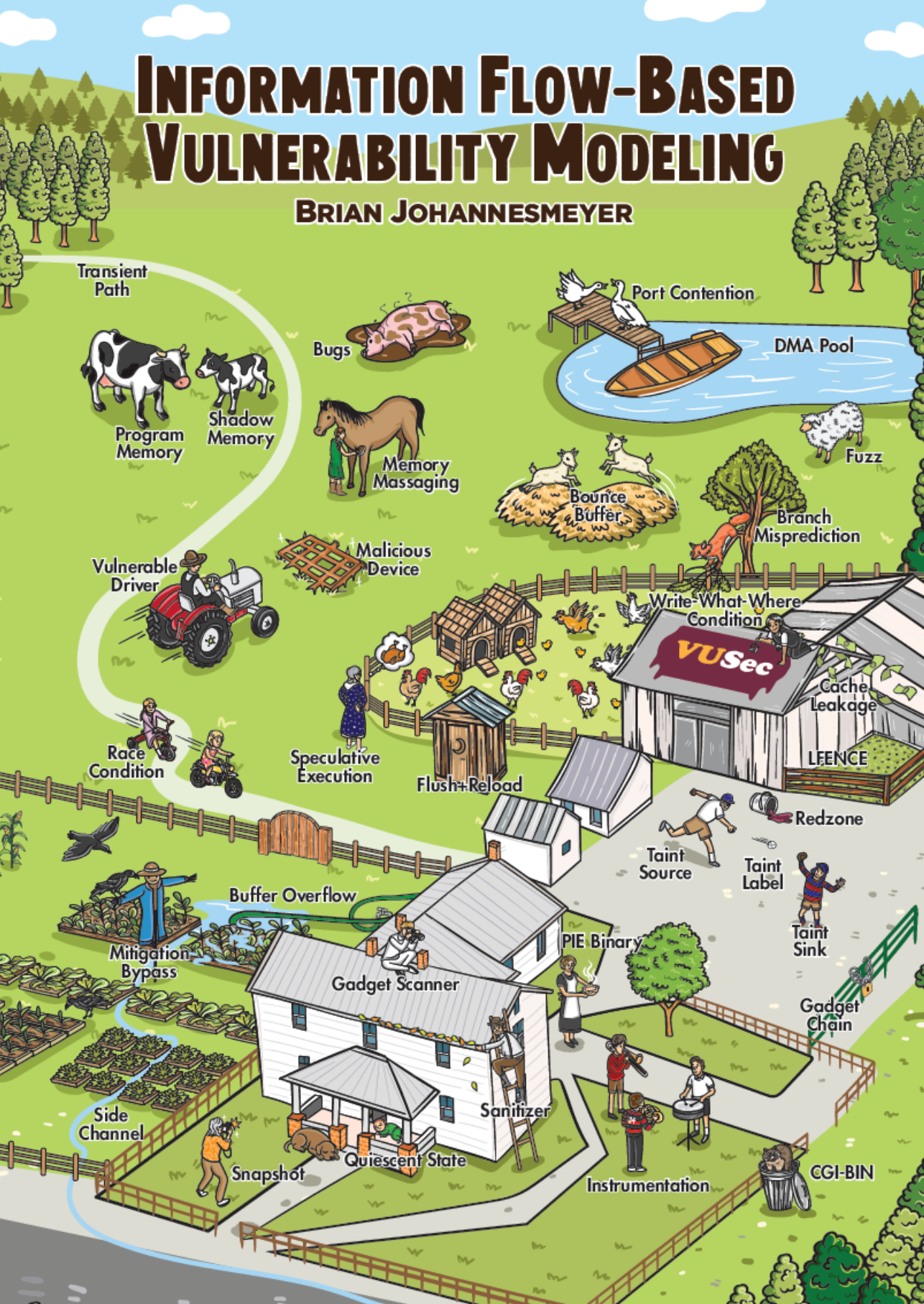

Information Flow-Based Vulnerability Modeling
Brian Johannesmeyer
Transient Path
Program Memory
Shadow Memory
Bugs
Memory Massaging
Port Contention
DMA Pool
Fuzz
Bounce Buffer
Branch Misprediction
Malicious Device
Vulnerable Driver
Write-What-Where Condition
VUSec
Cache Leakage
LFENCE
Race Condition
Speculative Execution
Flush+Reload
Redzone
Taint Source
Taint Label
Taint Sink
Buffer Overflow
Mitigation Bypass
Gadget Scanner
PIE Binary
Gadget Chain
Sanitizer
Side Channel
Snapshot
Quiescent State
Instrumentation
CGI-BIN

VRIJE UNIVERSITEIT

INFORMATION FLOW–BASED VULNERABILITY MODELING

PH.D. THESIS

Brian George Johannesmeyer

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 European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and 825377 (UNICORE).

Cover design by Aaron Yeo.

VRIJE UNIVERSITEIT

INFORMATION FLOW–BASED VULNERABILITY MODELING

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,
volgens besluit van de decaan
van de Faculteit der Bètawetenschappen
in het openbaar te verdedigen
op 12 januari 2026 om 13.45 uur
in de universiteit

door

Brian George Johannesmeyer

geboren te Bakersfield, Californië, Verenigde Staten

promotor:	prof.dr. H.J. Bos
copromotor:	prof.dr. C. Giuffrida
promotiecommissie:	prof.dr. C. Fetzer dr. T. Markettos prof.dr. K. Borgolte prof.dr. L. Davi dr. C. Gerritsen

A man of eighty, planting!
To sow at such an age might be no harm,
Argued three youngsters from a neighbouring farm,
But to plant trees! Th'old man was plainly wanting.
"For what, in Heavens name," said one of them
"Can possibly reward your pains,
Unless you live to be Methusalem?
Why tax what little of your life remains
To serve a future you will never see?"

"Is it so?" said he.

"My children's children, when my trees are grown,
Will thank me for their kindly shade.
What then, has any law forbade
A man to work for pleasure not his own?
To picture theirs is my reward today,
Perhaps tomorrow also; who shall say?"

Jean de la Fontaine, 1621–1695

Acknowledgements

This journey began during my master's at UC San Diego, in a weekly reading group. Each week, a student would select a recent publication to share and discuss with everyone. Over time, I found myself repeatedly drawn to these VUSec papers that completely blew my mind. After all, what absolute madness [81] would drive someone to combine cache side channels, page table walks, and JavaScript—three seemingly unrelated concepts? Or to even put the words GPU and Rowhammer in the same sentence [73]? Eventually, I realized that it is precisely this kind of bold, creative thinking that fuels true ingenuity[1].

Eager to pursue this kind of "mind-blowing" research myself, I moved my family across the pond, and the rest is history. In the end, I am proud to say I accomplished what I set out to do[2]—but I could never have done so without the help of some truly remarkable people.

First, to the titans that are Herbert and Cristiano: No one sparked motivation or such deep, thought-provoking discussions quite like either of you. Herbert, your reminder to "keep your eye on the ball," amid all the complexities of research prototypes, was foundational to my growth as both a researcher and a person. Cristiano, your unending stream of insights, coupled with your generosity of time and empathy for the challenges of PhD life (beginning with that two-hour chat about living in Amsterdam before I had even joined), kept me genuinely excited about our work. It was an absolute pleasure to work with both of you.

Next, I thank my PhD committee—Christof Fetzer, Theo Markettos, Kevin Borgolte, Lucas Davi, and Charlotte Gerritsen—for reviewing my thesis and providing valuable feedback that strengthened this work.

To Jakob: Mary, my wife, still jokes that during our work on KASPER, I spoke with you more frequently and more deeply than I spoke with her. But in all seriousness, your wizardry in kernel debugging[3], your drive to see it through, and your patience in disclosing vulnerabilities across the *entire* kernel tree made this monumental task both possible and fun. I could not have finished it with anyone else.

[1] "One man's insanity is another man's genius." —Joyce Carol Oates

[2] Whether I kept my sanity in the process is still up for debate.

[3] I remain convinced you used some kind of black magic to get the speculative emulation working.

To Raphael: Before you joined the work on DMARacer, the story was incoherent (no pun intended), the evaluation was haphazard, and the finish line was nowhere in sight. But our many discussions and shared determination to get something out were precisely the "drivers" (pun very much intended) we needed. I am deeply grateful to you for tackling this project with me.

To my fellow co-authors—Andreas, Asia, Erik B., and Kaveh—your invaluable insights and hard work were instrumental in making this research possible. To everyone else in VUSec: thank you for making the group not only an incredible place to work and exchange ideas, but also for bankrolling Bar Boele in the process.

Beyond VUSec, I thank the countless friends and neighbors who welcomed my family and me into the Netherlands. You showed us the warmth and *gezelligheid* of Dutch life, and we have happily exported much of that culture back home. Whether it is hauling an enormous bag of *kaasbroodjes* wherever we go, singing *Sinterklaasliedjes* (far better than most English carols, in my opinion), or deciding that *hagelslag* is perfectly acceptable at any time of day, your hospitality and traditions will stay with us forever[4]. Thank you for turning a foreign land into a second home for my family and me.

My heartfelt thanks go to my family, especially my parents, Herman and Nancy, for always supporting me, no matter how many thousands of miles away I wandered. Your unconditional love gave me the confidence to pursue my passions wholeheartedly, and I owe so much of who I am to your guidance.

To my children: Thank you for bringing a sense of wonder and anticipation to my everyday routine—knowing I would come home to your laughter (and occasional chaos!) kept me motivated and reminded me why this work truly matters. Every late-night bug fix or paper revision was made brighter by your presence in my life.

Finally, to my incredible wife, Mary: Words cannot begin to express my gratitude for your patience, love, and unwavering faith in me. You shouldered the ups and downs of PhD life alongside me—raising our children, lifting my spirits when I was discouraged, and celebrating every success, big or small. Your steady encouragement, your willingness to listen to my long-winded bug reports, and your calm, discerning perspective made it possible for me to persevere. This accomplishment is every bit yours as it is mine. You are my partner in every sense, and I simply could not have done any of this without you. Thank you for sharing this journey with me and for being the most important person in my life.

Brian George Johannesmeyer
Amsterdam, The Netherlands, January 2026

[4] I am still working on integrating clogs into my daily wardrobe, though.

Contents

List of Figures

List of Tables

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. The end of each corresponding chapter describes the contributions per author:

Brian Johannesmeyer[1], Jakob Koschel[1], Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. **KASPER: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel.** In *NDSS*, 2022.
[Appears in Chapter 2]

Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. **Practical Data-Only Attack Generation.** In *USENIX Security*, 2024.
[Appears in Chapter 3]

Brian Johannesmeyer[1], Raphael Isemann[1], Cristiano Giuffrida, and Herbert Bos. **Dynamic Detection of Vulnerable DMA Race Conditions.** In *CCS*, 2025.
[Appears in Chapter 4]

[1]Equal contribution shared first authors.

Related publications not included in the dissertation are listed in the following:

Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. **A Framework for Understanding Dynamic Anti-Analysis Defenses.** In *PPREW*, 2014.

Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. **Identifying Understanding Self-Checksumming Defenses in Software.** In *CODASPY*, 2015.

Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya Debray. **A Generic Approach to Automatic Deobfuscation of Executable Code.** In *S&P*, 2015.

Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. **Dead Store Elimination (Still) Considered Harmful.** In *USENIX Security*, 2017.

Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. **FaCT: A Flexible Constant-Time Programming Language.** In *SecDev*, 2017.

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. **FaCT: A DSL for Timing-Sensitive Computation.** In *PLDI*, 2019.

Sam Crow, Brown Farinholt, Brian Johannesmeyer, Karl Koscher, Stephen Checkoway, Stefan Savage, Aaron Schulman, Alex C. Snoeren, and Kirill Levchenko. **Triton: A Software-Reconfigurable Federated Avionics Testbed.** In *CSET*, 2019.

Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. **On the Effectiveness of Same-Domain Memory Deduplication.** In *EuroSec*, 2022.

Brian Johannesmeyer, Herbert Bos, Cristiano Giuffrida, and Asia Slowinska. **Data-Only Attacks Are Easier than You Think.** In *USENIX ;login:*, 2024.

1 Introduction

Wherever data goes, a vulnerability may arise. For example, if *untrusted data* corrupts a sensitive operation, catastrophic integrity failures can result—e.g., the Maroochy Shire sewage spill (2000) [2], the Stuxnet worm (2010) [68], and the Oldsmar water treatment plant hack (2021) [25]. Similarly, if *secret data* leaks out, large-scale confidentiality breaches ensue—e.g., the Heartbleed vulnerability (2014) [62], the Equifax data breach (2017) [244], and the SolarWinds supply chain attack (2020) [8]. In all these scenarios, the core trigger is the flow of data—untrusted or secret—throughout a system. Hence, tracking these *dangerous dataflows* becomes a natural foundation for identifying and thwarting potential attacks.

1.1 Tracking Dangerous Dataflows with Taint Analysis

The standard technique for such tracking is *taint analysis* (*information-flow tracking*), which marks ("taints") certain data as untrusted or secret at its source and then follows that taint as it propagates. If the tainted data reaches any forbidden sink, a warning is raised.

There are two main variants of taint analysis: (i) *Static Taint Analysis (STA)*, which explores all potential dataflows offline, without executing the program; and (ii) *Dynamic Taint Analysis (DTA)*, which tracks actual dataflows through a running system. STA can easily analyze large codebases, however, because it tracks *potential* dataflows through *potential* codepaths, it often raises excessive false alarms. On the other hand, because DTA tracks *real* dataflows through *real* codepaths, it tends to be more precise.

Evolution of DTA DTA techniques have evolved along with the growing pervasiveness of dangerous dataflows. In the 1970s, security mechanisms typically hinged on mandatory access controls; beyond that, data was largely "trusted". Yet, researchers

already recognized the threat of "implied" access controls, leading to the development of theoretical models such as Denning's Lattice Model [58, 59] and Fenton's Data Mark Machine [70, 71]. Although primarily academic at first, these ideas shaped how we consider confidentiality and integrity in data-centric terms.

From the 1990s–2000s, however, the situation changed dramatically: the explosion of network connectivity introduced an abundance of untrusted interfaces, from Internet-connected applications to USB-connected devices. As a result, a wave of vulnerabilities prompted the development of DTA tools, e.g., to identify buffer-overflow [149], SQL injection [86], and privacy leak vulnerabilities [64].

Modern vulnerabilities require modern solutions Despite these earlier developments, the complexity of modern exploits has outgrown many DTA solutions. Hence, current approaches in industry [175] generally either: (i) fail to capture entire multi-step vulnerabilities and instead detect only *parts* of a vulnerability—for example, AddressSanitizer focuses on memory safety bugs [187] but does not show *how* they may lead to full compromise; or (ii) forgo the precision of dynamic analysis in favor of static analysis (e.g., Coverity [96]), which is prone to *false alarms*.

Thus, a pressing question remains: *How do we generalize earlier DTA approaches to handle modern exploits?*

1.2 From Implicit to Explicit Vulnerability Modeling

Historically, DTA approaches relied on policies that modeled specific exploit techniques. This sufficed for well-defined, single-stage vulnerabilities, as it modeled the entire vulnerability *implicitly*. In particular, by simply tainting data at some monolithic source (e.g., a network read), and raising an alarm if that data appears at some monolithic sink (e.g., a return address or a SQL query), these approaches effectively modeled earlier exploits.

However, modern exploits stem from data that: (i) passes through nonstandardized interfaces (e.g., DMA), (ii) corrupts multiple operations (e.g., syscall chaining), and (iii) exploits flaws in both the hardware and software (e.g., transient execution). Hence, traditional DTA's monolithic, coarse-grained approach can lead to oversimplifications, missed vulnerabilities (false negatives), or spurious warnings (false positives).

Therefore, in this dissertation, we advocate a new perspective: *explicit vulnerability modeling*. In particular, instead of assigning a single taint policy for an entire vulnerability, we decompose vulnerabilities into their essential steps, and assign targeted taint policies for each step. This approach not only allows us to identify modern vulnerabilities, but it also yields fewer false alarms.

To illustrate this evolution, we develop three DTA tools—each targeting a different class of vulnerabilities:

1. **KASPER** models *transient execution gadgets*, which can come in all shapes and sizes—from software-based memory massaging to hardware quirks such as LVI, MDS, cache contention, and port contention. Because these gadgets do not follow a simple pattern, a single "source-to-sink" policy is insufficient. Instead, KASPER simulates branch mispredictions and applies specialized taint policies to capture the interplay between hardware and software bugs, effectively uncovering gadgets (including several novel variants).
2. **EINSTEIN** models *data-only attacks*, which may chain together otherwise benign syscalls into multi-step exploits. In this scenario, a single, monolithic taint policy would fail to capture the cross-syscall dependencies. Therefore, EINSTEIN tracks all attacker-corruptible data flowing into syscall arguments and uses custom policies to link attacker state across syscalls, automatically producing powerful data-only exploits.
3. **DMARACER** models *vulnerable DMA race conditions*, which exploit the nonstandard device-to-kernel interface to corrupt time-sensitive operations throughout the kernel. Traditional taint hooks, which are typically applied to well-defined operations, would struggle to capture such nuanced issues. Consequently, DMARACER monitors each memory access and applies taint policies tailored for each type of hardware-level concurrency bug, effectively identifying instances of this novel class of vulnerabilities.

We evaluate each prototype on large-scale, production-level targets (e.g., the Linux kernel) under realistic workloads, demonstrating the viability of explicit, multi-step modeling in practice. Our results reveal thousands of vulnerabilities and yield hundreds of working exploits, illustrating the power of DTA when guided by explicit sub-step policies.

1.3 Research Questions

This dissertation aims to answer four core research questions:

RQ1: How can modern, complex vulnerabilities be systematically modeled with DTA? Many existing approaches either: (i) rely on monolithic taint policies, and therefore fail to model modern attacks; or (ii) offload complexity to other techniques such as symbolic execution, and therefore lose out on the benefits of a pure-DTA approach. We ask how a purely DTA-focused method can directly model modern exploits without such external crutches.

RQ2: What techniques can we use to practically track data without incurring prohibitive performance overhead? Many taint engines either: (i) crash when faced with real-world workloads [198, 214], or (ii) impose limiting restrictions (e.g., sup-

porting only a few taint colors [56, 111, 199, 207]). Any combination of these issues essentially renders the analysis *practically unusable*—for example, by severely restricting coverage or inaccurately tracking dataflows—and thus we label them as "prohibitive overhead". We therefore ask which methods can remain robust enough to handle large, high-value software without imposing such prohibitive overhead.

RQ3: How does explicit modeling reduce false negatives and false positives compared to simpler, non-stepwise approaches? Prior methods often do not fully model vulnerabilities, resulting in missed exploit variants (false negatives) and false alarms (false positives). While providing a single, numerical "percent improvement" over existing tools is not always feasible—especially if no directly comparable baseline exists—we do offer a *qualitative* comparison of the vulnerabilities we capture versus prior work. Accordingly, we ask how our stepwise approach can achieve both low FN and FP rates, enabling its practical adoption in real-world systems.

