for the use of dedicated helpers (e.g., `list_empty`, `list_is_head`, `list_entry_is_head`) would be prone to false positives. In fact, a code may keep track of the list size in a separate variable and check it before any downcasting; we find this to happen frequently in the Linux kernel.

```
...
    ret = kobject_init_and_add(&f2fs_feat,
      f2fs_feat_ktype,
      NULL, "features");
...
ssize_t f2fs_attr_show(struct kobject *kobj,
                        struct attribute *attr, char *buf) {
  struct f2fs_sb_info *sbi = container_of(kobj,
            struct f2fs_sb_info,
            s_kobj);
  struct f2fs_attr *a = ...;
  return a->show ? a->show(a, sbi, buf) : 0;
}
```

Listing 23: Invalid `container_of` on `kobj` (originating from `&f2fs_feat`) in `f2fs_attr_show()`.

We thus conduct a forward analysis from any occurrence of `list_{entry, next, prev, first, last}` to any use of the output pointer. If we encounter no conditional check guarding a use in the control flow, we report a potential confusion.

When reviewing buggy code, we also noted that some code erroneously compares the assigned pointer to `NULL` (whereas, when the list is empty, the result would reference the owning structure). Therefore, we added an analysis that detects such checks and deems them as *incorrect* (unless the code did not explicitly initialize the pointer as such before list iteration).

Pattern ❸. Object flows between operations involving container-based data structures (e.g., insertion and retrieval in a list) are in general hard to reason about statically, as they involve memory contents manipulation. However, we can rely on domain knowledge on the identity of the operations to detect cases of container confusion from inconsistent member selection.

We do a forward analysis from any operation on a data structure type to any subsequent operation on the same structure (e.g., from `list_add` to `list_entry`). If the pointers supplied to both can be determined to be the same but the container type or field is different, we report a potential confusion.

Pattern ❹. To detect when an iterator may have outlived its validity and cause container confusion if dereferenced, we analyze the instances of iterator-related macros that take part in loops. For each of them, we conduct an intra-procedural forward analysis to see if the code uses it outside the loop. We deem such a use as potentially confused if it is not guarded by a conditional check (e.g., using a boolean variable set by the loop), as developers typically insert one to assess whether the loop stopped advancing the iterator (i.e., before invalidity).

Pattern ❺. Confusion cases on containers with contracts are hard to spot in terms of code manipulations alone. We find it reasonable to assume that, for a given code base, the identity of such container types is known. For the Linux kernel, we devise an analysis for `kobject` containers that one may in principle adapt to other types from other code bases. The analysis comes with a forward and a backward component.

For each occurrence of the `kobject_init_and_add()` function, which is designed to register an object with its contract, we run a backward analysis to identify the containment relationships of the registered object and collect its `ktype` contract.

For each contract, we gather what functions of *sysfs* may be called on the object by inspecting its related fields. Then, we run a forward analysis from the `kobject` argument in each such function, looking for `container_of` invocations incompatible with any valid containment identified by the backward component.

4.7.2 Implementation

We implement the general forward and backward information flow analyses and the rules for patterns ❶, ❷, ❸, and ❺ as a pass for LLVM IR in 1286 lines of C++ code. Similarly to the dynamic analyzer (Section 4.5.2), we intercept every `container_of` occurrence at the source level and expose its source and destination type and object at the IR level. We run the pass at link time, so we can effectively extend def-use chains across procedure boundaries. However, in this scenario LLVM would normally merge type definitions having an identical memory layout: to keep our analyses accurate, we disabled this behavior by changing ~25 lines of code in the compiler.

The forward analysis starts from an IR value representing a source and follows its uses. When a use eventually reaches a function call argument, the analysis continues by seeing the uses of the arguments in the callee, recursively. The analysis also accounts for uses that concur to the return value of a callee, returning to the caller for continuing the analysis.

The backward analysis proceeds from a source IR value to its reaching definition(s). When it meets a function argument, it continues by exploring the code of each possible caller.

Both analyses stop exploring a path upon reaching a sink, a memory dereferencing operation (as we modeled memory as a single location), or an instruction already visited when analyzing a particular source. The rules for the patterns to check specify sources, sinks, direction of the exploration, and filters (if applicable) to stop a path exploration early.

Since our analysis visits each instruction at most once for each source location, and source locations are generally limited in number, we can approximately estimate the cost of our analysis as linear in the number of LLVM IR instructions.

As an implementation refinement, for pattern ❷ we suppress false positives involving container confusion in functions passed as callbacks for `list_sort()` or `seq_⌋ operations` structures. The reason is that the latter come with additional logic for emptiness checks before invoking the callbacks.

To ease the analysis of the reported confusion cases, we implement a Visual Studio Code plugin that recovers and presents to the developer the relevant code locations involved.

For pattern ❹, when reporting the bug presented in Section 4.3.2, the kernel maintainers pointed us to a coccinelle script proposed in 2012 by Julia Lawall on their mailing list to flag uses of iterators after loops. We assume that it had limited impact because of the high false positive rates. However, since our analysis for ❹ is simple and local, coccinelle is a great fit for it. We therefore extended the script in ways (mainly, with detection of checking logic already in place) that significantly reduced its false positive rate.

Description	FP	AP	Bug
❶ Statically Incompatible Containers	72	27	3
❷ Empty-list Confusion	19	4	20
❸ Mismatch on Data Structure Operators	16	8	1
❹ Past-the-end Iterator	0	137	56
❺ Containers with Contracts	0	3	0

Table 4.1: Reports from the static analyzer categorized as False Positives (FP), Anti-Patterns (AP), and Bugs for each pattern.

4.7.3 Evaluation

We run our static analyzer on the same kernel code base studied in Section 4.5.3. Table 4.1 summarizes the findings from a manual analysis of the reported cases of potential container confusion: we identified 80 bugs, 179 anti-patterns, and 107 false positives. We disclosed and proposed patches (144 in total with 97 already merged at the time of writing) for all the bugs as well as for the anti-patterns that can be removed without intrusive program semantics changes.