RQ4: How can these approaches generalize to future exploit techniques? Every year, new vulnerabilities emerge in both hardware and software, with increasing complexity. We ask how our approach can scale to next-generation attacks rather than becoming obsolete once novel exploits arise.

1.4 Organization

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

- **Chapter 2** presents KASPER, which appeared at *NDSS 2022*.
- **Chapter 3** presents EINSTEIN, which appeared at *USENIX Security 2024*.
- **Chapter 4** presents DMARACER, which appeared at *CCS 2025*.
- **Chapter 5** formulates our contributions and discusses future directions.

2 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.

```
1 x = get_user(ptr);
2 if (x < size) {
3   y = arr1[x];
4   z = arr2[y];
5 }
```

Listing 1: Spectre-PHT BCB pattern, where attacker data bypasses an array bounds check in order to leak secret data.

2.1 Introduction

Ever since Meltdown and Spectre burst onto the scene in January 2018 [89, 118, 126], transient execution vulnerabilities have had the security community scrambling for solutions. While some of the vulnerabilities, such as Meltdown, can be mitigated fairly easily [40], 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 1)—e.g., suspicious copies from user space [22], potential out-of-bounds accesses [153], or attacker-dependent memory accesses [166]—they only approximate the presence of BCB gadgets. Our experiments show that today's state-of-the-art scanners [153, 166] yield a false positive rate of 99% (Section 2.9.1). 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

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 2.1: Essential steps in a Spectre attack.

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 2.1. 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 [29, 118], load value injection [211], cache-based covert channels [118], MDS-based covert channels [21, 184, 215], and port contention-based covert channels [15, 74]—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 [22, 115], which previous work observes is imprecise [153, 166]. 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 2.2 presents background on the attack primitives modeled by KASPER. Section 2.3 describes the threat model. Section 2.4 defines the problem scope for KASPER. Section 2.5 describes the design of KASPER at a high level. Section 2.6 explains speculative emulation, including unique implementation challenges posed by the kernel. Section 2.7 explains KASPER's vulnerability detectors and the taint policies which model their effects. Section 2.8 briefly describes implementation details. Section 2.9 evaluates the efficacy of KASPER compared to existing approaches and KASPER's gadgets found in the kernel. Section 2.10 discusses the limitations of our approach. Section 2.11 presents a case study of a gadget found by KASPER. Finally, Section 2.13 concludes.

2.2 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.

2.2.1 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.

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

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

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

Table 2.1: Overview of the possible primitives of a Spectre gadget. Kasper models the primitives in bold.

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 2.1.

Speculation type Spectre variants based on speculation from anything other than the *Pattern History Table*—that is, Spectre-*BTB* [118], Spectre-*RSB* [121, 138], and Spectre-*STL* [90]—are easily mitigated with relatively low overhead using microcode updates [99] or software updates such as retpolines [98] and static calls [242]. 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 [152]. 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*. While hardware defenses like Supervisor Mode Access Prevention (SMAP) [39] prevent the kernel from unintentionally accessing userspace data architecturally, they do not necessarily prevent transient accesses to such data. 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* [29, 30, 232, 243]—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)* [211]—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)* [169] 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 [118], 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 [235]. Second, a gadget may rely on a *microarchitectural data sampling (MDS)*-based channel [21, 184, 215], 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 [15, 74]. Finally, a gadget may rely on an *AVX*-based channel, which exploits the timing of AVX2 instructions [185].

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 hyperthreading, 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

```
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 [153] 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 [153] 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 [166] 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 [211].

Listing 2: Gadgets that differ slightly from the basic BCB pattern in Listing 1, and therefore foil existing approaches.

that PHT gadgets cannot be systemically mitigated via `lfence`, `array_index_nospec`, and user pointer sanitization, highlights the pressing need for Spectre-PHT gadget scanners.

2.2.2 Gadget Scanning

Existing gadgets scanners, however, cannot accurately identify gadgets that stray even slightly from the basic BCB pattern in Listing 1.

Static analyses Existing static analyses find gadgets through pattern-matching of source code [22, 82], pattern-matching of binaries [38], static taint analysis [220], and symbolic execution [84, 219]. Concurrent work [115]—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 2a, 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 [243], 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 Appendix 2.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 [153] 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 2b, whose *leak* instruction is in-bounds. Furthermore, it incurs false positives for any unrelated out-of-bounds accesses, such as the one in Listing 2c, 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 [166] 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 2d: even though the masking operation prevents it from accessing secret data, SpecTaint falsely reports this mitigated gadget as an exploitable gadget.

2.3 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 [208, 209], 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.

Spectre variant	Described in:	Verified signal?	Existing scanning techniques?
PHT-Arch-Cache	[118]	✓	✓
PHT-Arch-MDS	[21]	✓	-
PHT-Arch-Port	[74]	✓	-
PHT-LVI-Cache	This chapter[a]	✓	-
PHT-LVI-MDS	None[a]	-[b]	-
PHT-LVI-Port	None[a]	-	-

[a] The demonstrated LVI attack [211] exploits an LVI-Cache gadget.
[b] We verified a signal for LVI-MDS, but not for PHT-LVI-MDS.

Table 2.2: 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.

2.4 Problem Analysis

Not only do existing approaches fail to identify gadgets which stray slightly from the basic BCB pattern (Section 2.2.2), 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 2e, where an attacker can inject data via LVI [211]. 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 2.1)—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 [118] which exploited the BCB 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 [21] uses a PHT-Arch-MDS gadget and SpectreRewind [74] 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 [15] 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 [162] 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 2.2, 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.

2.5 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 2.2 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, and inline assembly all pose challenges. We explain how our approach overcomes these challenges in Section 2.6.

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 2.7.1.

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 2.1) 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 [215] covert

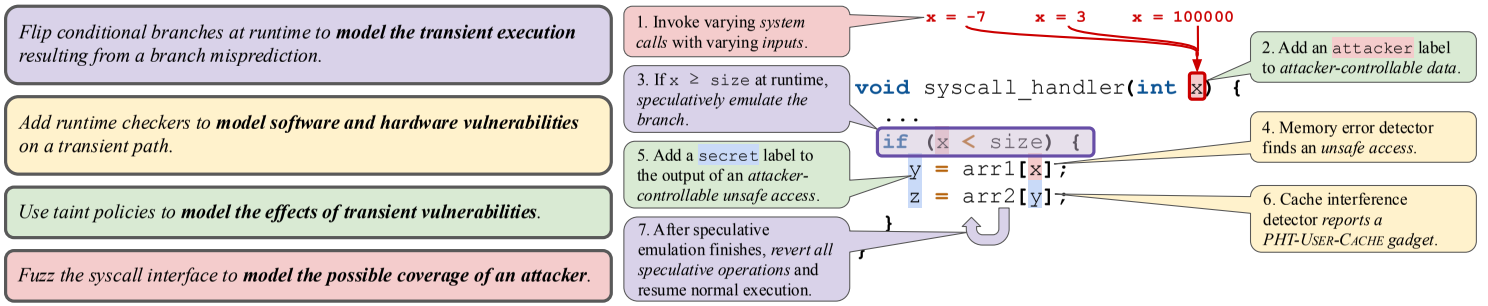


Figure 2.2: Components used by our approach and how they interact to detect a PHT-User-Cache gadget in a system call handler.

```
1   x = get_user(ptr);
2   // Sets in_checkpoint
3   if (!in_checkpoint) new_checkpoint();
4 L1:
5   if ((x < size)
6       XOR in_checkpoint) {
7     y = arr1[x];
8     z = arr2[y];
9   }
10  // Unsets in_checkpoint
11  if (in_checkpoint) {rollback(); goto L1;}
```

Listing 3: Conditional branch instrumented to enable speculative emulation.

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 2.7.2–2.7.4, including a summary of the policies in Figure 2.3.

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 2.8, including a summary of the end-to-end pipeline in Figure 2.4.

2.6 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 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 3 shows how we instrument the example of Listing 1 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`.

```
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 4: Executing past a potential page fault in speculative emulation.

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.

2.6.1 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 [153]. However, doing so in the kernel presents unique challenges which we address in Section 2.6.2.

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. 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).

```
1 static void arch_atomic64_inc(atomic64_t *v) {
2   kasper_track_store((void*)&v->counter);
3   asm volatile(LOCK_PREFIX "incq %0"
4     : "=m" (v->counter)
5     : "m" (v->counter) : "memory");
6 }
```

Listing 5: Enabling a common assembly sequence in speculative emulation.

Exception handling Exceptions do not necessarily stop speculative execution, as instructions past an exception can still be executed out-of-order [126, 215]. 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 4 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.

2.6.2 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 Appendix 2.A, 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 speculative code coverage. Listing 5 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 [193] (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).

2.7 Taint Policies

To model each of the three steps in Figure 2.1, we can draw upon previous work that uses information flow policies to detect sensitive data leakage [60, 64, 145, 173, 239], 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

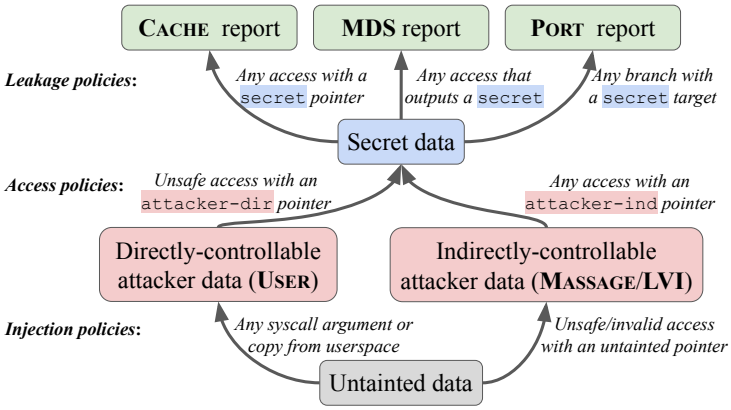


Figure 2.3: *Injection*, *access*, and *leakage* taint policies which describe how data rises from an untainted label, to an `attacker` label, to a `secret` label, and finally to a gadget report.

future for similar types of vulnerabilities, such as TLB, branch target buffer or DRAM bank side channels. With KASPER, we currently focus on the presented detectors to demonstrate its effectiveness. We summarize our policies in Figure 2.3.

2.7.1 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 [211] 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 [21]: (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 [21]. 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 [77] or TLB [130] 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 [215], SB [21], 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 [15]—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.

2.7.2 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 2.7.3). We refer to these labels as `attacker-dir` and `attacker-ind` (and `attacker` to refer to either one).

```
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 2e).

Listing 6: Gadgets that demonstrate the injection policies.

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 and stack, these policies provide the attacker with plenty of exploitation strategies [29, 30, 232]. As an example, consider the gadget in Listing 6a 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 6b, 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 `attacker-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 2.7.3), 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.

2.7.3 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 positives. 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 [153, 166] 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 7, 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.

```
1 x = get_user(ptr);
2 if (x < size) {
3   y = arr1[x & MASK]; // In-bounds access
4   z = arr2[y];
5 }
```

Listing 7: Gadget that is mitigated via a masking operation will not be falsely report as a gadget (see Listing 2d).

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 2.6). 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.

2.7.4 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 1 by propagating taint as in Listing 8a. Similarly, if `*ptr` in Listing 2a is `attacker`-controlled, the policy also identifies the gadget.

```
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 8: Gadgets that demonstrate the leakage policies.

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 2b—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 8b, 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, or indirect branch's targets is a* `secret`*—then we report it as a port contention-based covert channel.*

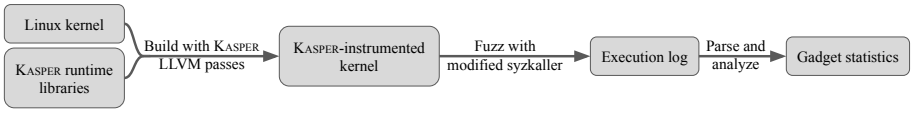


Figure 2.4: The workflow of Kasper, which takes a vulnerable Linux kernel, identifies gadgets, and finally presents statistics to aid developers in applying mitigations.

For example, consider the port contention-based gadget in Listing 8c. 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.

2.8 Implementation

Figure 2.4 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 [79] to fuzz the instrumented kernel and generate gadget reports. Finally, we calculate aggregate gadget statistics that aid developers in applying mitigations. Table 2.3 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 [129]) 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

[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 2.3: 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.

its flawed label union operation (similar to existing work [27, 202]), (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.

Taint Manager TManager implements the taint policies described in Section 2.7. 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) [120] to determine whether a memory operation is an unsafe or an invalid access.

syzkaller We used a customized version of syzkaller [79], 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 Appendix 2.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 Appendix 2.E, we present the interface that allows kernel developers to easily process the found gadgets.

		Accesses detected	*Leaks* detected
Static	MSVC [159]	–	7
	RH Scanner [38]	–	12
	oo7 [220]	–	15
	Spectector [84]	–	15
Dynamic	SpecFuzz [153]	15	0[a]
	SpecTaint [166]	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 2.4: Gadgets detected in the 15 Spectre Samples Dataset [117] by various solutions.

2.9 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).

2.9.1 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 [117]. Then, as a macro-benchmark, we evaluate KASPER and other dynamic approaches on the syscalls invoked by UNIX's `ls` command. Finally, in Appendix 2.B, we evaluate the performance of KASPER and compare it to existing approaches, where applicable.

Micro-benchmark We evaluate KASPER and previous work against Paul Kocher's 15 Spectre examples [117], 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 2.4 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 [84] on large codebases such as the Linux kernel without losing soundness.

	Total CACHE gadgets reported	FP rate	FN rate (USER-only)	FN rate
SpecFuzz [153]	662	99%	33%	60%
SpecTaint [166]	688	99%	0%	40%
KASPER (USER-only)	8	25%	0%	40%
KASPER	14	29%	0%	0%

Table 2.5: Cache gadgets reported in the kernel by various solutions when running the `ls` UNIX command.

In the dynamic analysis category, SpecFuzz [153] 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 [166] 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 [153] and SpecTaint [166] 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`-dependent speculative load as `secret` data, and report any `secret`-dependent access as a CACHE gadget.

For each solution, Table 2.5 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),

[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.

(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 2c). Third, we verified that all of SpecFuzz's (USER-only) FNs are due to its inability to identify in-bound *leaks* (as in Listing 2b). Fourth, we verified that all of SpecTaint's FPs are due to its inability to filter out safe *accesses* (as in Listing 2d). 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 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.

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 2.6: Different gadgets discovered by Kasper.

2.9.2 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 2.6. 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 2.4).

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 2.11. 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.

2.10 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 [77]), etc. Instead, like existing kernel Spectre mitigations and state-of-the-art gadget scanners [153, 166], 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 [115]. Nonetheless, in Section 2.11, we present a proof-of-concept exploit of a gadget found by KASPER. Furthermore, in Appendix 2.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 [153, 166], we inherit dynamic analysis's inherent limitation of incomplete coverage. In Appendix 2.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) [166], 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 [243]. 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

```
#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 9: 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.

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).

2.11 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 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 Appendix 2.D for an additional case study illustrating the need for a dynamic analysis tool for the Linux kernel.

2.11.1 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 9. 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.

```
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 10: PHT-Massage-MDS gadget in `find_keyring_by_name`.

2.11.2 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 10 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.

2.11.3 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 2.5. 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

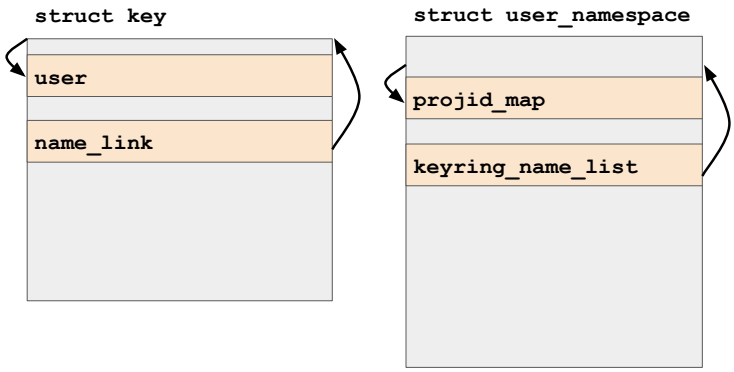


Figure 2.5: Through the type confusion `head` is assumed to be within a `key` struct instead of the `user_namespace` struct.

`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 [235] 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 [21, 184, 215] 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 11 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 list iterators, that are frequently used throughout the kernel, may cripple perfor-

```
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 11: Simplified proof of concept exploit consisting of two simultaneously executing threads.

mance 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.

2.12 Related Work

Checkpointing for transactional execution While checkpointing was used to recover from software faults [125, 216], 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 [207]. 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 [106]. Finally, a wide range of static taint tracking systems have found numerous bugs in the kernel [105, 137, 224].

Speculative gadget scanners Shortly after the publication of Spectre [118], the Red Hat Spectre V1 scanner [38] and Microsoft's Visual C/C++ Compiler [159] tried finding and mitigating Spectre V1 gadgets using well-defined code patterns. More advanced static analysis gadget scanners followed. oo7 [220] 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 kernel is impractical. SPECTECTOR [84] and KLEESPECTRE [219] use symbolic execution to find Spectre based information leaks. Without sacrificing soundness and completeness, symbolic execution becomes a bottleneck when scaling to large real-world codebases like the kernel [243]. The Linux kernel currently relies on manual analysis and static analysis tools such as smatch [22] 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 false positive rate. SpecFuzz [153], a dynamic analysis tool, combines fuzzing with the use of sanitizers to detect and mitigate potential speculative memory leaks in user programs. SpecTaint [166] uses dynamic taint analysis to track both attacker-controllable and secret data. Similar to SpecFuzz, it targets user-space programs which rarely have mitigations applied.

2.13 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 2.A Coverage Evaluation

We evaluate the completeness of KASPER's fuzzing results (Section 2.9.2) based on its coverage in both normal execution and speculative emulation.

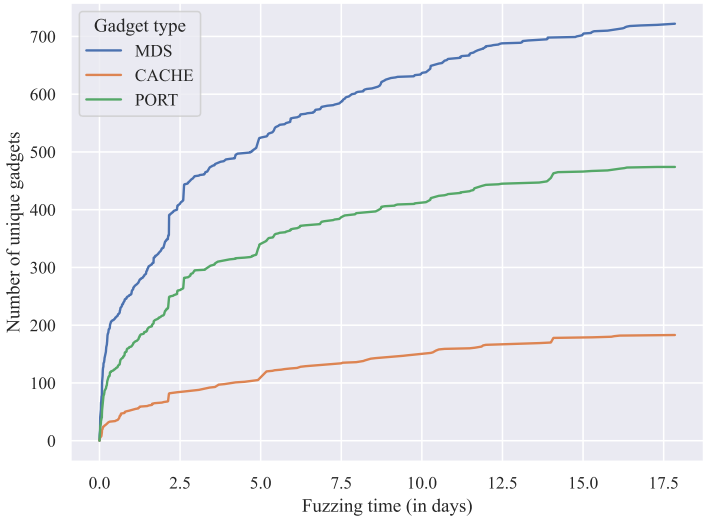


Figure 2.6: Gadgets found by Kasper over time, separated by gadget type.

Normal execution coverage First, we find that KASPER's total gadgets found (Figure 2.6) 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 2.7) 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 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.

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,

[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 [78].

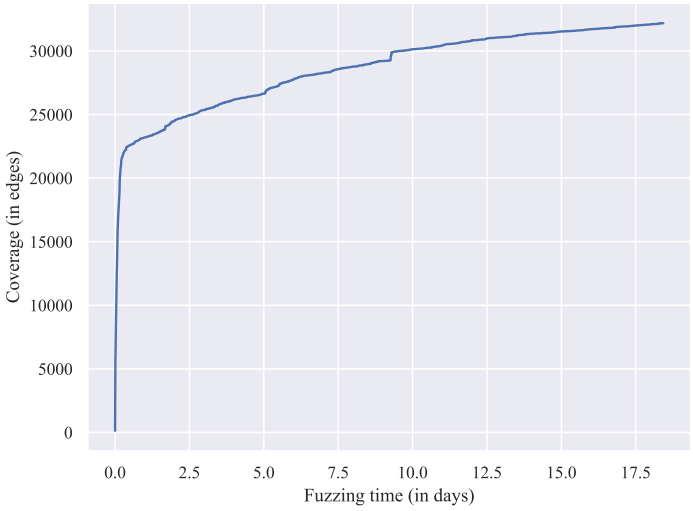


Figure 2.7: Covered edges in the Linux kernel reported by syzkaller.

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 2.6.2, 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 2.7 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 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 2.6.2. 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 [97] and what has been used in recent work (for x86 instructions) [153, 166]. 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

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 2.7: Causes for rollback in speculative emulation.

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 2.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 [153], 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 2.8), 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 2.9.2). 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 [179]. 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.

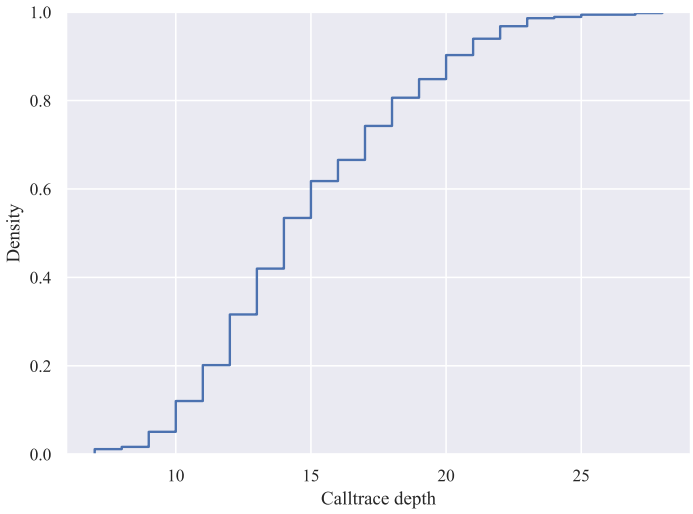


Figure 2.8: Distribution of the smallest calltrace for the found gadgets.

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.

Appendix 2.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 2.8. 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.

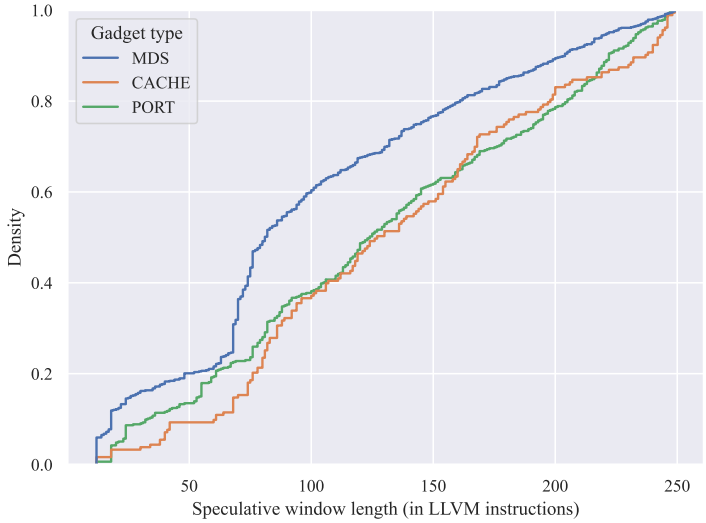


Figure 2.9: Distribution of the smallest speculative length in LLVM instructions for the found gadgets.

Required speculation window length In Figure 2.9, 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.

Appendix 2.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 12 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,

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

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 [22] may not have identified it. The dataflow from the `copy_from_user` to the (partially) mitigated gadget is only separated by a *direct* call, 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 [132], 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 12) 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 2.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 2.10 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.

Author and Contributor Statement

Conceptualization, Software, Validation, Investigation: Brian Johannesmeyer, Jakob Koschel; *Visualization*: Brian Johannesmeyer; *Methodology, Writing - Original Draft*: Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida; *Writing - Review & Editing*: Brian Johannesmeyer, Jakob Koschel, Herbert Bos, Cristiano Giuffrida; *Supervision*: Kaveh Razavi, Herbert Bos, Cristiano Giuffrida.

Figure 2.10: Screenshot of Kasper's web interface.

3 Practical Data-Only Attack Generation

As control-flow hijacking is getting harder due to increasingly sophisticated CFI solutions, recent work has instead focused on automatically building data-only attacks, typically using symbolic execution, simplifying assumptions that do not always match the attacker's goals, manual gadget chaining, or all of the above. As a result, the practical adoption of such methods is minimal. In this work, we abstract away unnecessary complexities and instead use a lightweight approach that targets the vulnerabilities that are both the most tractable for analysis, and the most promising for an attacker.

In particular, we present EINSTEIN, a data-only attack exploitation pipeline that uses dynamic taint analysis policies to: (i) scan for chains of vulnerable system calls (e.g., to execute code or corrupt the filesystem), and (ii) generate exploits for those that take unmodified attacker data as input. EINSTEIN discovers thousands of vulnerable syscalls in common server applications—well beyond the reach of existing approaches. Moreover, using `nginx` as a case study, we use EINSTEIN to generate 944 exploits, and we discuss two such exploits that bypass state-of-the-art mitigations.

Step 1: Exploit a *memory write* vulnerability.
Step 2: Steer the execution to a gadget's *entry point*.
Step 3: Execute some payload via a *gadget chain*.

(a) Steps in any non-control (or control) data attack.

Goal 1: Automatically model an arbitrary *memory write* vulnerability.
Goal 2: Automatically identify an arbitrary *gadget entry point*.
Goal 3: Automatically produce a full *gadget chain*.

(b) Goals for automating data-only attacks.

Figure 3.1: Data-only attacks: steps and goals.

3.1 Introduction

"Everything should be made as simple as possible, but not simpler."

— attributed to Albert Einstein

Since control-flow hijacking attacks came to the fore over three decades ago [197], attacks and defenses have battled over "control" of a program's control flow. Such attacks follow the general steps listed in Figure 3.1a: they overwrite certain *control data* of a program; force it to execute attacker-specified code snippets, i.e., *gadgets*; and may even chain enough of these gadgets together to achieve arbitrary computation, i.e., *Turing completeness* [18, 19, 26, 122, 174, 190]. This escalated into an arms race—between attacks exploiting some control flow, and defenses restricting that control flow—until the wide deployment of control-flow integrity (CFI) [1, 210, 212, 213, 240], which made exploitation of a program's control flow increasingly difficult.

The promise of data-only attacks In response, attackers turned to data-only attacks, which promised to exploit a program's (massive) set of data flows (i.e., any program data being used in any operation). Yet, although these attacks technically exploited *non-control data*, many of them still relied on exploiting control-*adjacent* data (e.g., branch conditions) to perform "legal" control-flow hijacking. In effect, they were still exploiting the (tiny) residual attack surface of a program's control flow. Hence, to identify data-only attacks from such a small attack surface, these approaches employed heavyweight analyses (e.g., symbolic execution) to reason about numerous complex constraints.

In reality, however, it is extremely difficult, if not impossible, to solve all constraints in bounded time for a sufficiently complex program. The problem space—i.e., overwriting *any* possible data to *any* possible value, using it in *any* potential gadget, from *every* possible program point, to achieve *arbitrary* computation—is

too large. Hence, in order to scale, these approaches are forced to make several simplifying assumptions, e.g., that: (1) an attack must only overwrite specific parts of memory, or (2) an attacker must use another exploit to reach the gadget's entry-point, or (3) an attacker must manually chain together gadgets to build an exploit. While these assumptions make exploit generation a tractable problem, they unfortunately either dramatically limit the scope of the resulting exploits, or leave parts of it to future or manual work. In short, the overarching complexity of these approaches prevents them from achieving all the goals listed in Figure 3.1b, thereby limiting their practical adoption.

A lightweight approach In this chapter, we investigate a novel lightweight technique to build data-only attacks. Our approach relies on four key insights that dramatically reduce the problem space. First, not unlike Newton's approach in the related domain of control-flow hijacking [214], we observe that dynamic taint analysis simplifies away many complex constraints for data-only attacks. In other words, rather than trying to reason about vast swathes of unknowns, we can simply reason about the concrete data used in a concrete program execution. Second, we observe that expressiveness (and especially Turing completeness) is not necessary. Rather, attackers simply want to achieve specific goals, such as running code via `execve`, or writing to a file via `write`. Third, we observe that manipulating a program's intended path is not necessary. In reality, programs will invoke `execve`, `write` or other interesting system calls all by themselves, without any additional interference from an attacker. Finally, we observe that after initialization, a program treats many types of data as immutable, and hence, that data has few (if any) constraints. By targeting these straightforward "identity" dataflows, we can corrupt data (e.g., a filename) at one program point and expect it to remain unchanged all the way to another program point (e.g., the filename getting passed to `execve`)—all the while without performing any heavyweight constraint solving. Using these insights, we make the problem tractable, even for lightweight analysis.

We present EINSTEIN, a data-only attack exploitation pipeline. EINSTEIN uses custom taint policies (rather than constraint solving) to identify attacker-controllable syscalls (rather than Turing completeness) along a program's (already valid) runtime path, generating exploits for syscall arguments that have an identity dataflow (rather than a complex dataflow) from attacker data. Given a vulnerable target program as input, it instruments the code to model and track each step of an attack, allowing it to identify gadget chains at runtime.

Specifically, we first taint any data that could be overwritten by an attacker with a unique color. Then, we track the flow of taint into security-sensitive system calls [33, 61, 93, 158, 212]. Moreover, we track the flow of taint into certain library state that is shared across system calls. If this state can be controlled by one system call, and used by another, we have evidence that an attacker can exploit it to chain

together two otherwise safe system calls in a data-only attack. Finally, we generate exploits that corrupt the identified syscall arguments with attacker data, and confirm the exploits' effectiveness by running them on the target application.

EINSTEIN's low complexity approach allows it to identify thousands of vulnerable gadgets. Taking the popular web server `nginx` as a case study, we use EINSTEIN to generate 944 exploits, including 1 CODE-EXECUTION primitive, 17 WRITE-WHAT-WHERE primitives, and 41 SEND-WHAT-WHERE primitives. Moreover, many of these exploits bypass state-of-the-art mitigations, e.g., syscall filtering [75, 76, 102, 167] and selective DFI [102, 194].

As case studies, we discuss the CODE-EXECUTION exploit, as well as one of the WRITE-WHAT-WHERE exploits. The former demonstrates the rich attack surface that data-only approaches offer, with plenty of low-hanging fruit available to an attacker. It exploits an `execve` syscall that takes its `pathname` and `argv` parameters directly from a global variable, allowing an attacker to trivially execute arbitrary code. The latter demonstrates the power of syscall chaining, which further expands the available attack surface. It exploits the `pathname` parameter of an `open` syscall and the `fd` and `buf` parameters of an unrelated `write` syscall, allowing an attacker to corrupt the server's filesystem.

Contributions We make the following contributions:

- We present a practical approach to building data-only attacks in complex programs.
- We develop EINSTEIN, an open-source[1] data-only exploitation pipeline.
- We evaluate EINSTEIN on popular server applications to identify thousands of previously unknown vulnerable syscalls and use `nginx` as a case study to generate hundreds of working exploits.

3.2 Background & Related Work

3.2.1 Data-Only Attacks & Defenses

Young and McHugh presented one of the first examples of a data-only attack in 1987 [238], and subsequent work suggested their viability in practice [43, 201, 218]. Then, Chen et al. presented the first extensive look of such attacks on real-world programs in 2005 [28].

Running example Figure 3.2 outlines one of the classic data-only attacks described in the literature [28]. In this example, the server uses the `sort-script` program to sort numbers specified by a client. In the benign case, a client first sends a `POST`

[1] EINSTEIN is available at `https://github.com/vusec/einstein`.

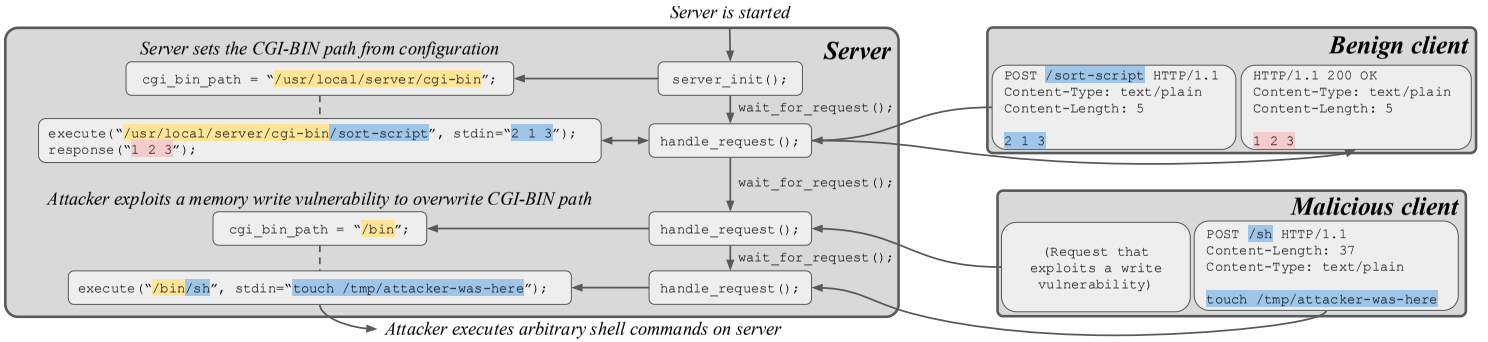


Figure 3.2: The data-only attack presented by Chen et al. [28] (with minor modifications for clarity). Despite its discovery almost two decades ago, Einstein still identifies similar gadgets that exploit the CGI-BIN path of popular web servers.

Approach		Design		Models this attack step?			Is practical?	
Name	Year	Type of dataflow analysis	Goal	Memory write vulnerability	Gadget entry	Gadget chaining	Models all primitives	Models primitive completely
FlowStitch [92]	2015	Concolic execution	Attacker-centric	○	◐	◐	✗	✗
MinDOP [94]	2016	Static taint analysis	Turing-complete	○	◐	○	✗	✗
BOPC [100]	2018	Symbolic execution	Turing-complete	●	○	◐	✗	✓
Steroids [163]	2019	Static constraint solving	Turing-complete	○	○	●	✗	✓
Limbo [182]	2020	Concolic execution	Attacker-centric	◐	◐	◐	✓	✗
EINSTEIN	-	Dynamic taint analysis	Attacker-centric	●	●	●	✓	✓

Table 3.1: Overview of approaches to building data-only attacks and how they model the steps in Figure 3.1a. Specifically, whether they: can model an attack step (●), can *partially* model an attack step (◐), or require an attack step to be defined by the user (○).

`/sort-script` request to the server with the unsorted numbers in the request body, which the server feeds to `/sort-script` in the the CGI-BIN directory. Finally, the program returns the sorted numbers to the server, and the server passes them on to the client. However, a memory write vulnerability in the Null HTTPD web server allows attackers to overwrite the CGI-BIN path, for instance by setting it to `/bin`. When the attacker now sends a `POST /sh` request to the server, it will execute the shell commands contained in the request body.

Practical vs. comprehensive defenses Mitigations for these attacks generally attempt to prevent one of two operations: (1) the *definition* (i.e., *def*) of malicious data, i.e., the memory write vulnerability; or (2) the *use* of malicious data, e.g., passing malicious data to `execve`. Unfortunately, *comprehensive* solutions to prevent the *def* of malicious data (e.g., memory safety [5, 6, 23, 103, 146, 147, 237], data space randomization (DSR) [12, 14, 20, 170]) or to prevent its *use* (e.g., dataflow integrity (DFI) [24, 128, 195]) either incur poor performance or require onerous changes in the software or hardware, thereby limiting their adoption in practice. On the other hand, *practical* measures against the *def* of malicious data (e.g., memory error scanning [4, 80, 187, 189, 199], selective DSR [156, 157]) or against its *use* (e.g., selective DFI [102, 194], syscall filtering [75, 76, 102, 167]) are non-comprehensive, because they leave a significant portion of the attack surface vulnerable to attacks. [2]

3.2.2 Automated Data-Only Attacks

Despite the discovery of data-only attacks such as the one in Figure 3.2 almost two decades ago and the lack of practical, comprehensive mitigations, automatic solutions to building attacks are still limited in a few ways: they are limited in scope, require significant manual effort, or make unrealistic assumptions. These limitations often relate to the inherent complexity and poor scalability of the concolic/symbolic execution that underlies most of these solutions, but sometimes their objectives are also different. For instance, some approaches strive for Turing completeness, which is interesting from an academic viewpoint, but not so relevant for attackers. In the remainder of this section, we break down various ways in which previous approaches fall short (see also Table 3.1).