For the analysis time, we recall that pattern ❷ employs two rules whereas the others just one (❺ included, as its two analyses run in combination). We measure it took an average of 33.6 seconds for a rule to process all the container downcasts in the code that meet the definition of source for it.

We classify a report as a *bug* when the container confusion is unintended, which can lead to errors and possibly security-sensitive behavior. We consider as *anti-pattern* (AP) those cases where confusion can happen but program semantics prevents any use of the pointer. We consider as *false positive* (FP) those cases where pointers cannot have a confused value but the over-approximation of static analysis fails to see it.

Pattern ❶. Reports about *Statically Incompatible Containers* cases include 3 bugs, 27 anti-patterns, and 72 false positives. This pattern is prone to false positives (67.3% of the total among all five patterns) due to imprecision of the static analysis: we found most of them to occur when some backward control flows are unfeasible as they are guarded by checks on fields carrying explicit type tags[5]. A similar semantics is also behind most of the anti-patterns we found. As for the bugs, static checking identifies the TUN bug from fuzzing that we discussed when presenting the pattern in Section 4.6, but also a similar variant for TAP socket interfaces.

Pattern ❷. Reports about *Empty-list Confusion* cases are the second most numerous: we found 20 bugs (5 from missing checks and 15 from checks against `NULL`) and 2 anti-patterns.

For example, we found a container confusion in code that incorrectly checks HID device drivers reports, affecting all the 9 kernel drivers that rely on it. The bug had been present in the kernel for almost 9 years. In other HID driver code, we found 2 use-after-free and 1 NULL pointer dereference bugs. We also found a bug in the RT scheduler for an incorrect check on the task queue that had been present for 15 years.

[5]It could be a one-time effort to add such domain knowledge to the checker and stop the analysis of the current path upon recognizing such explicit checks over fields. However, we found 72 false positives here to still be a reasonable number for the manual analysis we conducted.

The 19 false positives involve lists that cannot be empty due to program semantics, missing effects of indirect calls (like the sort comparators that we model already), and implementation limitations for non-nearby conditional checks.

Pattern ❸. We found a notable bug by looking for pattern *Mismatch on Data Structure Operators* cases. The bug affects the function `rds_rm_zerocopy_callback()`, which writes a cookie provided by user space to memory. The function issues a `list_entry()` directly on the `list_head` instead of using `list_first_entry()`. The code passes the container-confused pointer to a function that finalizes the write.

The function uses confused values to write data to an offset where both are under user-space control, offering a controlled out-of-bounds (OOB) write primitive. Due to the container confusion, also an overlapping `lock` structure gets corrupted in the process, de-synchronizing it and potentially causing a use after free. The bug had been present in the kernel for 5 years. As the OOB write does not overlap with redzones, ongoing continuous fuzzing efforts could not detect it.

Anti-patterns mainly originate from iterating a list with an incorrect type, sharing a few initial member fields with the intended type. False positives come from implementation limitations with complex cases of GEP instructions in LLVM IR and unfeasible control flows from switch-case constructs.

Pattern ❹. Reports about *Past-the-end Iterator* are the most numerous in our results: this is quite expected, being list iteration popular in the kernel. We identify 56 bugs and 137 anti-patterns where the code may use a list iterator without checking whether it surpassed the end of the data structure.

The most immediate effect of our reporting and patching activity was upgrading the C standard for the Linux kernel to C11 [32]: this makes it possible to declare iterators valid only within loops, forcing developers to use (valid) retrieved values in a safer way. Shortly after, Linus Torvalds and other maintainers followed up with a proposal under adoption for a safer design of list iterators [34] that prevents anti-patterns of this kind completely.

Pattern ❺ We conclude by briefly mentioning that our reports from searching for *Containers with Contracts* cases uncovered two anti-patterns involving `kobject` container confusion in addition to the one discovered by dynamic analysis.

4.8 Discussion

We find that the dynamic and static components of UNCONTAINED operate synergetically to expose typically different instances of bugs over large code bases such as the Linux kernel.

Thanks to precise runtime information, the sanitizer component offers high accuracy by incurring only a few false positives in our tests.

The wealth of information also allows it to detect bugs that are out of reach of the static analyzer due to the latter's inherent under-approximation (e.g., for cases of memory indirections that we cannot recover via domain knowledge). This can be seen in the limited overlap in the bugs found: only 2 of the 11 bugs found dynamically occur in the reports of the static analyzer.

On the other hand, the static analyzer succeeds in its intended goals, revealing a large number of bugs (80) originating often in kernel areas that the dynamic experiments did not stress sufficiently or at all—and are also fundamentally difficult to cover due to configuration and hardware entropy. These include virtual drivers, ptrace facilities, the RT scheduler, and the kernel components of NFS and KVM, among others. Being a static analysis, the main shortcoming of the approach when it comes to analyzing reports is the lack of actionable test cases to reach the involved code. While this is an inherently hard problem for any static analysis, the patterns that we propose are quite intuitive, greatly helping manual analysis.

The majority of false positives come from pattern ❶, primarily because the static analysis is currently unable to recognize explicit type checks on structure fields that act as runtime type information and prevent container confusion bugs (Section 4.7.3). Therefore, violations of pattern ❶ can be regarded with lower confidence compared to the other patterns.

False negatives in the static analysis may be caused by incomplete control-flow information (e.g., indirect calls) and by inaccuracies in our modeling of program state. For example, precise modeling of memory may be an area worth examining to improve the reach of the static analyzer. We opted not to use pointer analyses as accurate ones are expensive on large programs [165] and features desirable in this context (e.g., flow- and context-sensitivity) would increase their costs considerably. Moreover, they would be unaware of the many indirect control transfers to functions caused by userland activities. We leave this investigation to future work.

Similarly, it would be interesting to explore directed fuzzing [14] and/or fuzzers specialized for certain kernel areas (Section 4.6) to reach functions/regions where static analyses report potential container confusion cases. Doing so may enable both their in-depth exploration and input generation for some reports.