Vulnerability-dependent analysis While preventing the use of malicious data requires a comprehensive attack surface analysis: the identification (and elimination) of *any* dangerous data flow, regardless of its origin, previous approaches [92, 94, 163] have a narrower scope, limiting themselves to specific memory write vulnerabilities rather than generic primitives. In other words, the attacks they generate are *only*

[2] We refer the interested reader to previous surveys of data-only attacks [31, 32] and memory corruption defenses [203].

applicable to the *specific vulnerability* supplied by the user, including the specific addresses and values it may write. A partial exception is Limbo [182], which models arbitrary stack overflows, although that is still a long way from general memory corruption. In contrast, EINSTEIN starts its analysis from *generic attacker capabilities* (e.g., an arbitrary read/write primitive), regardless of the specific vulnerability.

Predetermined gadget entry point Next, previous approaches may require the user to predefine the gadget entry point [100, 163]. In other words, in order to build attacks, they need the user to *already know how to exploit the program* in a way that steers its execution to the first gadget. This is the typical model of any "weird compiler" whose goal is to chain together Turing-complete gadget chains (e.g., ROP compilers [181], which assume the user can already exploit one return instruction), and it is also the model for data-only gadget compilers [100, 163]. However, identifying an entry point can be a non-trivial task, because it often requires a user to develop a separate exploit that just forces the program's execution to reach the first gadget (e.g., by "legally" diverting control flow). Previous approaches may *partially* model the entry point by either being incomplete (e.g., due to the poor coverage of concolic execution [92, 182]) or unsound (e.g., by assuming any loop executed after a memory error could be a gadget dispatcher [94]).

Manual gadget chaining Finally, previous approaches may require the user to manually chain gadgets [94]. However, *chaining is often a non-trivial task*, as CFI limits which gadgets may be chained together and gadgets may be "volatile" (i.e., selecting one gadget in a chain may rule out the selection of other gadgets in that chain [100]). Several previous approaches do not handle chaining at all, while all others do so only partially (e.g., due to concolic execution's poor coverage [92, 182], or the problem being NP-hard [100]).

3.3 Threat Model

We assume mostly the same threat model as previous work [92, 94, 100, 163, 182]. In particular, we consider a target program that: (1) has a memory corruption vulnerability [44, 47, 48, 49, 50]; and (2) is protected with DEP [9], ASLR [160], state-of-the-art control-flow hijacking defenses (e.g., CFI [1, 210, 212, 213, 240]), and strong stack protections (e.g., perfect stack canaries, shadow stacks [54]). This differs slightly from previous work [92, 94, 100, 163, 182], which do not assume strong stack protections. For simplicity, we focus specifically on popular server programs, similar to much prior work in the area [17, 133, 150, 151, 186, 212, 213, 214].

Moreover, we assume that an attacker: (1) has access to a binary equivalent of the one deployed by their prospective victim; (2) can legitimately interact with the target program (e.g., by sending requests to a target web server); (3) can bypass ASLR via an information leak [67, 81, 95, 172, 191]; (4) can exploit the memory corruption vulnerability for an arbitrary write primitive from a quiescent program state, as in previous work [151, 214]; and (5) aims to force the target program to invoke system calls with attacker-controlled parameters [33, 61, 93, 158, 212]. This differs from previous work, which assumes: (1) an attacker can carry out the arbitrary write primitive from an *arbitrary* program state [100], rather than a *quiescent* state; and (2) an attacker that aims for *Turing completeness* [94, 100, 163], rather than an attacker that aims for *security-sensitive controllable system calls* (which we introduce later, see Table 3.2).

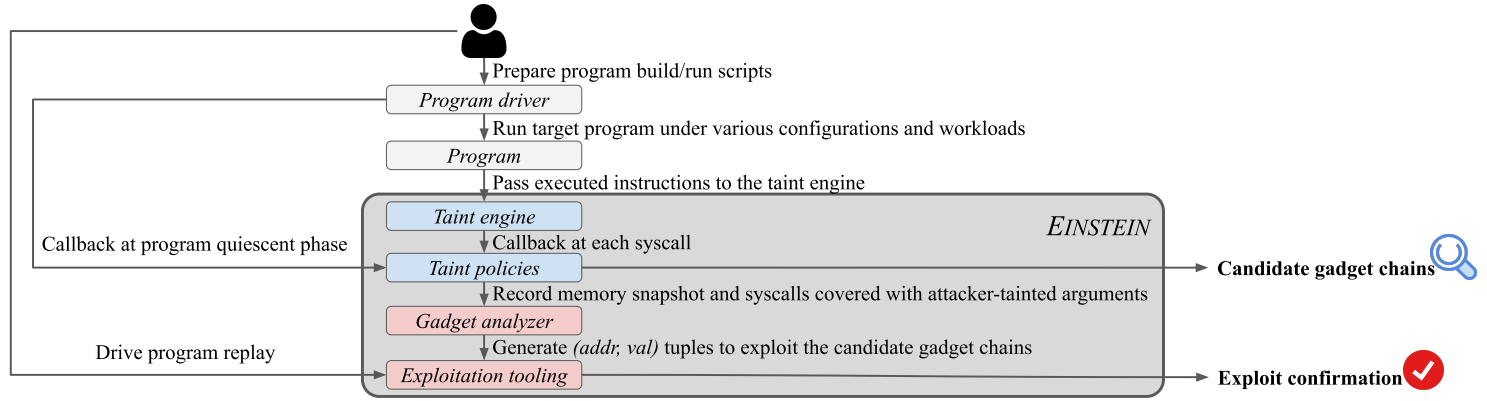
3.4 Overview of Einstein in Action

Before we discuss our design in detail, we walk through EINSTEIN's operation with our running example. We consider a target program, e.g., the web server, to have three application-dependent phases of execution: (i) an *initialization phase*, where the program starts and initializes long-lived data, e.g., configuration data; (ii) a *quiescent phase*, where the program waits for user interaction, e.g., by idly waiting for new client requests, at which point an attacker may exploit the memory write vulnerability; and (iii) a *processing phase*, where the program handles user interaction, e.g., by invoking `execve`.

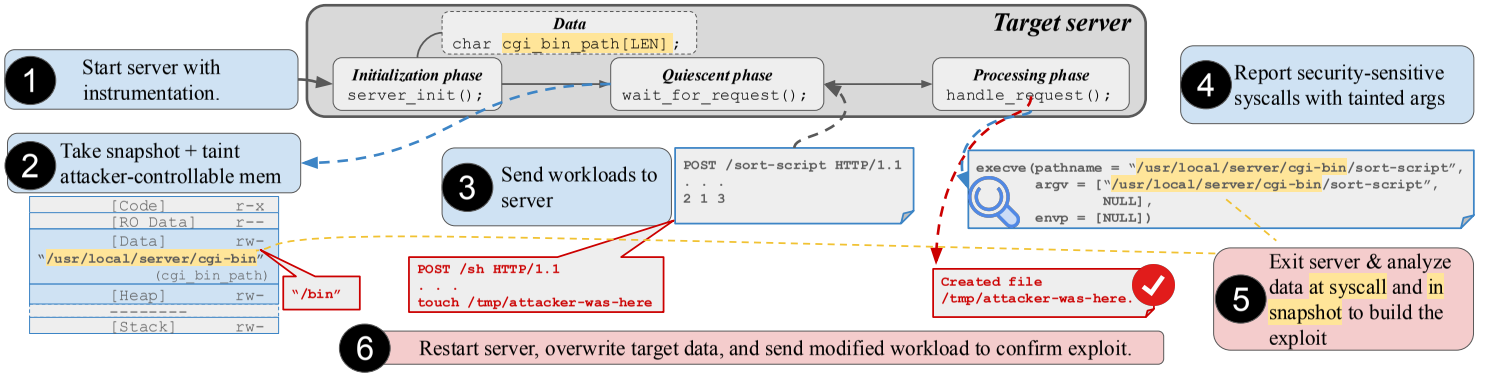
To guide our explanation of EINSTEIN's operation, we refer the reader to Figure 3.3, which shows the main components at the top and the main analysis steps at the bottom. Next, we explain how EINSTEIN identifies (Steps 1–4) and builds (Steps 5–6) the attack of Figure 3.2. We defer the more complex aspects of EINSTEIN, such as the exact taint policies and chaining, until later.

Step 1 We use a *program driver* (e.g., a test suite) to run the victim program with EINSTEIN instrumentation. As in previous work [151, 214], EINSTEIN begins its analysis with the program in a quiescent state, by which time, all initialization (e.g., of `cgi_bin_path`), has completed.

Step 2 EINSTEIN then models an arbitrary memory write vulnerability by tainting all data that could be affected by it (again, including `cgi_bin_path` string), thereby fulfilling **Goal 1** of Figure 3.1b. This is the first of our *taint policies*. Additionally, it records the tainted data in a memory snapshot.



(a) Architecture.



(b) Operation: Einstein identifies (Steps 1–4) and builds (Steps 5–6) the example attack in Figure 3.2.

Figure 3.3: Overview of Einstein.

Einstein

Step 3 To uncover gadgets, the driver sends standard requests to the server (e.g., the `POST /sort-script` request from the example), while the taint engine propagates the flow of attacker data through the target server's execution.

Step 4 As the server handles the example `POST` request, it invokes `execve` with attacker-tainted arguments. Recognizing the system call as sensitive, EINSTEIN's second *taint policy* identifies it as a *candidate gadget*, fulfilling **Goal 2**. Additionally, it records information about the syscall, such as its arguments and their taintedness.

Step 5 Having identified the candidate gadget, it now tries to build an exploit. First, EINSTEIN's *gadget analyzer* determines that parts of `execve`'s `pathname` and `argv` arguments are tainted with a color that corresponds to `cgi_bin_path`. Upon further inspection, it finds that they are in fact identical to `cgi_bin_path`. We refer to this kind of straightforward dataflow as an *identity dataflow*. EINSTEIN builds a candidate exploit by generating $(addr, val)$ pairs to overwrite the target data from `"/usr/local/server/cgi-bin"` to `"/bin"` (**Goal 3**).

Step 6 To confirm its effectiveness, EINSTEIN's *exploitation tooling* restarts the server, overwrites the target data defined by the $(addr, val)$ pairs, and sends the workload to exploit the gadget—in this case `POST /sh` with a shell command to create a file in the /tmp directory. If the file is created, EINSTEIN confirms the exploit to be successful.

For the running example, a single-blow overwrite of the arguments of a single system call is sufficient for exploitation. While simple, such cases are still common (Section 3.7). On the other hand, we will show that EINSTEIN supports system call chaining also when we discuss the complete set of EINSTEIN's taint policies. We demonstrate the usefulness of chaining in our second case study (Section 3.8).

3.5 Design

In this section, we discuss EINSTEIN's design by introducing each of the components in Figure 3.3a in some detail.

3.5.1 Taint Engine

To track attacker-controllable data at runtime, a dynamic taint analysis (DTA) engine models the flow of data from each executed instruction's inputs to its outputs. To do this, DTA engines make use of the following constructs (which may have different names depending on the engine): (1) a *color* is some kind of program metadata, which in our case, is an address that can be overwritten by the attacker; (2) a

tag (also called a *tagset*) is the set of colors corresponding to some program data (e.g., a memory location), which in our case, indicates that some data is attacker-controllable; and (3) the *tagmap* contains the tags for all program data. This is a fairly typical design of DTA [27, 104, 149, 171, 198, 214], and indeed our design is inspired by a DTA engine [214] that was used by previous work to identify control data attacks[3].

However, our use case, i.e., non-control data attacks, poses two unique challenges, which existing DTA engines cannot solve. First, because all non-control data is a potential attack vector, we need an approach that can uniquely track an *unbounded* number of taint colors over an *unbounded* amount of data. For instance, in our example attack, we need to know that, out of all the possible data that our exploit could overwrite, it should specifically overwrite the server's CGI-BIN path. Second, because programs typically invoke system calls at complex points (e.g., a time-sensitive network write, or a large file read), we need an approach that can *scale* to complex workloads covering complex features. For instance, in our example attack, we need to indeed be able to cover the codepath that handles the exploit's `POST` request. In contrast, we confirmed previous similar DTA engines cannot easily scale beyond a single default request to a server program, as the excessive slowdowns cause complex requests to time out [214]. Hence, it is around *scalability* and supporting *unbounded colors* for *unbounded data* that we design three fundamental aspects of our DTA engine, in particular: (1) how it accesses the tagmap, (2) how it initializes the tagmap, and (3) how it combines tags.

Accessing the tagmap Every time a program accesses data (e.g., via a load or store), the DTA engine accesses its tag. The DTA engine typically does this by taking the program data's address and performing some mapping to locate the corresponding tag. Some DTA engines may be *slow but memory-efficient* by navigating through multiple layers of a dynamically managed data structure (e.g., page tables) on every access [111, 112, 207, 214]. Others may be *fast but memory-inefficient* by indexing into a statically managed data structure (e.g., shadow memory) on every access [56, 199].

Our approach takes the best of both worlds by combining a *fast* shadow memory organization with a *memory-efficient* design. Specifically, we instruct the linker to constrain the user address space to the top 4 GB and reserve the lower part for shadow memory. This organization yields (i) fast shadow memory-based mappings between addresses and corresponding tags (only 2 arithmetic and 1 masking operations needed) and (ii) 64-bit user pointers easily compressible to 32 bits, since the

[3]Their approach, which was the successor to prior work named Galileo [190] (*"space"*), was named Newton [214] (*"absolute space and time"*). Since our approach generalizes the approach of Newton to non-control data attacks, we name it EINSTEIN (*"spacetime"*).

top 32 bits are always identical. Since we use pointers as taint colors, such pointer compression strategy [123] also translates to our (32-bit) colors, thereby reducing the overall size of the tagmap by 50%.

Initializing the tagmap Before accessing a tainted entry in the tagmap, the DTA engine must first initialize the tagmap. Some DTA engines may be *fast but taint limited data* by starting with an untainted (i.e., zero-initialized) tagmap [56, 112, 187, 199, 207]. Such approaches take advantage of the fast hardware MMU, because any access to an uninitialized tag page generates a page fault, which then automatically maps a zero-initialized page into the tagmap. Other DTA engines may be *slow but taint unbounded data* by starting with a mostly-tainted tagmap [198, 214]. Such approaches require a slow software-based MMU to check every access, so that when an uninitialized tag page is identified, they can map a custom-initialized page into the tagmap.

Our approach takes the best of both worlds by combining a tagmap that can taint *unbounded data* while enabling *fast* access checks by the hardware MMU. In particular, we use Linux's `userfaultfd` feature to: (1) quickly identify an uninitialized tag page via a hardware page fault, and (2) handle the page fault in software, so our DTA engine can taint the entries of the tagmap that represent attacker-controllable data.

Combining tags Every time a program combines data (e.g., via arithmetic), the DTA engine combines their tags. The DTA engine does this by performing a set union of the colors in one tag with the colors in another tag. Some DTA engines may aim to be *fast but support few colors* by representing a tag as a size-limited array, with an entry allocated for every possible color—e.g., an array containing either 1 color [111, 199, 207], 8 colors [56, 111], 256 colors [55, 112], etc. Other DTA engines may aim to be *slow but support unbounded colors* by representing a tag as an unbounded set, with an entry for each color added at runtime [198, 214].

Our approach takes the best of both worlds by using a tagset that *supports unbounded colors*, while limiting its runtime capacity in order to stay *fast*. In particular, our array-based tagset contains an entry for each color added at runtime, up to some limit N. If a tagset contains more than N colors at runtime, then we denote the tagset as *overfilled* and do not combine any more colors into it. We rely on the insight that if a tag contains numerous colors, then it likely represents complex data, with multiple sources (e.g., the output of a cryptographic hash), and hence, that data would be very challenging for an attacker to exploit than a tag with few colors. For our experiments, we select $N = 16$ as it produces few tagset overfills while maintaining an acceptable memory overhead.

3.5.2 Taint Policies

We design taint policies to model the steps of a data-only attack listed in Figure 3.1a. At a high level, we initially mark attacker-controllable regions as tainted, while EINSTEIN's DTA engine propagates the taint information to syscall callsites, and checks the tags of syscall arguments to identify those that may be exploitable.

Modeling an arbitrary memory write

We model the arbitrary memory write by tainting all memory that an attacker could overwrite. We discuss *which* data is attacker-controllable, and *what* we taint it with.

Because we consider an arbitrary memory write primitive, we taint data in *any writable memory segment*. However, since we focus on long-lived (and exclude short-lived) data corruption effected by the primitive, we *exclude the stack's segment*. (It is not a fundamental limitation, and our approach can easily accommodate tainting the long-lived data on the stack.) Moreover, to model the effect of long-lived data corruption, we taint memory when the program is in a quiescent state, as done in prior work [214].

For EINSTEIN's analysis it is vital to be able to precisely identify the origin of every tainted byte. To track memory dependencies at a byte granularity, we configure our DTA engine with a unique tag for every attacker-controllable byte: specifically, the compressed (32-bit) version of its memory address. To track the specific *value* that it originates from, we also record a memory snapshot tainting all memory. Hence, for any tainted tag, we have its origin: its $(addr, val)$ pair. The source of the taint is later a candidate location for the attacker to corrupt, control a syscall argument, and launch their attack.

Tracking dependencies

Traditional taint implementations [111] model taint strategies that track direct attacker-controlled memory dependencies (i.e., syscall argument X was read at tainted address Y), and ignore indirect ones (i.e., syscall argument X was read at address Y' using a tainted pointer). To support the latter, we additionally implemented load pointer propagation (i.e., taint every value read via a tainted pointer), allowing us to model attackers corrupting data pointers, array indexes, etc. to indirectly control syscalls arguments.

Modeling syscall exploitation

We model the next part of an attack, the execution of a gadget, with taint sinks that check whether syscall arguments are attacker-controllable. Specifically, we insert hooks before the invocation of the security-sensitive syscalls listed in Table 3.2. If we identify any attacker-tainted arguments, we generate a syscall report that

Syscall type	Chaining type	Syscalls
Security-sensitive	Does not chain	`execve, execveat, mprotect, mremap, remap_file_pages`
	Input: File FD	`mmap`
	Input: File or socket FD	`pwrite64, pwritev, pwritev2, sendfile, write, writev`
	Input: Socket FD	`sendmmsg, sendmsg, sendto`
FD-configuring	Output: File FD	`creat, open, openat, openat2`
	Output: File or socket FD	`dup, dup2, dup3`
	Output: Socket FD	`bind, connect, setsockopt, socket, socketpair`

Table 3.2: Syscalls targeted by Einstein. We target a similar set of security-sensitive syscalls as previous work [33, 61, 93, 158, 212] and a set of FD-configuring syscalls that demonstrates their feasibility. As this is not an exhaustive list of syscalls that an attacker can exploit, Einstein is easily extendable to handle more syscalls, and to target different classes of syscalls [119, 154, 217].

includes: (1) info about the syscall, such as its name, arguments, arguments' taint, and backtrace; (2) info to distinguish it from other reports, such as the target process's PID and TID, and an incrementing report number; and (3) info to help reproduce the exploit.

Modeling syscall chaining

We model gadget chaining with a combination of taint sources and taint sinks. We note, however, that chaining may not even be required for many end-to-end attacks. Namely, because our gadgets are syscalls—rather than small snippets that e.g., perform arithmetic, or conditional branching. [18, 19, 26, 100, 122, 163, 174, 190]—one gadget may suffice for an entire attack. For instance, the example attack in Figure 3.2 only exploits *one* syscall, `execve`. However, if such a syscall is unavailable to the attacker (e.g., due to mitigations), syscall chaining greatly expands the set of available gadgets.

Chaining syscalls via file descriptors Just like any other gadget compiler [100, 163, 181], we chain syscalls by chaining the data output by one gadget into the input of another gadget. The syscall interface may chain together many different types of data: process IDs (from `getpid` to `kill`), semaphore IDs (from `semget` to `semop`), and so on. EINSTEIN chains *file descriptors (FDs)*, as they are used by some of the

security-sensitive syscalls that we target. An FD uniquely identities an open *I/O stream*, e.g., a file or a network socket. If an attack controls an I/O stream, then it controls *where* the target program sends and receives data. We focus on output streams as they allow attackers to effect changes, but our approach equally applies to input streams.

Table 3.2 lists the syscalls that EINSTEIN targets and that are relevant for chaining. In particular, some security-sensitive syscalls only operate on some types of FDs (e.g., `sendto` only takes a socket FD), so they can only be chained to specific FD-creating syscalls (e.g., `socket`). On the other hand, `write` can operate on multiple types of FDs, and hence, can be chained with multiple types of FD-creating syscalls.

I/O streams can be either *directly* or *indirectly* controllable by an attacker. First, an I/O stream is *directly* controllable if the attacker controls how it was created, i.e., the arguments to an FD-creating syscall, e.g., the `filename` argument to `open`. Second, an I/O stream is *indirectly* controllable if the attacker controls an FD argument to a security-sensitive syscall and can redirect it to a directly-controllable I/O stream. For example, by controlling the `fd` argument to `write`, the attacker can redirect it to the aforementioned directly-controllable file. Because an indirectly-controllable stream depends on the existence of a directly-controllable stream, we first explain how we track directly-controllable streams, then we explain how we identify directly- and indirectly-controllable accesses.

Tracking file descriptors To identify directly-controllable output streams, we add taint sinks to FD-creating syscalls that check their arguments. First, our sinks determine whether the created stream can indeed be an *output* stream. For some syscalls, this is not necessary to check, but for others, this is specified by the arguments (e.g., `open`'s `flag` argument determines whether the file is writable). In such cases, we conclude that the syscall can create an output stream if: (1) the relevant arguments specify to create one, or (2) the relevant arguments are attacker-tainted, and hence, an attacker can coerce it into creating one. Second, our sinks determine whether an attacker can *directly control* "where" the output stream will go by checking if relevant arguments are attacker-tainted, e.g., `open`'s `pathname` or `connect`'s `addr` arguments.

Next, we need a way to track these directly-controllable streams. Namely, the kernel maintains an FD table, which maps FDs to their corresponding I/O streams. If attackers control a created stream, then they control its entry in the FD table. Hence, any future references to an attacker-controllable stream should be treated as tainted—regardless of whether the FD is tainted. To model this, we maintain our own *controllable FD table*, which mirrors the kernel's FD table, but with a few differences. Specifically, our table:einstein: (1) only creates entries for controllable

output streams, (2) maps FDs to the report number of the syscall that created it, and (3) is consulted to taint any subsequent FD syscall argument that may refer it (whether directly or indirectly).

Identifying accesses to controllable file descriptors The final question we reach in our design is: How do we identify accesses to attacker-controllable I/O streams? The naive approach would be to use the same taint sinks described in Section 3.5.2 to simply check whether FD arguments to security-sensitive syscalls are tainted. However, this is imprecise because it would: (1) fail to identify many controllable accesses, e.g., if the FD argument is *untainted*, but its I/O stream is *tainted*; and (2) falsely identify many non-controllable accesses as controllable, e.g., if the FD argument is *tainted*, but all open I/O streams are *untainted*. Hence, we modify our taint sinks for FD arguments to instead check our *controllable FD table*. In particular, we determine that a syscall's FD argument is: (1) directly-controllable if it exists in our controllable FD table; or (2) indirectly-controllable if it is tainted and our controllable FD table contains an FD of the same type (e.g., a socket FD, if the syscall requires a socket FD).

3.5.3 Gadget Analyzer

To build exploits for the identified gadget chains, one could employ a heavyweight analysis that determines the exact constraints of all attacker dataflows e.g., via symbolic execution. However, as EINSTEIN aims to identify low-effort attacks, our gadget analyzer takes a lightweight approach that targets *identity dataflows*, i.e., dataflows where the value at the sink is identical to the value at the source. As we show in Section 3.7.1, this already yields a rich attack surface.

In our case, a *sink value* is a tainted syscall argument, and a *source value* is the data it originates from in the memory snapshot. We can map *sink values* to their corresponding *source values* by checking the *sink value*'s taint color, which contains its *source address* in the snapshot. In other words, we use the taint color of each reported syscall argument to identify the value it originates from in the memory snapshot. If these values are equivalent, then the argument has an identity flow.

Then, to exploit the identity flows, we generate $(addr, val)$ pairs to overwrite some *source value*. The specific exploit pairs that we may generate depend on a particular attacker's goals and the particular target program. Hence, for simplicity, we generate generic per-syscall exploits, e.g., with `execve` creating a particular file, or `write` writing a particular string. If the user wishes, the exploits may be modified from our exploitation tooling.

3.5.4 Exploitation Tooling

Because our gadget analyzer forgoes complex constraint solving, we consider the set of $(addr, val)$ pairs that it generates to only make up a *candidate* exploit. In order to confirm it as a *working* exploit, we must confirm that our exploit indeed stems from an identify data flow not violating any program invariants. Hence, rather than e.g., symbolically verifying all possible constraints that our exploit may effect, we punt the problem to the program itself—i.e., to a real, concrete execution. In particular, our exploitation tooling simply re-runs the target program and confirms whether the exploit is indeed successful. It does this by: (1) rewriting the data specified by the $(addr, val)$ pairs from the arbitrary memory write primitive; then (2) sending some workload to the target program to trigger the gadget chain; and finally (3) checking whether the exploit achieved its desired outcome. If the exploit is successful, then we confirm it to indeed be a *working* exploit.

We observe that some candidate exploits have a better chance of being successfully confirmed than others. Fortunately, because our approach casts such a wide net—e.g., by tracking *any possible* attacker data to *any combination* of security-sensitive syscalls—we have plenty of candidate exploits at our disposal. We prioritize those gadgets that we consider to be more robust and stronger attack primitives, e.g., gadgets that (1) exploit higher-value syscalls (e.g., `execve`), (2) exploit multiple syscall arguments, (3) overwrite deterministic addresses (e.g., global data), and (4) exploit syscall arguments with larger identity flows (for our experiments, we target identity flows that span at least 4 bytes).

3.6 Implementation

We base the implementation of our DTA engine on libdft [111]. Like other analyses that use libdft, the runtime component of EINSTEIN (which specifies the taint policies) is statically linked with our libdft variant. We invoke the target program with our tool enabled and with ASLR disabled. Although our threat model already assumes an attacker that bypasses ASLR, disabling ASLR for the analysis does provide a couple of implementation-level benefits: (1) it allows us to enforce 32-bit addressing, so that we can compress our tags from 64 bits to 32 bits; and (2) it causes deterministic addresses to stay constant across multiple runs, which, while not important for an attacker armed with an information leak, it does simplify the confirmation of candidate exploits that target such addresses. We expand upon other parts of our implementation in Appendix 3.A.

Syscall*	Total covered	Total covered where this argument has a dataflow from attacker data and the percentage of those that are identity dataflows.					
		Arg. 1	Arg. 2	Arg. 3	Arg. 4	Arg. 5	Arg. 6
`execve`	11	7 (86%)	7 (86%)	7 (86%)			
`mmap`	787	55 (5%)	441 (8%)	55	16 (100%)	17	73
`mprotect`	144	97 (68%)	98 (27%)	1			
`mremap`	11	11	7	11	-	-	
`pwrite64`	1270	1217 (3%)	741 (47%)	399 (19%)	506 (36%)		
`pwritev`	10	10 (100%)	10 (10%)	-	10 (20%)		
`sendfile`	1	-	-	1	1		
`sendmmsg`	2	2	2	-	-		
`sendmsg`	12	5 (100%)	3 (100%)	-			
`sendto`	431	389 (7%)	410 (38%)	408 (6%)	-	-	-
`write`	3083	768 (74%)	2877 (91%)	1265 (10%)			
`writev`	754	244 (18%)	742 (76%)	-			

* Did not cover `execveat`, `pwritev2`, or `remap_file_pages`.

Table 3.3: Number of gadgets found per syscall and argument type.

Target program	*Non-control data* gadgets (% identity dataflow)	*Control data* gadgets* (% identity dataflow)		Runtime (h:mm:ss)		Peak PSS (MB)			Code coverage
		With register operand	With memory operand	Baseline	EINSTEIN	Baseline	EINSTEIN (1 color/tagset)	EINSTEIN (16 colors/tagset)	
`httpd`	1834 (97%)	2082 (42%)	15,179 (94%)	0:04:00	0:35:08	26	1560	3129	27.3%
`lighttpd`	92 (98%)	50 (34%)	155 (94%)	0:00:05	0:01:51	3	176	257	27.8%
`nginx`	1623 (82%)	503 (60%)	564 (99%)	0:08:41	3:27:05	24	479	848	49.1%
`postgres`	2105 (27%)	1382 (11%)	4690 (13%)	0:00:56	1:47:27	175	1568	11,621	46.5%
`redis`	218 (84%)	160 (22%)	53 (100%)	0:04:01	1:01:13	191	3257	32,188	33.6%

* Mitigated by code reuse defenses.

Table 3.4: Number of gadgets found and performance per target program.

3.7 Evaluation

We evaluate EINSTEIN in terms of attack surface and performance on an AMD Ryzen 9 3950X CPU with 128GB of RAM running Ubuntu 22.04.3 LTS (kernel v6.2). To run Newton [214], we use an Intel Xeon Silver 4108 CPU with 32GB of RAM running Ubuntu 16.04.7 LTS (kernel v4.3).

Programs and drivers We target the web servers `httpd`, `lighttpd`, and `nginx`; and database servers `postgres` and `redis`—all of which have been shown to be at risk of memory write vulnerabilities [44, 45, 47, 48, 49, 50]. Although we follow previous work by targeting server applications [17, 133, 150, 151, 186, 212, 213, 214], we note that even programs with more limited user-input interaction and few obviously dangerous syscalls (like `execve`), may well be at risk. Because EINSTEIN's approach only builds on a standard write vulnerability, it can indeed generalize to such applications.

We use each server's test suite to drive the analysis, since test suites are designed to test a variety of features and therefore cover a diverse set of code paths. If EINSTEIN is able to craft exploits based on simple test suites, it validates our claim that our approach greatly simplifies data-only attacks. Moreover, because `nginx` is a common target for exploitation case studies [144, 214, 225], we use our exploitation tooling to confirm the candidate exploits that EINSTEIN generates for `nginx`.

Program phase determination As explained in Section 3.4, the phases of program execution (i.e., the initialization, quiescent, and processing phases) are application-dependent. Although more sophisticated approaches can determine the different phases of execution for a program (e.g., syscall monitoring [76]), all of our target servers initialize in a couple of seconds. Thus, we simply conservatively wait 10 seconds before establishing post-initialization quiescence.

3.7.1 Attack Surface Analysis

We investigate the attack surface uncovered by our approach based on: how effectively an attacker can control security-sensitive syscalls, and what an attacker can do with those syscalls. In Appendix 3.B, we also evaluate how this attack surface compares with the attacks surface of control-flow hijacking attacks.

As in previous work [139], we present the number of syscalls in terms of *unique backtraces* since other possible metrics are problematic: presenting the *total executed syscalls* would artificially inflate the numbers (because the longer a program runs, the more syscalls it executes); while presenting *unique callsites* would artificially deflate the numbers (because a single syscall site may be called from many parts of a program).

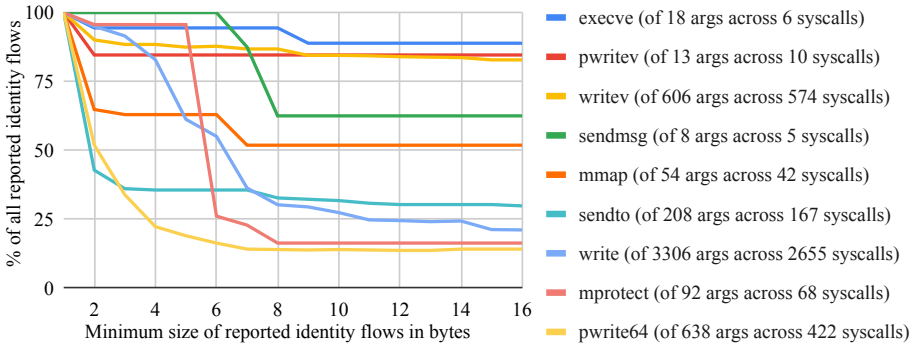


Figure 3.4: Percentage of all identity flows reported versus the minimum size of reported identity flows in bytes.

Controllability To evaluate the degree to which an attacker can control syscall arguments, we first refer to Table 3.3, which presents the number of tainted syscalls arguments, and in particular, the percentage of those with an identity flow. The first observation we make is that *most arguments have an identity flow*. Unsurprisingly, many of the identity flows are for data types that are generally treated as "immutable", e.g., the strings used in: the `pathname` and `argv` arguments to `execve` (86%); and the `buf` argument to `write` (91%) and `writev` (76%). Similarly, FD arguments tend to have a high rate of identity flow (e.g., 100% for `pwritev`, 100% for `sendmsg`, and 74% for `write`); this is also unsurprisingly because these FDs are often configured by syscalls that also take string-based arguments, e.g., the `pathname` to `open` and `creat`, and the `addr` to `connect` (which may be a string depending on its `sa_family`).

The second observation we make is based on Figure 3.4, which presents the lengths of the identity flows, broken down by syscall. We observe that many of the identity flows span at least 16 bytes, and hence, much of the data that an attacker can corrupt *remains unchanged* all the way to the vulnerable syscall. Given the prevalence of attacker data remaining unchanged to so many syscall arguments, we conclude that our data-only attacks are *low-effort*.

Syscall-based exploitation To evaluate what an attacker can do with these controllable syscalls, we once again refer to Table 3.3 and observe that *every argument can be attacker-tainted*. Even for syscall arguments that we do not expect to be attacker-tainted (e.g., `mprotect`'s `prot` and `sendmsg`'s `flags` arguments, which are typically compiled down into a constant), we still encounter cases of tainted arguments. For example, we identify a case where `pthread_create` calls `mprotect` with a `prot` argument that has an identity flow from attacker-controllable data on the heap.

Attack primitive	Count
CODE-EXECUTION	1
WRITE-WHAT-WHERE	17
WRITE-WHAT	375
WRITE-WHERE	79
SEND-WHAT-WHERE	41
SEND-WHAT	372
SEND-WHERE	59
Total	944

Table 3.5: Confirmed exploits for `nginx`.

Next, we refer to Table 3.5, which presents our confirmed exploits for `nginx`, and observe that they offer *many primitives to an attacker*. For example, a vulnerable `execve` gives us a CODE-EXECUTION primitive; vulnerable file-configuring syscalls (e.g., `openat`) combined with vulnerable file-write syscalls (e.g., `write`) give us 17 WRITE-WHAT-WHERE primitives; vulnerable socket-configuring syscalls (e.g., `connect`) combined with vulnerable socket-write syscalls (e.g., `sendmsg`) give us 41 SEND-WHAT-WHERE primitives; and vulnerable syscalls combined with non-vulnerable syscalls give us less powerful, but still more numerous primitives (i.e., 885 total WRITE-WHAT, WRITE-WHERE, SEND-WHAT, and SEND-WHERE primitives). In comparison, FlowStitch [92]—a previous approach that also (i) generates syscall-based exploits, and (ii) assumes an arbitrary memory write primitive in its evaluation of `nginx` [45]—is hampered by concolic execution's poor coverage, and as a result, only generates 2 exploits for `nginx`. Because EINSTEIN identifies such a broad range of tainted arguments, and generates a variety attack primitives, we conclude that our data-only attacks are *diverse*.

3.7.2 Performance Analysis

We investigate the performance of EINSTEIN along two dimensions. First, we investigate the overall performance of EINSTEIN per target program, as shown in Table 3.4. The baseline that we compare to is the vanilla target program, i.e., without any instrumentation.

Second, we investigate the performance of EINSTEIN's DTA engine. To do this, we compare the overhead of EINSTEIN to the overhead of Newton [214], which, in contrast to EINSTEIN, does not have the DTA engine optimizations described in Section 3.5.1. Specifically, its DTA engine: (1) accesses its tagmap via a *page table walk*, (2) initializes its tagmap via a *software-based MMU check*, and (3) represents tags as *unbounded sets*. Figure 3.5 presents the overheads of Newton, EINSTEIN, and their baselines from running `nginx` under a minimal configuration and processing

multiple HTTP requests. We note two issues that led to this evaluation setup: (1) any moderately complex workload (e.g., running with a configuration that loads more than a few modules, or processing an HTTPS request) caused Newton to immediately crash, so we only run with a minimal configuration and process simple requests; and (2) for a variety of technical reasons (e.g., Newton requiring an older kernel version, full 32-bit support, etc.) we could not evaluate EINSTEIN and Newton on the same machine, so we present the baseline overheads on both machines; T1 is EINSTEIN's machine, and T2 is Newton's machine.