The security impact of type confusion bugs depends on the memory layout of the objects involved. In an exploitation scenario, an attacker would leverage a controlled type confusion to overlap and corrupt interesting fields. On the other hand, the type confusion bugs found by our approach have no control over which types overlap. This may influence the immediate exploitability of the bugs we found and require more effort to turn a type confusion into memory corruption. However, 8 of our bugs were considered security-relevant for their exploitability and got assigned 6 CVEs (3 bugs got merged into the same CVE, as listed in Section 4.A). As a concrete illustration of security impact, we have also demonstrated a controllable out-of-bounds write on the heap for one of the CVEs reported.

4.9 Related Work

This section covers literature on type confusion, sanitization, and static analysis that the research in this chapter relates to.

Type Confusion Detection. Most existing type confusion detectors are limited to C++. UBSan [114], for instance, replaces static casts with dynamic casts in C++ to expose bugs. CaVeR [106], TypeSan [62], HexType [82], and Bitype [133] are specialized to find type confusion for C++ classes by managing runtime type metadata and performing checks on cast operations. CASTSan [123] efficiently detects type confusion leveraging C++ virtual tables, but is limited to polymorphic classes only. While all other existing approaches rely on dynamic analysis, TCD [205] uses a field-, context- and flow sensitive pointer analysis to detect type-confused C++ code.

libcrunch [88] and EffectiveSan [39] support C programs. However, both approaches rely on intercepting object allocations and binding them with their top-level allocation type. In practice, this would be hard, if not impossible, to collect in projects with the complexity of a kernel. For this reason, the typed allocator mitigation in XNU resorted to manual annotations in allocations [6]. Our approach overcomes the need of both allocation-time type inference and manual annotations.

Speculative Type Confusion. Previous work has explored speculative type confusion while dealing with objects of multiple types. Confusion in the speculative domain fundamentally differ from non-speculative one for observability and/or explainability. Kasper [83] scans the Linux kernel for arbitrary speculative gadgets. It shows how the current list iterator implementation is subject to speculative container confusion when dealing with the list heads if the terminating condition is mispredicted. Kirzner et al. [93] focus on speculative type confusion in the Linux kernel. The paper highlights possible type confusion originating from eBPF code, compiler-introduced vulnerabilities, and polymorphic types. BHI [8] leverages a speculative type confusion in eBPF code in their exploit. FPVI [141] and Spook.js [2] exploit speculative type confusion in JavaScript engines.

Other Sanitizers. Similarly to ASan [154], several sanitizers rely on redzones: Purify [67], Memcheck [155], Dr. Memory [20] and LPC [66] leverage them to detect memory corruptions in the form of spatial and temporal safety violations.

MSan [164] targets reads from uninitialized memory using a shadow map mechanism. Other sanitizers, such as Undangle [21], FreeSentry [196], DangNull [105], and DangSan [179] detect dangling pointers that cause use-after-free errors.

For boundary identification, other techniques encode tracking metadata within pointers, as with low-fat pointers [40, 103] and delta pointers [102]. For example, our approach could replace redzones with low-fat pointers on supported systems.

Static Analyzers. We conclude by mentioning a few popular static analysis tools for the Linux kernel.

Coccinelle [130] is pervasively used as a program matching and transformation tool. In addition to its use for refactoring and code hardening, it also has provisions to find intra-procedural bugs. Sparse [19] uses Linux kernel-specific annotations to perform few specialized checks. Smatch [25] followed in its footsteps to build a generic static analysis framework for several kernel bug types; it can only conduct intra-procedural dataflow analyses.

4.10 Conclusion

We presented a sanitization scheme for container confusion designed as a compiler-based runtime checker. For demonstration, we implemented the sanitizer for the Linux kernel, finding 11 bugs, which were undetected by previous work. Those bugs have often existed in the kernel for several years. Based on our results, we identified common bug patterns and used those categories to build a tailored static analyzer to discover bugs in code often unreachable by dynamic analysis. With our static analyzer, we unveiled

CVE	Description
CVE-2023-1073	Type confusion in `hid_validate_values()`, Type confusion in `bigben_probe()`, NULL pointer dereference in `betopff_init()`
CVE-2023-1074	KASLR leak in `inet_diag_msg_sctpasoc_fill()`
CVE-2023-1075	Type confusion in `tls_is_tx_ready()`
CVE-2023-1076	Incorrect UID assigned to tun/tap sockets
CVE-2023-1077	Type confusion in `pick_next_rt_entity()`
CVE-2023-1078	Heap OOB write in `rds_rm_zerocopy_callback()`

Table 4.2: CVEs assigned to the reported type confusion bugs.

78 additional, previously undiscovered bugs. We conclude that bad downcasting is not only problematic in object-oriented programming languages but also occurs in large C projects, with serious security impact.

We have disclosed and proposed possible fixes for all found bugs and relevant anti-patterns to the Linux kernel mailing list, with a total of 149 patches and 102 already merged. Some of the disclosed issues have prompted significant changes to core kernel design patterns, with fixes even requiring the kernel to transition to the modern C11 standard.

Appendix 4.A Assigned CVEs

Table 4.2 presents the list of CVE identifiers assigned to the type confusion bugs we reported.

Appendix 4.B LMbench Evaluation

Table 4.3 presents detailed results for the LMbench tests mentioned in Section 4.5.3.

Appendix 4.C Static Analysis Rules

Table 4.4 shows the definitions for our static information flow analyses. For each pattern, we report the source where the dataflow starts from, the sinks that the dataflow searches, the path filters that inhibit the report (i.e., stop path exploration) when met, and additional checks that the analysis performs at a sink before reporting a potential container confusion.