Runtime overhead As we see in Table 3.4, the runtime overhead of EINSTEIN ranges from 8x–26x. Our extensive experiments, including full test suites, confirm that the DTA engine easily scales to a complete testing campaign. Moreover, we see in Figure 3.5a that EINSTEIN handles `GET` requests at least two orders of magnitude faster than Newton. We attribute EINSTEIN's speedup primarily to the way it accesses the tagmap: on every access, it performs a fast shadow memory lookup, whereas Newton navigates its page table structure. Because our DTA engine optimizations enables a practical analysis platform, we consider the runtime overhead *acceptable*.

Memory overhead Next, we evaluate the memory overhead of EINSTEIN when using both 1 color per tagset and when using 16 colors per tagset (i.e., the default). With 1 color/tagset, tags are easily prone to overfill, because adding any more than 1 color to a tagset results in it no longer tracking individual colors. Hence, this configuration cannot build candidate exploits for many gadget chains, because many syscall arguments' tags are overfilled. Nonetheless, it still identifies all the candidate gadget chains that the 16 colors/tagset configuration identifies.

We find in Table 3.4 that the memory overhead of EINSTEIN with 1 color/tagset is 9–60x relative the uninstrumented baseline. This overhead accounts for the internal data structures of EINSTEIN, `libdft64-ng`, etc.—most significant of which is the tagmap, which contains a tagset for every byte of program memory. If we increase the number of colors/tagset from 1 to 16, we would expect the resulting memory overhead to be 16x in the worst case. However, we find that our target databases have a 7.4–9.9x overhead, and our target web servers only have a 1.6–2.0x overhead. We attribute worse overheads to target applications having more spatially fine-grained access patterns, because they need to page in more of the tagmap per page of program memory.

Next, we see in Figure 3.5b that EINSTEIN has a constant memory overhead, whereas Newton's memory overhead grows for each additional `GET` request, until it eventually crashes from reaching the 2 GB limit for 32-bit processes. We attribute this difference to the way the DTA engines represent tags: whereas EINSTEIN uses *bounded* sets of 16 tags/set, Newton uses *unbounded* sets. As a result, Newton will combine any new colors with a tag until it runs out of memory. In principle, this

has a worst case $O(n^2)$ memory overhead (where n is the total program memory), if every byte of program data is combined with every other byte of program data. Hence, an application such as `nginx`, which requires about 16 MB to run normally, would in the worst case, require 256 TB to run with Newton. In reality, however, the 2 GB limit masks this overhead, and it simply crashes well before this limit is reached. We conclude that because of our DTA engine optimizations and because of its configurable number of colors per tagset (i.e., to accommodate the tradeoff between total available memory and number of exploits generated), our memory overhead is *acceptable*.

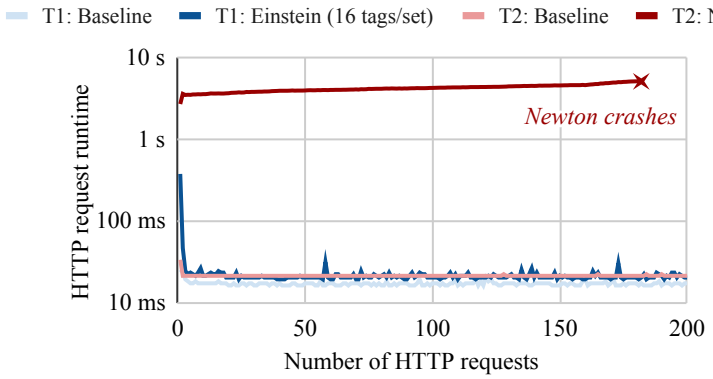
Coverage As with any dynamic analysis, its efficacy is dependent on program coverage: poor coverage will generally yield poor results, and vice-versa. Yet, even though the test suites only covered 27–49% of the programs' code, EINSTEIN still uncovered many gadgets. Future work to increase code coverage, e.g., by exploiting syscall-guard variables [236], would undoubtedly *yield more gadgets*.

3.8 Case Studies

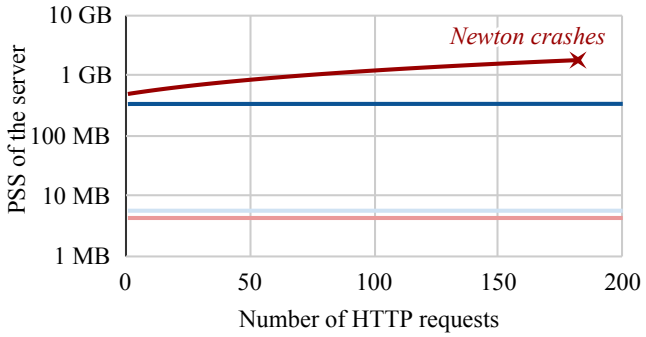
We have demonstrated that EINSTEIN builds diverse, low-effort attacks for popular servers. To demonstrate that such attacks pose a serious threat, we present two case studies of attacks against `nginx`. For each case study, we will: (1) explain how EINSTEIN identifies the gadget chain, builds a candidate exploit for it, and confirms the exploit's effectiveness; and (2) discuss how the exploit bypasses state-of-the-art defenses.

Defenses targeted In considering which defenses to center our discussion around, we recall our explanation in Section 3.2 about the *practical* and *comprehensive* defenses against vulnerable *defs* and *uses*. Because we assume the existence of a memory write vulnerability, any *def*-centric defenses (e.g., memory safety, DSR, and memory error scanning) are implicitly out of scope. Moreover, because the *comprehensive* defenses (e.g., memory safety, full DSR, and full DFI) are not widely deployed, we also consider them out of scope. Hence, we center our discussion around the *practical use*-centric defenses, i.e., syscall filtering [75, 76, 102, 167] and selective DFI [102, 194]. As we demonstrate that such *practical* defenses are fundamentally unsecure, we conclude that more *comprehensive* defenses should be deployed. [4]

[4] Indeed, developers may deploy the comprehensive (but costly) defenses to thwart EINSTEIN's attacks. Moreover, they may use EINSTEIN's reports to harden their program (e.g., by sanitizing particular syscall arguments).



(a) Runtime of a request versus the number of requests.



(b) PSS of the server versus the number of requests.

Figure 3.5: Comparison of running `nginx` with Einstein and Newton [214] to running `nginx` as is (i.e., the baseline).

3.8.1 Bypassing Syscall Filtering

This case study exploits an `execve` gadget in `nginx`'s live binary upgrade feature, wherein `nginx` replaces its own running binary while maintaining all of its connections. Figure 3.6 walks through how we identify and exploit the gadget with EINSTEIN.

Identification and exploitation First, when starting up, `nginx` saves its `argv` to the global variable `ngx_argv`, which is subsequently tainted by EINSTEIN. Next, when sending workloads to `nginx`, EINSTEIN sends the `SIGUSR2` signal to the `nginx` master process, which initiates the live binary upgrade. This raises the `ngx_change_binary` condition, which causes `nginx` to call `ngx_exec_new_binary` with `ngx_argv` as an argument. Eventually, this results in `nginx` calling `execve`, passing to it the same `argv` that it had initially saved during start up. EINSTEIN identifies the attacker-tainted argument to `execve` and reports it.

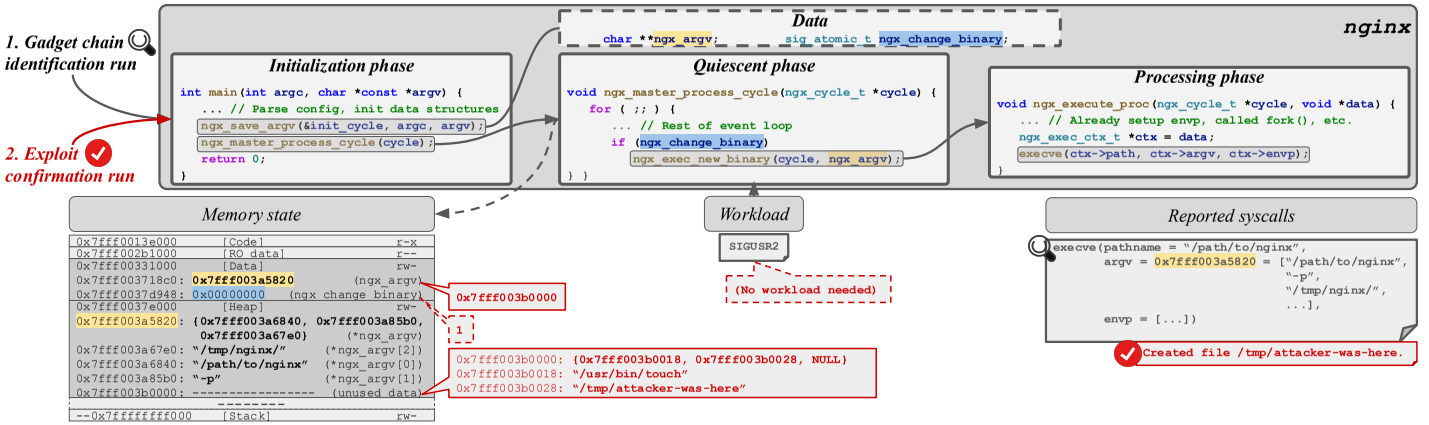


Figure 3.6: An `execve`-based gadget in `nginx` that Einstein identified and built an attack for.

Einstein

EINSTEIN builds an exploit for this by first identifying the identity dataflow from `execve`'s `argv` argument to the `ngx_argv` global variable. Then, EINSTEIN builds an "arbitrary code execution" candidate exploit by generating $(addr, val)$ pairs that overwrite `ngx_argv` to point to an array of two strings: `"/usr/bin/touch"`, `"/tmp/attacker-was-here"`, and a `NULL` terminator. After re-running `nginx` to confirm this exploit, it indeed creates the file `/tmp/attacker-was-here`, indicating that our exploit was successful.

Because this gadget is triggered server-side (via `SIGUSR2`) rather than client-side (via a network request), it would appear at first glance that a remote attacker cannot trigger it—rather, the attacker would have to wait until the `nginx` process upgrades itself. However, our exploitation tooling (which parses the gadget's backtrace) makes it immediately clear to us that we can indeed trigger this gadget remotely. In particular, by trivially overwriting the tainted global variable `ngx_change_binary`, we can force `nginx` to take the conditional branch before `ngx_exec_new_binary`, and therefore call `execve`. Hence, with just a bit of manual inspection, an attacker can escalate non-triggerable gadgets such as this into triggerable gadgets. We note that this approach to "legal" control-flow hijacking differs from previous data-only approaches for a couple reasons. First, it is *optional*, because this gadget can also be triggered: (1) by `nginx`, when it upgrades itself, and (2) by the attacker, by hijacking a tainted `kill` syscall. Second, it is considerably *lower effort*, because we simply flip a single branch that precedes our already-identified gadget, whereas other approaches analyze all possible branches in an attempt to identify new gadgets [94, 100, 182].

Discussion The objective of syscall filter-based defenses [75, 76, 167] is to disable high-value syscalls, e.g., `execve`. However, *syscalls filters are fundamentally imprecise* because they cannot disable legitimate syscalls. In other words, if `nginx` can legitimately invoke `execve`, then a defense cannot disable `execve`—otherwise it would break `nginx`. EINSTEIN exploits this imprecision because it does not divert control flow, and hence, it only covers legitimate syscalls, thereby eliding syscall filter-based defenses.

In principle, if a security-conscious sysadmin designed a configuration for `nginx` that completely disabled all legitimate uses of `execve` (e.g., by disabling CGI, live binary upgrade, etc.), then a syscall filter could indeed disable `execve`, thereby mitigating this gadget. However, in practice, this is nearly impossible because: (1) it is not possible to disable some features from a configuration (e.g., live binary upgrade); and (2) it is non-trivial to determine that no possible program point could invoke `execve` (e.g., due to the presence of dynamically loaded libraries and modules). This observation is confirmed by the evaluation of state-of-the-art syscall filters on `nginx`: none of them are able to completely disable `execve` [75, 76, 167].

3.8.2 Bypassing Selective DFI

This case study exploits an `openat` to `write` gadget chain in `nginx`'s error log. Figure 3.7 walks through how we identify and exploit the gadget with EINSTEIN.

Identification and exploitation After `nginx` enters its quiescent phase, EINSTEIN taints three `nginx` objects in particular: (1) the `err_levels[]` global array, which contains pointers to logging strings (e.g., `"crit"`, `"error"`, `"warn"`) and their lengths; (2) the pathname of a configuration-defined cache file, which `nginx` opens upon requests for `/cache`; and (3) the FD of `nginx`'s error log. Then, EINSTEIN sends requests to `nginx`, two of which trigger the syscalls of our gadget chain. First, it sends a `GET /cache` request, which causes `nginx` to open the cache file. It opens it via an `openat` with a tainted `pathname`. EINSTEIN then adds the created file to its directly-controllable FD table. Second, EINSTEIN sends a `GET /does-not-exist` request, which causes `nginx` to write an `"error"` message to the error log. It does this via a `write` with a tainted `fd`, `buf`, and `count`. Because the FD argument is tainted, and there is a file in the directly-controllable FD table, EINSTEIN determines that the `write` could be redirected to another file, thereby identifying the chain from the `openat` to the `write`.

EINSTEIN builds an exploit for this by first identifying an identity flow between: (1) `openat`'s `pathname` and the cache pathname on the heap, (2) `write`'s `fd` and the error log's FD on the heap, and (3) `write`'s `buf` and the pointer to the `"error"` string in global data. Then, EINSTEIN builds a "write-what-where on the filesystem" candidate exploit by generating $(addr, val)$ pairs that overwrite: (1) the cache file's pathname to a file `"/tmp/attacker-was-here"`, (2) the error log's FD to the cache file's FD, and (3) the pointer to the `"error"` string to a pointer to a `"Hello this is the attacker"` string. Next, we re-run EINSTEIN to confirm this exploit, but unfortunately, it only writes `"Hello"` to our file. Upon closer inspection as to why, we quickly notice from the tagset of `write`'s `count` argument that it depends on the `"error"` string's length—i.e., `0x5`, thereby explaining why only the first five characters of our target string were written to the file. Our analyzer did not initially target the string length because it does not have an identity dataflow to `write`'s `count`, since their values differ. Hence, we modify our exploit to also overwrite the `err_levels[]` string length to `0x1e`, i.e., the length of our string. As a result, we are able successfully write our entire attacker-specified string to our attacker-specified file.

Discussion The objective of selective DFI is to only enforce DFI for select *def-use* chains, e.g., for syscall argument *use*s [102]. However, *selective DFI is fundamentally imprecise* because: (1) it leaves many dataflows unsecure (i.e., it is "selective"), and (2) for any complex dataflows it does attempt to secure, it must statically over-

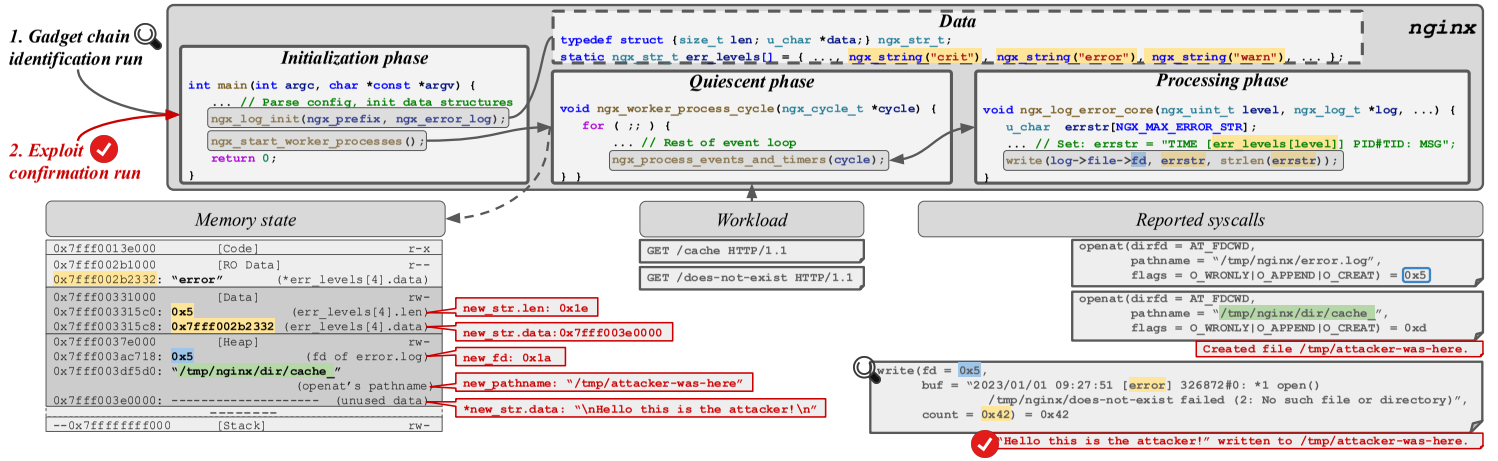


Figure 3.7: An `openat`-to-`write`-based gadget chain in `nginx` that Einstein identified and built an attack for.

approximate the set of legal *def*s for a given *use*, or otherwise risk breaking the target program. EINSTEIN exploits this imprecision because it uses a lightweight dynamic analysis, which encounters no issues with complex dataflows.

Specifically, if the selective DFI's points-to analysis cannot determine all the possible *def*s for some loaded data—e.g., due to notoriously difficult interprocedural analyses, pointer aliasing problems, etc. [88]—then it must assume that *any def* is legal. For example, in our gadget, the dataflow from the *def* of the `"error"` string to the *use* of `write`'s `buf` is difficult to resolve: `err_levels[]` has four address-taken locations that may be passed interprocedurally up to five calls deep (e.g., to a custom implementation of the variadic `sprintf`). Hence, securing every possible *def* for `write`'s `buf` is non-trivial, and over-approximation would likely incur a significant performance overhead. Unsurprisingly, recent work that applies selective DFI to syscall arguments does not attempt to secure `write`s [102]. Finally, we note that this gadget is also resistant to syscall filtering because numerous legitimate program features rely on `openat` and `write`.

3.9 Limitations

Coverage The coverage of our analysis is limited for a few reasons. First, like any dynamic analysis, our code coverage is not complete. Nonetheless, we still find many exploits, and as Section 3.7.2 explains, further work into improving code coverage would also identify more attacks. Second, like other DTA engines, we do not track implicit flows. However, we consider this acceptable, because implicit flows are less useful to an attacker since they typically only propagate one bit of data. Third, EINSTEIN does not build every kind of data-only attack, e.g., those that aim for Turing-completeness. Instead, EINSTEIN aims for a simpler, but more powerful primitive: controllable syscalls. Identifying and eliminating these will raise the bar for attackers significantly.

Constraints Like other approaches that use DTA, our analysis is limited as it does not track all possible dataflow constraints in two ways. First, it does not track exactly how arbitrary program operations (e.g., arithmetic) affects attacker data. However, this is not an issue for us, because our identity dataflow analysis filters out these more complex gadgets, and we are still left with plenty of candidate gadgets. Second, it does not track exactly how attacker data affects the program's path. However, this is also not an issue, because our exploit confirmation filters out these more complex candidate exploits, and we are still left with plenty of working exploits.

3.10 Conclusion

We presented EINSTEIN, a lightweight approach to building practical attacks that are out of reach of prior solutions. We use EINSTEIN to build hundreds of data-only attacks and present two low-effort case studies that bypass state-of-the-art defenses. We conclude that data-only attacks are well within reach of attackers, and hence, vendors should consider stronger defenses such as full DFI and memory safety.

Appendix 3.A Implementation Digression

In this appendix, we discuss the features we added and merged into our DTA engine to make it general-purpose and extendable. We base the implementation of our DTA engine on libdft [111], a DTA engine built on top of Intel Pin [134], which was released more than 10 years ago.

Merged features Since libdft's release, many forks of it have emerged, each adding their own features to suite their own particular analyses [27, 171, 198, 214]. Unfortunately, no single version of libdft exists that supports all of these features, so we merge many of them into one *general-purpose* DTA engine, i.e., our libdft variant. In particular, we merge support for: (1) 64-bit instructions; (2) load pointer propagation, i.e., on a load instruction, propagating not only the tag of the loaded data to the output, but also the tag of the pointer; (3) tagging all data with its own address; (4) an interface to the process's memory map to e.g., check whether some memory segment is writable; and (5) a uniform logging interface, which supports applications that are multi-threaded and applications that redirect standard logging interfaces (e.g., `nginx`).

Added features We also add several new features to libdft. The first set of features we add provide scalability (as discussed in Section 3.5.1). In particular, we add: (1) `userfaultfd`-based tagmap initialization, (2) bounded array-based tagsets, and (3) a shadow memory-based tagmap.

Figure 3.8 describes the layout of our shadow memory implementation. At a high level, the region from `MAIN_START` to `MAIN_END` is the main address space, and the region from `SHADOW_START` to `SHADOW_END` is the corresponding shadow address space. Because the region from `MAIN_START` to `MAIN_START+RESERVED_BYTES` would map into null shadow memory, we cannot use it for program memory. Hence, we reserve this region to store custom tags. For example, to taint a file read (as Angora does [27]), we can create unique tags in this reserved region for each byte of a file. Hence, by supporting custom, unique tags—e.g., for each byte of a file, network read, etc.—our shadow memory provides an *expressive* interface for other analyses.