Benchmark	baseline	UNCONTAINED	KASAN	UNCONTAINED overhead	KASAN overhead
Simple syscall	1.05 μs	1.21 μs	1.93 μs	16 %	84 %
Simple read	1.28 μs	1.64 μs	2.32 μs	28 %	82 %
Simple write	1.02 μs	1.24 μs	1.83 μs	21 %	79 %
Simple stat	8.34 μs	72.10 μs	37.59 μs	764 %	351 %
Simple fstat	5.01 μs	59.24 μs	21.24 μs	1083 %	325 %
Simple open/close	18.14 μs	86.89 μs	66.97 μs	379 %	269 %
Select on 10 fd's	2.05 μs	2.41 μs	3.68 μs	18 %	80 %
Select on 100 fd's	6.29 μs	6.79 μs	9.07 μs	08 %	44 %
Select on 250 fd's	13.38 μs	14.13 μs	18.06 μs	06 %	35 %
Select on 500 fd's	25.79 μs	29.10 μs	38.73 μs	13 %	50 %
Select on 10 tcp fd's	2.19 μs	2.55 μs	3.95 μs	17 %	81 %
Select on 100 tcp fd's	11.85 μs	12.74 μs	19.37 μs	07 %	63 %
Select on 250 tcp fd's	28.23 μs	29.83 μs	45.37 μs	06 %	61 %
Select on 500 tcp fd's	56.05 μs	61.16 μs	95.02 μs	09 %	70 %
Signal handler installation	1.32 μs	1.57 μs	2.46 μs	19 %	87 %
Signal handler overhead	4.75 μs	7.65 μs	14.51 μs	61 %	206 %
Pipe latency	16.58 μs	20.99 μs	39.54 μs	27 %	139 %
AF_UNIX sock stream latency	22.71 μs	38.03 μs	74.32 μs	67 %	226 %
Process fork+exit	627.32 μs	1076.48 μs	1869.73 μs	72 %	197 %
Process fork+execve	718.54 μs	1210.79 μs	2099.22 μs	69 %	191 %
Process fork+/bin/sh -c	2530.20 μs	5370.25 μs	6756.88 μs	112 %	167 %
UDP latency using localhost	44.56 μs	135.34 μs	106.43 μs	204 %	139 %
TCP latency using localhost	56.33 μs	113.18 μs	141.90 μs	101 %	152 %
TCP/IP connection cost to localhost	240.82 μs	494.53 μs	672.52 μs	105 %	179 %
geomean				74 %	126 %

Table 4.3: LMbench experiments: comparing the native execution baseline against uncontained and KASAN.

Bug Patterns	Direction	Source	Sink	Filters	Checks
❶ Statically Incompatible Containers	B	`container_of()` input	Origin object of input pointer		Mismatch between `container_of()` destination type and origin type
❷ Empty-list Confusion (rule 1)	F	`list_entry()` result	Any use	Conditional Checks	
❷ Empty-list Confusion (rule 2)	F	`list_entry()` result	Comparison with NULL	Flows with explicit NULL values	
❸ Mismatch on Data Structure Operators	F	Any list operation (e.g. `list_add()` or `list_entry()`)	Any list operation (e.g. `list_add()` or `list_entry()`)		Mismatch between member field/type used
❹ Past-the-end Iterator	F	Any iterator variable used in a loop over a list, e.g., `list_for_each_entry()`	Any use after the loop	Checks on *found*-like variables	
❺ Containers with Contracts (backwards part)	B	Arguments of `kobject_init_and_add()`	Collect containing structure of the `kobject` and `sysfs_ops` functions		
❺ Containers with Contracts (forward part)	F	`kobj` argument of collected `sysfs_ops` functions	`container_of()`		Mismatch between collected containing structure of the `kobject` and `container_of()` destination type

Table 4.4: Details of rules for the patterns defined for static analysis. Showing the direction (B for backwards dataflow, F for forward dataflow), source and sink matched, and eventual filters and/or additional checks. ❺ employs a single rule in two parts.

5 | Conclusion

In this thesis, we uncovered multiple new classes of kernel vulnerabilities across all the major vulnerability kinds. These new vulnerability classes identify gaps in existing mitigations and bug finding methodologies for software vulnerabilities, side channels, and transient execution attacks.

To this day, finding and fixing security vulnerabilities is still a big issue in the kernel. For that reason, we aimed to address the research question of whether the kernel is sufficiently protected against new attack vectors and if fuzzing efforts for those classes of vulnerabilities still need improvement. To highlight areas where security is still lacking and remains an ongoing issue, we presented (1) a new side-channel attack against the kernel, (2) an automated scanner for transient execution attack gadgets, and (3) a new type confusion software bug pattern.

In the following, we outline our contributions for each chapter and possible future directions.

Combined side-channel attack using the TLB architecture In TagBleed, we demonstrated that isolating the address space organization of the kernel is not enough to prevent attackers from breaking randomization in the kernel. In fact, the feature to make isolating address spaces performant in modern TLBs, address space tagging, breaks the isolation guarantees. We reverse engineered the TLB architecture to infer the parts of the virtual address influencing the TLB set indexing. With that knowledge, we complemented our TLB side channel with a second side channel, using the cache activity caused by a page table walk. Both side channels individually were limited and could not reduce the randomization entry, however, combining them TagBleed was able to completely break KASLR in the Linux kernel. In conclusion, the transition to isolated address spaces will be more costly, due to optimizations causing new unintended information sharing.

Transient execution gadget scanning In Kasper, we presented a scanner for generalized transient execution gadgets in the kernel. Existing gadget scanners where either too focused on detecting specific patterns, only applied to user space or suffered from many false positives. Kasper abstracts away patterns and instead models the essential steps of an attack. Additionally, we focused on more than the traditional bounds-check bypass gadget, extending the scanner to leakage from CPU-internal buffers or port contention. Kasper finds gadgets that would have remained out of reach for existing techniques and conclude that current mitigations against transient execution attacks in the kernel are insufficient.

After publication, we received a lot of appreciation and interest from both academia and industry. A recent paper by Kogler et al. [97] used a gadget found by Kasper and combined it with Correlation Power Analysis (CPA). InSpectre Gadget [190] also explored the attack surface of Spectre gadgets in the kernel, however focused on Spectre v2 gadgets. Zomer and Sandulescu [204] explored gadget finding specifically for the Linux kernel with static analysis. Google aims to deploy Address Space Isolation (ASI) as a defense against a wide range of transient execution attacks, such as Spectre v2 [95], L1TF [188], Retbleed [191], Inception [174], however, conditional branches exploited with Spectre v1 [95] remains difficult to mitigate completely without per-gadget mitigations.