```
==== 0x000000000000: SHADOW_START
...
==== 0x000000100000: SHADOW_START+RESERVED_BYTES
...
...
...
==== 0x............: SHADOW_END, MAIN_START
...
==== 0x............: MAIN_START+RESERVED_BYTES
...
...
==== 0x7fff00101000: BIN_START
...
==== 0x800000000000: MAIN_END
```

Figure 3.8: Shadow memory layout.

The shadow memory layout assumes the target program is an `x86_64` binary built as position-independent (the default for most compilers), so that its stack and `mmap_base` are at the end of the address space. Next, the layout requires us to relink the target program such that its base address is at `BIN_START`. The value of `RESERVED_BYTES` and `SHADOW_END` depend on the tag size—i.e., the number of colors per tagset, and the size of each color (which depends on whether pointer tag compression is enabled).

The second set of features we add provide other various functionalities. First, we add support for recording memory snapshots. We do this by invoking `gcore` when tainting all memory to produce a core dump. Because the shadow memory comprises a significant portion of the address space, we advise the kernel to not dump shadow pages so that the core dumps are not too large. Moreover, to aid our exploitation tooling, we use `lldb` to also log all runtime symbol information when the snapshot is recorded. Second, we add an interface to receive miscellaneous runtime info. We use this interface to receive the most recently executed line of the program driver, so that we can include it in the syscall reports, thereby aiding our exploitation tooling in knowing the workload that covers a particular gadget chain. Third, we fix many instruction-level taint propagation rules, e.g., by: (1) considering the byte-level semantics of masking operations (e.g., `AND`, `OR`); (2) adding support for various vector instructions (e.g., `VZEROUPPER`, `PUNPCKLQDQ`); (3) correctly handling various x86-64 quirks (e.g., zero-extending a 32-bit `MOV`, but not an 8-, 16-, or 64-bit `MOV`); etc. Fourth, we update our libdft variant to use the newest version of Pin and to support newer syscalls (e.g., `execveat`, `openat2`).

Appendix 3.B Control Data Attack Surface Comparison

In this appendix, we compare our *non-control data* attack surface to the *control data* attack surface. To do so, we re-implement the taint policies of Newton [214] (with our taint engine) for identifying control-flow hijacking attacks, and re-run our evaluation. We note that our non-control data gadgets are inherently more powerful than control data gadgets, because even if an attacker can divert a branch, the attacker still needs to divert it to some target code that e.g., invokes a syscall with controllable arguments. In contrast, our non-control data gadgets already are syscalls with controllable arguments.

Table 3.4 presents the number of non-control data gadgets (i.e., syscalls with attacker-tainted arguments) and control data gadgets (i.e., indirect branches with an attacker-tainted branch target) that we identified in each target program, as well as the percentage which have identity flows. Just as previous work does [214], we count indirect branch sites, rather than backtraces; however, counting backtraces give us similar results. We distinguish control data gadgets based on the type of branch target, i.e., whether it is a register or memory operand.

First, we observe that control data gadget with memory operands have higher rates of identity flows than those with register operands. Upon closer inspection, we notice this is because register operands tend be more constrained, as they are typically used for branches with a bounded set of targets (e.g., switch statements), whereas memory operands are typically used for branches with a relatively unbounded set of targets (e.g., indirect calls). Second, we observe that the rates of identity flows for non-control data gadgets and control data gadgets with memory operands are roughly similar. This is unsurprising, because a program treats many types of long-lived data as "immutable", regardless of whether it is control data (e.g., function pointers), or non-control data (e.g., strings). Hence, we conclude that our lightweight identity flow-based analysis is applicable to domains beyond data-only attacks.

Appendix 3.C Artifact Appendix

3.C.1 Abstract

In this artifact, we provide the means to reproduce our main results. Specifically, we show that our exploitation pipeline, EINSTEIN, identifies vulnerable syscalls across a range of applications, and that it generates working data-only exploits against `nginx`. We have evaluated our artifact using an AMD Ryzen 9 3950X CPU (32 cores), with 128GB of RAM, 4 TB storage, and running Ubuntu 22.04.3 LTS (kernel v6.2). Our source code is available on GitHub[5].

[5] `https://github.com/vusec/einstein/`

3.C.2 Description & Requirements

Security, privacy, and ethical concerns

Although EINSTEIN indeed produces working exploits, they are non-destructive proof-of-concept exploits, which write the string `"HELLO"` to either a file (`"/tmp/hi"`) or a local socket (address `"192.0.2.0"`). Hence, evaluating EINSTEIN poses no risks for machine security, data privacy, or other ethical concerns.

How to access

The files for the artifact evaluation are available at the `ae` tag of the EINSTEIN repository[6].

Hardware dependencies

EINSTEIN requires an x86-64 machine (Intel recommended); enough RAM to simultaneously load multiple program snapshots into memory, so EINSTEIN can post-process reports in parallel (recommended 100 GB RAM); and enough storage for hundreds of program snapshots (minimum 2 TB storage for this evaluation). We recommend using a machine with a high core count to speed up EINSTEIN's report post-processing.

Software dependencies

To build EINSTEIN and the target programs, we expect certain packages to be installed. In the Section 3.C.3, we detail the steps to install such dependencies on Ubuntu 22.04, but similar steps are needed for other distributions.

Benchmarks

We use each target application's test suite to drive the analysis.

3.C.3 Set-up

To download and install dependencies, including go-task as a task-runner, from the EINSTEIN repository, run: `sudo snap install task --classic && task init`.

Installation

To build `libdft`[7], the command server, the EINSTEIN tool, and all target applications, run: `task libdft-build cmdsvr-build einstein-build apps-build`.

[6] `https://github.com/vusec/einstein/releases/tag/ae`
[7] `https://github.com/vusec/libdft64-ng`

Basic test

We first make a couple notes about running EINSTEIN:

- Due to the non-deterministic nature of dynamic analysis (from concurrency issues, system variability, etc.) [13, 57], the actual results may slightly deviate from the expected results.
- If the `db-analyze-reports` task fails, try running the `db-analyze-reports-singleproc` task instead. It will be slower, but will avoid any system load-related crashes.

Test that the different components work as follows:

(T1): *libdft memory tainting [1 compute-second].*
To test libdft's "taint all memory" functionality, run `task libdft-test -- memtaint` and compare its output to the expected output.

(T2): *libdft instruction tainting [1 compute-second].*
To test libdft's per-instruction taint policies, run `task libdft-test -- ins` and compare its output to the expected output.

(T3): *EINSTEIN tool [1 compute-minute].*
To test EINSTEIN on a simple program, run `task einstein-test`. Then, compare the output of `task db-print-candidates` with the expected output.

(T4): *Target applications [4 compute-minutes].*
To test EINSTEIN running each target application with a simple workload (e.g., sending a simple GET request to a web server), run `task reports-clean apps-test db-add-reports db-analyze-reports`. Then, compare the output of `task db-print-candidates` with the expected output.

(T5): *Target application test suites [20 compute-minutes].*
To test EINSTEIN running each target applications' test suites for 2 minutes each (rather than the entire test suites), run `task reports-clean apps-eval-brief db-add-reports db-analyze-reports`. Then, compare the output of `task db-print-candidates` with the expected output.

(T6): *Exploit confirmation [2 compute-minutes].*
To test EINSTEIN's exploit confirmation for `nginx`, run `task reports-clean einstein-nowrite-config nginx-eval-custom db-add-reports db-analyze-reports db-analyze-candidates`. Then, compare the output of `task db-print-exploits` with the expected output.

3.C.4 Evaluation workflow

Major claims

We make the following claims:

(C1): *EINSTEIN identifies thousands of vulnerable syscalls in common server applications. This is proven by Experiment (E1).*

(C2): *EINSTEIN generates hundreds of working exploits against `nginx`. This is proven by Experiment (E2).*

Experiments

We prove the above claims using the following experiments:

(E1): *Vulnerable syscall identification [24 compute-hours].*
How to: *We will run each application with EINSTEIN, then analyze the reports to identify vulnerable syscalls.*
Preparation: *Run `task reports-clean` to remove past reports.*
Execution: *Run `task apps-eval db-add-reports db-analyze-reports`.*
Results: *Compare the output of `task db-print-candidates` to the expected output. The output contains thousands of vulnerable gadgets, broken down by: (i) syscall and argument type (i.e., Table 3), and (ii) target application (i.e., Table 4)—thereby proving Claim (C1).*

(E2): *Exploit generation [12 compute-hours].*
How to: *We will run `nginx` with EINSTEIN, then analyze the reports to identify vulnerable syscalls, then confirm candidate exploits as working exploits.*
Preparation: *Run `task reports-clean` to remove past reports.*
Execution: *Run `task nginx-eval db-add-reports db-analyze-reports db-analyze-candidates`.*
Results: *Compare the output of `task db-print-exploits` to the expected output. The output contains hundreds of confirmed exploits for `nginx` (i.e., Table 5)—thereby proving Claim (C2).*

3.C.5 Notes on Reusability

This prototype may be expanded in a few directions:

- To modify EINSTEIN's taint policies (e.g, to target more syscalls, or to target syscall-guard variables), modify the EINSTEIN tool in `src/einstein`.
- To run the target applications (e.g., `nginx`) with other workloads, first start the application with EINSTEIN (`cd apps/nginx-1.23.0 && RUN_EINSTEIN=1 ./serverctl restart`), then run the custom workload (e.g., `echo 'Hello!' | netcat 127.0.0.1 1080`).
- To run EINSTEIN on other applications:
  1. Add the application to the `apps/` directory;
  2. Copy the files `serverctl` and `clientctl` from another application's directory into its directory, and modify them to start the application's server and a client for it; and
  3. Ensure that the application's build script generates position-independent code (i.e., the default on most compilers).

- To write another Pin tool that uses `libdft`:
  1. Copy the EINSTEIN tool, e.g.: `cp -r src/einstein src/my-tool`;
  2. Modify `MY_TOOL` and `MY_OBJS` in the `Makefile`;
  3. Modify the source code to suite your analysis;
  4. Build it: `cd src/my-tool && -DLIBDFT_TAG_PTR -DLIBDFT_PTR_32 -DLIBDFT_TAG_SSET_MAX=16' make obj-intel64/my-tool.so`; and
  5. Run it on some target application: `setarch x86_64 -R ./src/misc/pin-3.28-98749-g664 -t src/my-tool/obj-intel64/my-tool.so -- echo 'Hello!'`.

3.C.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

Author and Contributor Statement

Conceptualization, Software, Validation, Investigation, Visualization: Brian Johannesmeyer; *Methodology, Writing - Original Draft, Writing - Review & Editing*: Brian Johannesmeyer, Asia Slowinska, Herbert Bos, Cristiano Giuffrida; *Supervision*: Asia Slowinska, Herbert Bos, Cristiano Giuffrida.

4 Dynamic Detection of Vulnerable DMA Race Conditions

The drivers of modern operating systems use Direct Memory Access (DMA) to efficiently communicate with peripheral devices. Since the memory accessed by DMA is a shared resource between driver and device, it is a possible source of race conditions. Peripheral devices are also often untrusted, so these race conditions open up a new potential attack vector against a trusted OS kernel.

In this chapter, we present DMARacer, a dynamic detector called for these DMA-based race conditions in kernel code. DMARacer tracks memory accesses to DMA memory throughout the kernel's lifetime and analyses them for various indicators of race conditions. Additionally, upon detecting a race condition, DMARacer uses taint tracking to trace its impact and identify any potential vulnerabilities it may trigger, such as memory corruption or denial-of-service. We used DMARacer to search the drivers of the Linux kernel for DMA-based errors and find that DMA-based race conditions are a systemic issue in driver code. In total, DMARacer was able to detect 817 problematic memory accesses and 344 vulnerable operations in the scanned Linux kernel drivers.

4.1 Introduction

> *"We trust the hardware pretty implicitly. There are bits and pieces of the kernel that are starting to nibble away at the "do we trust this?" portion, but for the most part, this is not how Linux was designed at all."*
>
> *— Greg Kroah-Hartman, Linux kernel maintainer (in response to our disclosure)*

Where OS developers in the past limited their security concerns to malicious user programs [177], operating system kernels today *have* to include peripheral devices in their threat model [3]. Indeed, most general-purpose computers now come equipped with IOMMUs [3]—mirroring established protection mechanisms that prevent unauthorized memory access by user programs, by means of MMUs [155], to also isolate memory for external devices. However, there is also memory that the kernel intentionally shares to communicate with either user space or peripherals—and doing so introduces the risk of race conditions.

For example, the shared memory used to transfer data from a user program to the kernel in a system call may easily become a source of race conditions. Moreover, the kernel and user/device have different privilege levels, putting them out of reach of standard synchronization approaches and verification tools [65, 66, 164, 178, 188, 206] that rely on all processes synchronizing voluntarily. Previous work extensively studied race conditions, such as double fetches, at the system call interface [16, 63, 107, 131, 135, 183, 221, 222, 231].

However, the problem also exists for drivers at the boundary between kernel and untrusted devices. As with the system call interface, the kernel and device both access shared memory to communicate. Using Direct Memory Access (DMA), a malicious device may surrepitiously modify data that the kernel assumes is valid. Recent work studied DMA-based communication between driver and device and highlighted that standard security practices (e.g., input validation and IOMMU protection) are not yet consistently adopted [7, 11, 42, 69, 101, 113, 136, 140, 142, 161, 165, 192, 196, 204, 230, 233, 241].

Despite these findings, the research community has largely ignored the issue of DMA-based race conditions. The shared memory and the different privilege levels mirror the conditions that lead to race conditions and double fetch bugs on the user-kernel interface. However, at the device-kernel interface, they may lead not just to "classic" TOCTOU-like conditions, but also other, less-known but equally dangerous, ones. Examples include races in streaming DMA, as well as cases where attackers corrupt kernel-initialized data. The latter category is of particular concern as they are often exploitable [7, 140] and we find that such vulnerabilities are much more common than TOCTOU ones. This raises the question of how to detect such conditions in the various device drivers found in modern kernels?

In this work, we fill this gap by identifying DMA-based race conditions using insights from prior research on user-to-kernel race conditions. Unfortunately, applying these insights to the *device-to-kernel* domain poses unique challenges. For instance, few (if any) of the interactions across this boundary are standardized and interposition is hard: devices may interact with the kernel through custom driver interrupts, and the kernel may access DMA through ordinary memory operations. Thus, we cannot incorporate the methods from previous studies of unsafe DMA accesses directly and must instead adapt them specifically to DMA race conditions. In particular, unlike unsafe DMA accesses, DMA race conditions require analyzing memory accesses collectively, as well as tracking of long-lived state. For this purpose, we use dynamic taint analysis (DTA) [149] to track such complex state across complex interactions.

We present DMARacer, a dynamic DMA race condition detector for the Linux kernel. DMARacer monitors runtime invariants to determine whether DMA accesses are race-free. When DMARacer identifies a violation—a race condition—it reports the access as an *errant access*. If the errant access involves attacker-controllable data (e.g., loading from coherent DMA), we apply taint to the data to track it throughout the kernel's execution. Finally, if the tainted data reaches a security-sensitive operation (e.g., a memory write pointer), DMARacer flags it as a *vulnerable operation*.

Like other sanitizers, DMARacer instruments the kernel to detect these issues during runtime. Specifically, it (i) tracks DMA state by hooking into every DMA operation, (ii) identifies errant accesses by monitoring all memory accesses, (iii) identifies vulnerable operations through DTA policies, and (iv) covers DMA race conditions by executing a variety of device-to-kernel interactions. In combiation, it enables DMARacer to detect DMA race conditions.

By applying race condition detection to the novel domain of DMA data, DMARacer identifies 817 errant accesses and 344 vulnerable operations across the kernel—with false positive rates of 0% and 9%, respectively. Moreover, our case studies show that these race conditions are not easily eliminated: many are deeply embedded in the semantics of existing drivers and APIs.

Contributions We make the following contributions:

- We present a new dynamic analysis approach for detecting DMA race conditions and vulnerable device-to-kernel interactions.
- We develop DMARacer, an open-source[1] DMA race condition detector for the Linux kernel.

[1]DMARacer is available at `https://github.com/vusec/dmaracer`.

- We evaluate DMARacer on a recent kernel to identify hundreds of errant accesses and vulnerable operations, and present case studies that highlight that the issues are difficult to mitigate.

4.2 Background

Race conditions Race conditions are non-deterministic errors that occur when several processes access a shared resource without synchronization[148, 178]. A common synchronization mechanism are locks that grant exclusive access to the variable for a limited time. Many synchronization primitives require the cooperation of all involved processes to work correctly. They do not work if one of the processes is malicious and ignores the synchronization primitive.

TOCTOU bugs A special kind of bug caused by race conditions are time-of-check to time-of-use (TOCTOU) errors. Here, a piece of code first checks whether a certain property holds for a shared resource. The rest of the code then incorrectly assumes this property still holds [168], even though there is no mechanism in place that ensures the immutability of the shared resource. An unprivileged attacker can abuse this behavior to potentially exploit the code that relies on the checked property. In a concrete attack, the attacker would set the shared resource to a 'good' state to satisfy the check, and then modify it before the first to a 'bad' state that causes unintended behavior.

Double fetch bugs A common attack scenario involving TOCTOU errors are kernel system call handlers where they are also referred to as *double-fetch bugs* [107, 131, 183, 221, 222, 231]. In this scenario, the attacker performs a system call as an unprivileged user and the shared resource is the memory containing the system call arguments. This memory is accessible by both user and kernel while the system call is being performed. The attacker can therefore modify an argument between the time the kernel checks it for validity (1st fetch) and the actual processing of the arguments (2nd fetch). If the attacker is successful and is able to bypass a critical check such as a buffer size check, the consequences of such an attack can result in information leakage or privilege escalations [221].