Object-oriented type confusions in C In uncontained, we presented a new sanitization scheme for an overlooked type confusion bug using a compiler-based runtime checker and a static analysis component. Many large C code bases simulate object-oriented programming using struct containerization which, however, lacks sufficient checks against potential type confusions on downcasts. With uncontained, we found dozens of such bugs, many remained undetected for several years, with serious security impact. We conclude that bad downcasting is also a problem affecting large C projects and was previously overlooked by the community. Our work resulted in discussions about introducing a new list iterator, widely used across the Linux kernel, to eradicate one of the discovered bug patterns and caused the Linux kernel to transition to the modern C11 standard since C89 does not yet allow declaring a local variable in the scope of a loop iteration. In more recent work, type++ [7] demonstrates safe downcasts in C++ but the problem in C, especially in the kernel where the memory layout of structs cannot easily change, persists.

5.1 Future directions

We have demonstrated the effectiveness of new vulnerability classes and improved on finding them in the kernel. There are several opportunities in the discovery and generalization of such vulnerability classes in future work.

Explore new combination of side channels We demonstrated the possibility to combine two entirely independent side-channel sources to explore a new attack vector in the kernel. This allows applying side-channels to new targets, turning user-space attacks into attacks against kernels or hypervisors. However, there are many more combinations of side channels possible to explore. For example, port contention or TLB sets are often used to improve covert channel signals. The information revealed by those can also be used as additional input, similar to how we used the TLB set indexing to break KASLR. Recently, SLAM [68] demonstrated that the TLB can actually be used to disclose information from the victim, rather than just breaking randomization. We believe there are many more opportunities to use the TLB architecture to minimize noise and as a covert channel for various gadget types in the kernel.

Systematic approach to break KASLR kasld [10] combines different techniques to break KASLR in a single tool and can select a working technique based on the kernel version, configuration and given architecture. While, it implements several software-based and one hardware-based KASLR break, most hardware vulnerabilities breaking KASLR are not yet implemented. Having a systematic approach to break KASLR would allow better testing of which combinations of system configuration basically render KASLR useless in the scenario of a local or even remote attacker.

Systematic approach to transient execution attacks Canella et al. [24] presented a systematic evaluation and classification of transient execution attacks. Since its publication, many new attacks were published, pressing the need for an updated classification and terminology used by the community. More importantly, some recent works don't provide source code or are not easily reproducible in the future. For instance, we worked with a proof of concept for a transient execution attack targeting a specific Ubuntu kernel version that is no longer easily available in the public repositories. Ideally, we would have a tool similar to kasld for breaking KASLR to leak sensitive memory by automatically using a suitable exploit based on the target, e.g., kernel version, hosted microarchitecture and enabled kernel mitigations. General adoption of such a tool would incentivize authors to make proof of concepts more general, allowing better reproducibility and evaluation against previous work.

Regression testing for transient execution attacks As a follow-up to the systematic approach for transient execution attacks, we are in deep desire of better regression testing. While a systematic tool would simplify regression testing already, many attacks are gadget-based and therefore dependent on the specific kernel version and even used compiler configuration. Currently, mitigations against transient execution attacks are developed after the proof of concept is sent to the vendors. The mitigations are then merged in a newer kernel version than the one used for the proof of concept and, even with back porting, proof of concepts are usually specific to an exact compilation of the kernel, making testing the mitigation with the original proof of concept quite tedious. In the past it already happened, that a defense [5] was widely deployed, only to realize later that its implementation was actually flawed. We should e.g., have test gadgets in the kernel and the hypervisor that can be enabled for regression testing to ensure the (often obscure) mitigations put in place by the vendors actually work on all affected architectures and remain working with future kernel changes.

Improve transient gadget exploitability analysis With KASPER, we demonstrated a scanner to find potential transient execution gadgets, but analyzing their exploitability remains future work. We utilized dynamic taint tracking in the kernel to evaluate over controllability from user space. While being performant, taint tracking only gives limited information about the control an attacker has over the value. To do better exploitability analysis, future work can deploy symbolic or concolic execution to gather accurate constraints of attacker controlled data. At the sink, then the constraints can be solved to get accurate information about the possible control of an attacker. Concretely, with KASPER the found gadgets can be reproduced with concolic execution to evaluate which parts of the kernel can actually be leaked by the given gadget.

Automatic transient gadget mitigation With the additional information gained from exploitability analysis, an attacker can automate further which gadgets are exploitable, but also defensive efforts can more effectively deploy (expensive) mitigations for exploitable gadgets. Previous work [126] has improved performance over the conservative mitigation of all branches, by protecting branches from gadgets in user space. We believe such effort can be further improved by applying them to crucial targets, like the kernel, and detecting specific gadget patterns to apply faster mitigation types. Currently, automated mitigations insert the expensive `LFENCE` on the branch instruction, but for example with better detecting of a bounds-check-bypass the more performant `array_index_nospec` macro can be deployed in the kernel.

A general purpose gadget fuzzer KASPER already combines searching for gadgets of various transient execution attacks, such as Spectre v1, RIDL, LVI and SMoTherSpectre. However, the framework can further be improved and expanded with other speculation types and disclosure channels. Additional speculation types could be Straight Line Speculation (STL), Branch Target Buffer (BTB) speculation for indirect branches and Return Stack Buffer (RSB) speculation for returns. For disclosure channels, the already mentioned translation-based covert channel using LAM features [68] together with the TLB architecture would be a great addition to KASPER making it an all-purpose gadget scanner. Gadget scanners for different transient execution attacks will often suffer from the same challenges to be answered; e.g., *how* controllable is the gadget or *how* to reach the gadget's code path in the target. By providing a generic approach that implements the fuzzing environment, dynamic taint analysis and concolic execution to extract the necessary constraints, the researcher instead can focus on extending the framework with the new attack primitive.

More generic type confusion scanning in C UNCONTAINED exposed new specific type confusion bug patterns in C. We focused on specific patterns since generic type confusion detection in C is extremely difficult. Often integers and different pointer types are used interchangeably making detection of invalid casts difficult without false positives. Future work however can generalize different classes of type confusions, similarly how UNCONTAINED focused on containerization bugs.

Shrink the type confusion surface in C Type confusion cannot be solved in C since the language does not forbid invalid casting and simply defines it as undefined behavior. Individual code bases can however employ policies against specific bug types to effectively eradicate them. This often requires extensive rewrites and is non-trivial to retrofit into a memory unsafe language. One such example currently adopted by some C++ code bases, are SafeBuffers [29] used to harden C++ code against buffer overflows. Similarly, Apple is deploying a hardened allocator in their XNU kernel, called `kalloc_type` [6]. It isolates allocations of a different type to e.g., render use-after-free attacks ineffective but in order to do so it requires type annotated allocations. Such type annotation would be extremely useful to protect against type confusions by knowing that memory in a specific cache is of a specific type, or by maintaining a shadow map with RTTI information to perform necessary type checks when casting. We believe in the future the Linux kernel will also adopt such typed allocation to deploy better mitigations against type confusions.

6 | Contributions to papers

TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs.
Koschel, J.: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Giuffrida, C.:** Methodology, Supervision, Writing - Original Draft, Writing - Review & Editing. **Bos, H.:** Methodology, Supervision, Writing - Original Draft, Writing - Review & Editing. **Razavi, K.:** Methodology, Supervision, Writing - Original Draft, Writing - Review & Editing.

KASPER: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel.
Koschel, J.: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Johannesmeyer, B.:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Razavi, K.:** Methodology, Supervision, Writing - Original Draft. **Bos, H.:** Methodology, Supervision, Writing - Original Draft, Writing - Review & Editing. **Giuffrida, C.:** Methodology, Supervision, Writing - Original Draft, Writing - Review & Editing.

UNCONTAINED: Uncovering Container Confusion in the Linux Kernel.
Koschel, J.: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Borrello, P.:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **D'Elia, D.:** Writing - Original Draft, Writing - Review & Editing. **Bos, H.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Methodology. **Giuffrida, C.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Methodology.

References

[1] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In *ICICS*, 2006.

[2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: attacking Chrome strict site isolation via speculative execution. In *S&P*, 2022.

[3] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. Multi-tenancy in cloud computing. In *SOSE*, 2014.

[4] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, 2016.

[5] AMD. LFENCE/JMP Mitigation Update for CVE-2017-5715. `https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1036.html`.

[6] Apple Security Research. Towards the next generation of XNU memory safety: kalloc_type. `https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/`, 2022.

[7] Nicolas Badoux, Flavio Toffalini, Yuseok Jeon, and Mathias Payer. Type++: prohibiting type confusion with inline type information. In *NDSS*, 2025.

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: on the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security*, 2022.

[9] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: silently breaking ASLR in the cloud. In *WOOT*, 2015.

[10] bcoles. Kernel Address Space Layout Derandomization (KASLD). `https://github.com/bcoles/kasld`.

[11] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M Frans Kaashoek, and Nickolai Zeldovich. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *OSDI*, pages 1139–1154, 2020.

[12] Daniel J. Bernstein. Cache-timing attacks on AES. In 2005.

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

[14] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *CCS*, 2017.

[15] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *CHES*, 2006.

[16] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: memory deduplication as an advanced exploitation vector. In *S&P*, 2016.

[17] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[18] Niel Brown. Smatch: pluggable static analysis for C. `https://lwn.net/Articles/691882/`.

[19] Niel Brown. Sparse: a look under the hood. `https://lwn.net/Articles/689907/`, 2016.

[20] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *CGO*, 2011.

[21] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[22] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *CCS*, 2019.

[23] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: break it, fix it, repeat. In *ASIACCS*, 2020.

[24] 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.

[25] Dan Carpenter. Smatch check for Spectre stuff. `https://lwn.net/Articles/752409`, 2018.

[26] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *S&P*, 2018.

[27] 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.

[28] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *CCS*, 2019.

[29] Clang. C++ Safe Buffers. `https://clang.llvm.org/docs/SafeBuffers.html`.

[30] Nick Clifton. SPECTRE Variant 1 scanning tool. `https://access.redhat.com/blogs/766093/posts/3510331`, 2018.

[31] Kees Cook. Bounded Flexible Arrays in C. `https://people.kernel.org/kees/bounded-flexible-arrays-in-c`, 2023.

[32] Jonathan Corbet. Moving the kernel to modern C. `https://lwn.net/Articles/885941/`, 2022.

[33] Jonathan Corbet. The current state of kernel page-table isolation. `https://lwn.net/Articles/741878`, 2017.

[34] Jonathan Corbet. Toward a better list iterator for the kernel. `https://lwn.net/Articles/887097/`, 2022.

[35] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: interface aware fuzzing for kernel drivers. In *CCS*, 2017.

[36] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[37] Arnaldo Carvalho De Melo. Profiling data structures. `https://lpc.events/event/16/contributions/1200/attachments/1054/2013/ProfilingDataStructures.pdf`, 2022.

[38] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *SOSP*, 1975.

[39] Gregory J. Duck and Roland H.C. Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *PLDI*, 2018.

[40] Gregory J. Duck, Roland H.C. Yap, and Lorenzo Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS*, 2017.

[41] Marco Elver and Dmitry Vyukov. Kernel Concurrency Sanitizer. `https://google.github.io/kernel-sanitizers/KCSAN.html`, 2019.

[42] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones. *TOCS*, 2014.

[43] Isaac Evans, Sam Fingeret, Julián González, 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.

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

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

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

[47] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: practical side-channel resilient kernel-space randomization. In *RAID*, 2017.

[48] Cristiano Giuffrida, Clin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: automating live update for generic server programs. In *Middleware*, 2014.

[49] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security*, 2012.

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

[51] Google. syzbot dashboard. `https://syzkaller.appspot.com`.

[52] Google. syzkaller. `https://github.com/google/syzkaller`.

[53] 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.

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

[55] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. `http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html`, 2018.

[56] grsecurity. Open Source Security Inc. Announces Respectre: The State of the Art in Spectre Defenses. `https://grsecurity.net/respectre_announce`, 2018.

[57] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security*, 2017.

[58] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *ESSoS*, 2017.

[59] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: bypassing SMAP and kernel ASLR. In *CCS*, 2016.

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

[61] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *S&P*, 2011.

[62] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. TypeSan: practical type confusion detection. In *CCS*, 2016.

[63] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. Demystifying the dependency challenge in kernel fuzzing. In *ICSE*, 2022.

[64] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 1988.

[65] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *TOPLAS*, 16(2):175–204, 1994.

[66] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *CGO*, 2012.

[67] Reed Hastings. Purify: fast detection of memory leaks and access errors. In *USENIX Winter*, 1992.

[68] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. Leaky address masking: exploiting unmasked spectre gadgets with noncanonical address translation. In *S&P*, 2024.

[69] Jann Horn. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, 2018.

[70] Jann Horn. speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2019.

[71] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[72] Nur Hussein. Randomizing structure layout. `https://lwn.net/Articles/722293/`, 2017.

[73] Intel. Indirect Branch Restricted Speculation. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html`, 2018.

[74] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, Volume 1. Order Number: 253668-060US, 2023.

[75] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 (Table 4-13). Order Number: 253668-060US, 2016.

[76] Intel. Retpoline: A Branch Target Injection Mitigation, 2018.

[77] Intel. Speculative Execution Side Channel Mitigations. `https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf`, 2018.

[78] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: a shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, 2015.

[79] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *CCS*, 2015.

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

[81] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *CCS*, 2016.

[82] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: efficient detection of type confusion errors for C++. In *CCS*, 2017.

[83] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: scanning for generalized transient execution gadgets in the Linux kernel. In *NDSS*, 2022.

[84] Ken Johnson. KVA Shadow: Mitigating Meltdown on Windows. `https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows`, 2018.

[85] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *USENIX Security*, 2004.

[86] Mateusz Jurczyk. Bochspwn Reloaded: Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking. *Black Hat USA*, 2017.

[87] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[88] Stephen Kell. Dynamically diagnosing type errors in unsafe code. In *OOPSLA*, 2016.

[89] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *ESORICS*, 1998.

[90] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: banishing the spectre of a meltdown with leakage-free speculation. In *DAC*, pages 1–6. IEEE, 2019.

[91] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[92] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: a defense against cache timing attacks in speculative execution processors. In *MICRO*, pages 974–987. IEEE, 2018.

[93] Ofek Kirzner and Adam Morrison. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security*, 2021.

[94] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html`, 2018.

[95] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[97] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. {Collide+ power}: leaking inaccessible data with software-based power side channels. In *USENIX Security*, 2023.

[98] Andrey Konovalov and Dmitry Vyukov. KernelAddressSanitizer (KASan): a fast memory error detector for the Linux kernel. *LinuxCon North America*, 2015.

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

[100] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: mitigating spectre attacks using cfi informed speculation. In *S&P*, pages 39–53. IEEE, 2020.

[101] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: breaking KASLR on the isolated kernel address space using tagged TLBs. In *Euro S&P*, 2020.

[102] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. In *EuroSys*, 2018.

[103] Albert Kwon, Udit Dhawan, Jonathan M. Smith, and Andre Knight Jr Thomas F.and DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *CCS*, 2013.

[104] Paul Larson. Testing Linux with the Linux test project. In *Ottawa Linux Symposium*, 2002.

[105] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[106] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: stopping an emerging attack vector. In *USENIX Security*, 2015.

[107] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. Hypervisor-assisted Application Checkpointing in Virtualized Environments. In *ASPLOS*, 2011.

[108] 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.

[109] 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.

[110] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst: defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[111] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[112] LLVM. Clang Static Analyzer. `https://clang-analyzer.llvm.org/`.

[113] LLVM. DataFlowSanitizer. `https://clang.llvm.org/docs/DataFlowSanitizer.html`.

[114] LLVM. UndefinedBehaviorSanitizer. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[115] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security*, 2021.

[116] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *CCS*, 2019.

[117] 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.

[118] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[119] Larry W. McVoy and Carl Staelin. Lmbench: portable tools for performance analysis. In *USENIX ATC*, 1996.

[120] Danie Micay. Comparing ASLR between mainline Linux, grsecurity and Linux-hardened. `https://gist.github.com/thestinger/b43b460cfccfade51b5a2220a0550c35#file-linux-vanilla`, 2022.

[121] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature. `https://mskb.pkisolutions.com/kb/875352`, 2006.

[122] Daniel Moghimi. Downfall: exploiting speculative data gathering. In *USENIX Security*, pages 7179–7193, 2023.

[123] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. CastSan: efficient detection of polymorphic C++ object type confusions with LLVM. In *ESORICS*, 2018.

[124] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, 1999.

[125] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[126] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX Security*, 2020.

[127] open-std. Defect Report 051. `https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_051.html`, 1993.

[128] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: practical cache attacks in javascript and their implications. In *CCS*, 2015.

[129] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006.

[130] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. *ACM SIGOPS Operating Systems Review*, 2008.

[131] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel, 2002.

[132] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: optimizing OS fuzzer seed selection with trace distillation. In *USENIX Security*, 2018.

[133] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: efficiently detecting type confusion errors. In *ACSAC*, 2018.

[134] Andrew Pardoe. Spectre mitigations in MSVC. `https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc`, 2018.

[135] Mathias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[136] Hui Peng and Mathias Payer. USBFuzz: a framework for fuzzing USB drivers by device emulation. In *USENIX Security*, 2020.

[137] Colin Percival. Cache missing for fun and profit, 2005.

[138] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for Cross-CPU attacks. In *USENIX Security*, 2016.

[139] Alexander Potapenko. Add KernelMemorySanitizer infrastructure. `https://lwn.net/Articles/878652`, 2021.

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

[141] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*, 2021.

[142] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: speculative data leaks across cores are real. In *S&P*, pages 1852–1867. IEEE, 2021.

[143] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

[144] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[145] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: systems, languages, and applications. *TISSEC*, 2012.

[146] Rough auditing tool for security (RATS). `https://code.google.com/archive/p/rough-auditing-tool-for-security/`.

[147] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *J-SAC*, 2003.

[148] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Worner, and Thorsten Holz. NYX: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security*, 2021.

[149] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security*, 2017.

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

[151] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: leakage-free transient execution. *arXiv preprint arXiv:1905.09100*, 2019.

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

[153] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: remote side channel attacks on diversified code. In *CCS*, 2014.

[154] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*, 2012.

[155] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX ATC*, 2005.

[156] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: harvesting bugs in device drivers from golden seeds. In *USENIX Security*, 2022.

[157] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN Workshops*, 2011.

[158] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: Automatic Software Self-Healing Using Rescue Points. In *ASPLOS*, 2009.

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

[160] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: an effective probing and fuzzing framework for the hardware-OS boundary. In *NDSS*, 2019.

[161] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *USENIX Security*, 2020.

[162] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B. Bobba, Sibin Mohan, and Roy H. Campbell. A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds. *arXiv preprint arXiv:1708.09538*, 2017.

[163] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned AES T-tables. In *COSADE*, 2013.

[164] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[165] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *CC*, 2016.

[166] Evan Sultanik. Two New Tools that Tame the Treachery of Files. `https://blog.trailofbits.com/2019/11/01/two-new-tools-that-tame-the-treachery-of-files`, 2019.

[167] 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.

[168] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: securing speculative execution via microcode customization. In *ASPLOS*, pages 395–410, 2019.

[169] Gil Tene. PCID is now a critical performance/security feature on x86. `https://groups.google.com/forum/m/#!topic/mechanical-sympathy/L9mHTbeQLNU`, 2018.

[170] The Linux Kernel documentation: MDS - Microarchitectural Data Sampling. `https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/mds.html`.

[171] The Linux Kernel documentation: Spectre Side Channels. `https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html`.

[172] Linus Torvalds. LKML: Page Colouring. `https://yarchive.net/comp/linux/cache_coloring.html`, 2003.

[173] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 2010.

[174] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: exposing new attack surfaces with training in transient execution. In *USENIX Security*, pages 7303–7320, 2023.

[175] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 2003.

[176] 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.

[177] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[178] Erik Van Der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: practical and complete type-safe memory reuse. In *ACSAC*, 2018.

[179] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: scalable use-after-free detection. In *EuroSys*, 2017.

[180] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: why stopping cache attacks in software is harder than you think. In *USENIX Security*, 2018.

[181] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *S&P*, 2019.

[182] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. Lightweight Memory Checkpointing. In *DSN*, 2015.

[183] Dmitry Vyukov. Syzbot and the Tale of Thousand Kernel Bugs. `https://lssna18.sched.com/event/FLYI/syzbot-and-the-tale-of-thousand-kernel-bugs-dmitry-vyukov-google`.

[184] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *TOSEM*, 2020.

[185] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. *TSE*, 2021.

[186] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[187] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *OSDI*, 2012.

[188] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[189] David A. Wheeler. Flawfinder. `https://dwheeler.com/flawfinder/`.

[190] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. Inspectre gadget: inspecting the residual attack surface of cross-privilege spectre v2. In *USENIX Security*, 2024.

[191] Johannes Wikner and Kaveh Razavi. {Retbleed}: arbitrary speculative code execution with return instructions. In *USENIX Security*, pages 3825–3842, 2022.

[192] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.

[193] 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.

[194] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: making speculative execution invisible in the cache hierarchy. In *MICRO*, pages 428–441. IEEE, 2018.

[195] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[196] Yves Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[197] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*, 2019.

[198] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

[199] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[200] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate {slh}: taking speculative load hardening to the next level. In *USENIX Security*, pages 7125–7142, 2023.

[201] 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.

[202] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *CCS*, 2016.

[203] Peter Zijlstra. Add static_call(). `https://lwn.net/Articles/824406`, 2020.

[204] Jordy Zomer and Alexandra Sandulescu. Finding Gadgets for CPU Side-Channels with Static Analysis Tools. `https://github.com/google/security-research/blob/master/pocs/cpus/spectre-gadgets/README.md`.

[205] Changwei Zou, Yulei Sui, Hua Yan, and Jingling Xue. TCD: statically detecting type confusion errors in C++ programs. In *ISSRE*, 2019.

[206] 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*, 2021.

Summary

Modern operating systems evolved into massively complex pieces of software with tens of millions lines of code. It is inevitable to have bugs in such large code bases, many of them with serious security implications. For decades, the kernel of such operating systems has been an interesting target for attackers due to its elevated privileges. Initially, attacks primarily targeted traditional software vulnerabilities like memory corruption. However, recent academic research has increasingly highlighted side-channel and transient execution vulnerabilities as well. While kernels have mitigations deployed against the most common vulnerability classes, many are too expensive for production systems. Instead, they are often used during continuous fuzzing efforts to find bugs. In recent years the amount of bugs discovered increased steadily with the improvements in bug detection during fuzzing, indicating that we are still scratching the surface and far from bug-free kernels. Additionally, state-of-the-art kernel fuzzers only focus on well-known bug classes and still find too many bugs to fix, urging the need to improve the security of our kernels.

In this thesis, we uncover new classes of kernel vulnerabilities. Within the category of side-channel vulnerabilities, we demonstrate a novel way to combine multiple side channels to overcome limitations when attacking the kernel. With our attack we demonstrate that the very same feature that makes mitigation of side channels efficient, opens up a new attack surface. For transient execution vulnerabilities, we demonstrated the first gadget scanner based on dynamic analysis for the kernel. Detecting such gadgets is often difficult without suffering from large amounts of false positives, we showed that we can yield more precise detection by facilitating dynamic taint tracking. We implemented our scanner as a sanitizer to expose transient execution to traditional fuzzing environments to rely on existing bug detection capabilities. For software vulnerabilities, we find previously undiscovered type confusion bugs which we call container confusion bugs. Such bugs can be found in many large C code bases, such as kernels, that use nested structures to implement object-orientated functionality. We designed a specialized sanitizer to detect such bug patterns with continuous fuzzing and designed static analyzers to expand our search to sections of the kernel that are difficult to reach during fuzzing.

In conclusion, we demonstrate that it is not enough to focus on currently well-established bug types and need to continue looking for new classes of vulnerabilities. We explored such new classes and improved fuzzing in all the main categories: software vulnerabilities, side channels, and transient execution attacks. Only by exploring such new exploitation angles and including them in our bug detection capabilities, we can slowly turn our kernels into a safe foundation of modern computing.