Direct memory access Direct memory access (DMA) is a hardware feature that reduces the CPU workload when communicating with peripheral hardware. In a system with DMA, a dedicated DMA controller is responsible for moving data between system memory and hardware. The CPU itself is only responsible for initiating the data transfers and can be utilized for other tasks while the data is being moved.

Kernel drivers utilize DMA by allocating a special DMA buffer that is bound to a device's registers or memory. The buffer itself is used like any other plain memory buffer in C and can be directly written to and read from without invoking special functions. The driver can decide between two kinds of DMA buffers that differ in when they synchronize their contents with the device.

1. *Streaming* (or asynchronous) DMA buffers are used for single DMA-based data transfers. The transfer to and from the device is explicitly initialized by the driver. As the assumption is that driver and device do not access the same memory region at the same time, there are generally no strong guarantees with respect to cache coherence.
2. *Coherent* (or synchronous) DMA buffers are implicitly synchronized between device and system memory. These buffers are set up in cache-coherent memory, therefore any memory writes done by either device or CPU are immediately made visible to the other. Coherent buffers are used when the driver and the device require concurrent write access.

DMA errors The two kinds of DMA buffers impose each a very different challenge for a driver that uses them. For streaming DMA buffers, the main challenge is the correct timing of the synchronization request. All data that needs to be transferred must have been written to the buffer before it is synchronized with the device.

Coherent buffers do not impose this requirement, but instead come with different security implications. Because their contents can be concurrently accessed by an untrusted device and the kernel, they can be a potential source of TOCTOU bugs. The possible attack scenario using these bugs is very similar to the TOCTOU attacks on kernel system call handlers (see the previous section). Instead of the user switching out system call arguments, a device could change the contents of a DMA buffer after the driver performed the necessary sanity checks on it.

4.3 Threat Model

We adopt the threat model of previous work [11], which considers a local attacker who controls a peripheral device (e.g., a USB keyboard) and its workload. The kernel and its drivers are trusted and not compromised. The attacker's goal is to cause a denial of service (e.g., trigger a kernel panic) or achieve privilege escalation (e.g., corrupt kernel memory). The ability to control the peripheral device allows the attacker to observe all DMA accesses performed by the kernel driver and control the values of the accessed DMA memory. E.g., the attacker can observe that a certain part of the DMA memory is accessed twice within a short time span and change the memory contents so that two different values are read by each access. The system memory is protected by an idealized IOMMU that can protect each individual

byte of memory against unintentional writes from an external device. This means the system is invulnerable against accidental exposure of memory caused by too coarse-grained IOMMU protection capabilities [140].

4.4 Defining DMA Race Conditions

In this section, we will define three possible race conditions that can arise out of improper DMA usage.

4.4.1 Coherent DMA-based Race Conditions

Coherent DMA is simultaneously accessible to both the kernel and a malicious device, similar to how userspace data is simultaneously accessible to both the kernel and a malicious program. Hence, we can gain insights from userspace-based race conditions when defining coherent DMA-based race conditions.

TOCTOU bugs Previous work on user-to-kernel TOCTOU bugs [107] holds that the kernel should not load the same user data twice within a single execution context (i.e., an interrupt, system call, or new kernel thread). We apply this same rule to coherent DMA:

> Invariant 1. Avoiding TOCTOU bugs.
>
> The kernel should not read coherent DMA data more than once within an execution context.

If this invariant is violated, the kernel unsafely races against a malicious device corrupting the previously kernel-loaded data. Although drivers may sometimes busy-poll MMIO/PMIO status registers for ultra-low-latency I/O [124], polling a DMA buffer itself is uncommon—completion is normally reported via interrupts or other asynchronous mechanisms [41]. Thus, our invariant holds in practice (see Section 4.8).

TOITOU bugs Recent DMA exploitation techniques [7, 140] target bugs that involve the corruption of kernel-initialized DMA data. We term such cases *TOITOU*, time-of-*initialization* to time-of-use, bugs. A TOITOU bug is defined as a race condition with the following three phases: (1) the victim *stores* valid data in a shared resource, (ii) the attacker overwrites this data after it was stored, and (iii) the victim later loads the data and assumes it is still valid, thus omitting any validity checks. The difference to a TOCTOU bug is that instead of a check being skipped, the user's attempt to satisfy a check by modifying a shared resource is disabled.

Unlike the user-to-kernel domain, where the kernel only rarely initializes data in userspace for later use (e.g., when signal handling [19]), kernel drivers will oftentimes initialize entire data structures in DMA. These drivers also expect the device to play nice and only access the parts that it is supposed to. Typically, the kernel initializes such data early on (e.g., during boot time), then uses it much later on (i.e., after boot time), all the while, expecting it to remain unchanged. Hence, we can derive a second invariant:

Invariant 2. Avoiding TOITOU bugs.

The kernel should not read coherent DMA data if it previously initialized it.

If this invariant is violated, the kernel unsafely races against a malicious device corrupting the previously kernel-initialized data.

4.4.2 Streaming DMA-based Race Conditions

In contrast to coherent DMA, streaming DMA is only accessible to either the kernel or the device—but not both. The kernel governs this accessibility by invoking synchronization operations that transfer access rights between the kernel and the device.

Inconsistent access bugs Previous work on identifying unsafe DMA accesses notes that the kernel should not access a streaming DMA region when it is synchronized for device access [11]. Doing so results in an *inconsistent DMA access bug* because the data accessed by the kernel (either in the CPU cache or a bounce buffer) may not be consistent with the actual data in the (device-accessible) DMA buffer [41]. Hence, we apply this same invariant:

Invariant 3. Avoiding inconsistent access bugs.

The kernel should not access streaming DMA data if the region is synchronized with the device.

If this invariant is violated, the kernel unsafely races against the CPU cache or bounce buffer inadvertently corrupting the data. Notably, this bug is very difficult for a malicious device to exploit, so we consider such exploitation out of scope. Instead, it primarily poses a reliability issue when the kernel later uses the corrupted data. Moreover, because synchronization operations are non-blocking, the

manifestation of this bug is nondeterministic: the race condition may only occur when the synchronization commits, making it difficult to consistently reproduce and diagnose.

4.5 Overview

Having defined the various DMA race conditions, we now present the main design challenges and how we addressed them to detect these conditions. To detect DMA race conditions, DMARacer has to track all memory operations and determine which are accessing DMA memory. DMARacer must then identify which parts of the kernel are potentially affected by attacker-controlled data from these errant accesses.

Figure 4.1 presents an example of how DMARacer detects vulnerable code. Although this example features a hypothetical TOCTOU bug for illustrative purposes (given its similarity to well-studied double-fetch bugs), it is important to note that, as discussed in Section 4.8.3, most vulnerabilities in practice stem from TOITOU bugs instead.

Ⓐ Tracking DMA regions DMA memory regions are dynamically created and then accessed like any other buffer via C pointers in the rest of the kernel code base. We therefore cannot hook a standardized access method (e.g., `copy_from_user()` on the kernel-user barrier) to detect DMA memory accesses. Instead, DMARacer hooks all relevant DMA methods and maintains its own metadata for the location and state of all DMA buffer. In Section 4.6.1, we detail how we hook the variety of DMA APIs within the kernel.

Ⓑ Identifying errant accesses DMARacer has to determine all DMA memory operations and identify any *unsafe* DMA accesses among them. As DMA is accessed via normal store and load instructions, DMARacer uses its collected DMA metadata to determine which memory operations access DMA buffers. For errors like TOCTOU that involve several memory operations, DMARacer also has to reason about the relationship between different memory accesses. We support this post-hoc analysis of several DMA operations by additionally maintaining a list of all DMA memory accesses. In Section 4.6.2, we describe how our monitor checks whether DMA is accessed, and if so, whether the access violates an invariant.

Ⓒ Identifying vulnerable operations Once DMARacer detects an errant DMA access, it is still unclear whether the attacker-controlled from the access is actually vulnerable. DMARacer searches for potentially exploitable code by applying taint to attacker-controlled DMA data and tracking it through the kernel's execution. This taint spreads across the kernel during execution until it reaches operations that can

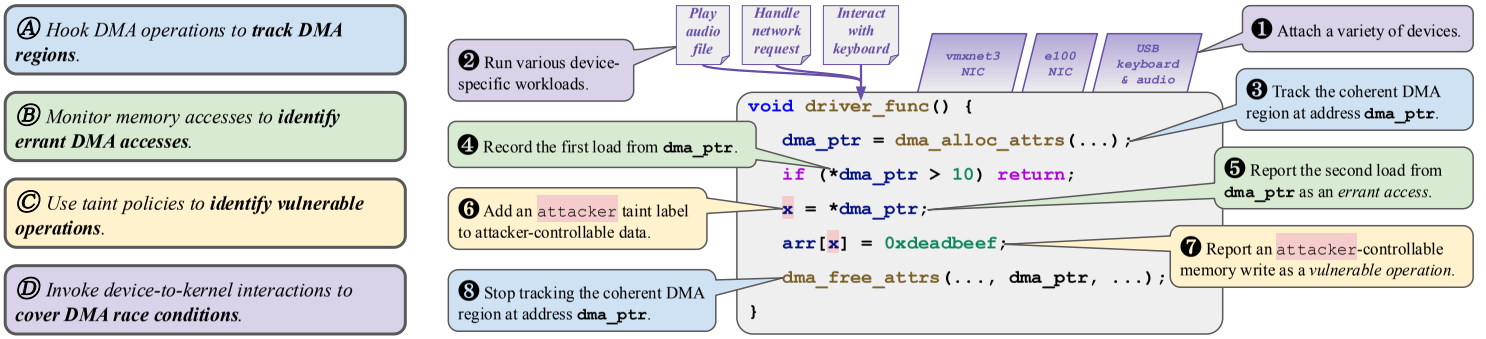


Figure 4.1: The components (Ⓐ–Ⓓ) used by our approach and the steps (❶–❽) they take to identify an example TOCTOU bug and a dependent attacker-controllable memory write.

Taint policy type	Operate on	Justification
Source: Track attacker data	Output of a TOCTOU/TOITOU bug	Controllable by a malicious device
Sink: Identify vulnerable operations	Pointer of a memory write	Enables an attacker-controllable memory corruption primitive
	Condition of a loop/assertion	Enables an attacker-controllable denial-of-service primitive
Clear: Avoid taint explosion	`AND` instruction	Instruction used to access specific bits of DMA data
	Kernel hot paths	Execution may spill taint into frequently accessed global data
	Execution context entry	Cross-execution context dataflows are challenging to reproduce

Table 4.1: Taint policies to track attacker data to dependent vulnerable operations, while avoiding taint explosion.

DMA op.	Function hooked	DMARacer handler
Map	`dma_alloc_attrs()`	Track a *coherent* DMA buffer
	`dma_map_single_attrs()`	Track a *streaming* DMA buffer
	`__dma_map_sg_attrs()`	Track a *streaming* DMA SG list
Unmap	`dma_free_attrs()`	Stop tracking a *coherent* DMA buffer
	`dma_unmap_page_attrs()`	Stop tracking a *streaming* DMA buffer
	`dma_unmap_sg_attrs()`	Stop tracking a *streaming* DMA SG list
Sync	`dma_sync_single_for_cpu()`	*CPU*-sync a streaming DMA region
	`dma_sync_single_for_device()`	*Device*-sync a streaming DMA region

Table 4.2: DMA operations hooked by DMARacer to track DMA state at runtime.

enable exploits if certain operands are attacker-controlled (e.g., the address of a store operation). In Section 4.6.3, we describe our taint policies, and the subtleties in tracking DMA data.

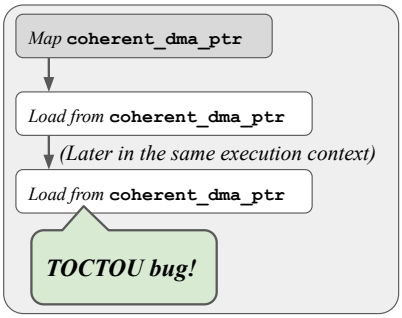
Ⓓ Covering DMA race conditions Unlike the user-kernel boundary with syzkaller [79], there is no production-ready tooling for testing the device-kernel boundary. Given that DMARacer is a *dynamic* analysis tool, we therefore also need a way to run drivers and give them meaningful workloads that utilize DMA operations and spread them across the kernel. To overcome this challenge, we leverage the flexibility of a virtual environment to invoke device-to-kernel interactions. This allows us to easily configure the kernel to support a variety of drivers, attach multiple devices, and run diverse device-specific workloads. In Section 4.8.1, we explain this pipeline, and how future work could build upon our flexible tooling.

4.6 Detecting DMA Race Conditions

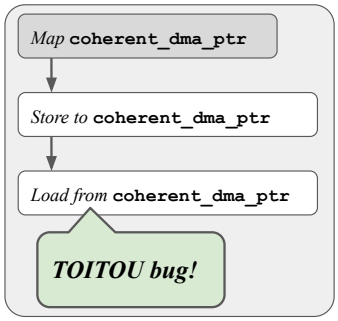
In this section, we detail each component of our approach to detecting DMA race conditions.

4.6.1 DMA Operation Hooks

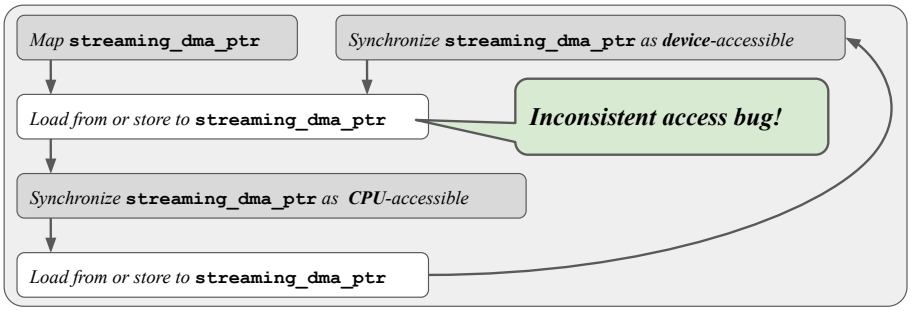
Table 4.2 presents our DMA operation hooks, which allow us to track DMA state at runtime. For both coherent and streaming DMA, we begin tracking a region at a *map* (or allocate) operation, and stop tracking it at an *unmap* (or free) operation. When streaming DMA is mapped, we designate it is as device-accessible; for every *synchronization* operation thereafter, we designate it as either CPU- or device-accessible. Moreover, because the kernel may synchronize DMA partially—i.e., only a subset of the region—we track synchronization state at a per-byte granularity.



(a) Invariant 1 violation.



(b) Invariant 2 violation.



(c) Invariant 3 violation.

Figure 4.2: Memory access monitor policies for detecting invariant violations.

Applicability to various DMA APIs The kernel offers a variety of APIs to map, unmap, and synchronize DMA. However, because we hook the lowest-level DMA operations, which are called by these APIs, we therefore also hook into these various APIs. For example, the various interfaces for allocating coherent DMA—e.g., the generic (`dma_alloc_coherent()`), managed (`dmam_alloc_coherent()`), pool (`dma_pool_alloc()`), and driver-specific (e.g., `hcd_buffer_alloc()`) APIs—all eventually call `dma_alloc_attrs()`. Therefore, by hooking `dma_alloc_attrs()`, we also hook such allocation interfaces.

4.6.2 Memory Access Monitor

To monitor every memory access, we insert callbacks to our DMARacer runtime library at memory access instructions. From our callback, we first check whether the access is to a DMA region, and if so, whether it violates an invariant, as outlined in Figure 4.2.

Coherent DMA accesses

For loads from coherent DMA data, we check whether the data was previously accessed. Specifically, whether it was previously: (i) loaded from within the same execution context, which indicates a TOCTOU error (Invariant 1), or (ii) stored to within the lifetime of the kernel, which indicates a TOITOU error (Invariant 2).

For this purpose, we record every access into one of two structures: (i) if an access *loads* from DMA, we record it into a *local access map*, which is local to a particular execution context, and is cleared when the execution context exits; or (ii) if an access *stores* to DMA, we record it into a *global access map*, which is global to the entire kernel, and persists through the kernel's lifetime. If a load from DMA has a preceding load in the *local* access map, then we report a *TOCTOU bug* (as in Figure 4.2a). Otherwise, if it has a preceding store in the *global* access map, then we report a *TOITOU bug* (as in Figure 4.2b).

Storing kernel pointers to coherent DMA We observe one concerning pattern in DMA usage: the kernel frequently storing pointers to DMA. Typically, this may be because it maintains a data structure (e.g., a linked list) in DMA, and expects the device to only access certain parts of the structure (e.g., the "data" of a linked list, but not the pointers).

However, because devices generally do not have access to the same address space as the kernel, the only practical reason that the kernel would store a pointer to DMA is if it will use it later. Hence, such an operation is either bad practice (at best), or the beginning of a vulnerability[2] (at worst). If it uses the pointer later, we would report it as a TOITOU bug. Therefore, if the kernel stores a pointer to DMA, we report it as an errant access, because we expect the kernel to use it later, which would then be a TOITOU bug. By reporting the initial pointer store as an errant access, we mitigate the case where our dynamic analysis' imperfect coverage fails to cover the subsequent pointer load.

Streaming DMA accesses

For accesses to streaming DMA, we check whether any part of the accessed region is device-synchronized. If yes, the access violates Invariant 3 and demonstrates an invalid DMA memory access. Figure 4.2c outlines this process: The kernel *may* access streaming DMA data if it is immediately preceded by a CPU-synchronization operation. However, the kernel *may not* access it if it is immediately preceded by a mapping or device-synchronization operation; otherwise, we report the access as an *inconsistent access bug*.

[2]This may also be considered information leakage (e.g., breaking KASLR). However, we focus on the risk of race conditions in this work.

4.6.3 Taint Policies

Table 4.1 summarizes our taint policies, which track attacker data from errant accesses to dependent vulnerable operations.

Taint sources

From our memory access callbacks (Section 4.6.2), we add a unique taint label to the output of attacker-controllable errant accesses (i.e., TOCTOU or TOITOU bugs). Hence, as is typical of DTA systems [149], we are able to track the flow of attacker data at runtime by checking its taint label: untainted data *is not* attacker data, and tainted data *is* attacker data. In our case, our taint label identifies the specific errant access (i.e., the backtrace) that loaded the attacker data.

Taint sinks

We track the flow of attacker data to certain *security-sensitive operations*. Following previous work [11], we target operations leading to memory corruption and denial-of-service (DoS) vulnerabilities. Specifically, our security-sensitive operations are: (i) A tainted pointer of a store instruction, because an attacker may redirect it to corrupt memory; hence, we report it as a *vulnerable write*. (ii) A tainted loop condition, because an attacker may corrupt it to loop indefinitely, leading to a DoS or buffer overflow; hence we we report it as a *vulnerable loop*. (iii) A tainted assertion condition, because an attacker may corrupt it to fail the assertion, leading to a DoS, so we report it as a *vulnerable assert*.

We note that this is not an exhaustive list of *all possible* security-sensitive operations; rather, it is a set to demonstrate the feasibility of our approach. DMARacer can be easily extended to hook into more security-sensitive operations.

Taint clears

A known challenge of DTA is avoiding taint explosion—i.e., the accidental spilling of taint into data that is not intended to be tainted. If unaddressed, we may incorrectly report certain non-vulnerable operations as vulnerable. The ideal solution for avoiding taint explosion is to perfectly model all possible dataflows in the program. However, doing so for every possible operation in a program is a difficult task [234].

Instead, we follow an approach used by other DTA systems [104, 180, 207] and clear taint at various operations to mitigate this issue. In general, our taint policies aim to be conservative to reduce false positives. That is, we clear taint unless there is a likely dataflow for a certain operation or code pattern.

Bit-level accesses It is not uncommon for the kernel to access specific *bits* of DMA data (e.g., a 1-bit flag). In such a case, the kernel may e.g., load a one-byte word from DMA, then perform a bitwise `AND` to isolate the particular bit. Unfortunately, *bit-level* taint analysis—which could be used to accurately model such a dataflow—is known to be difficult [234].

Hence, to avoid false positives arising from such a dataflow, we instead clear its taint. Specifically, we modify the taint analysis' `AND` instruction callback—which normally propagates the taint from its inputs to its output—to instead clear the taint of its output.

Flows to frequently-accessed global objects Certain parts of the kernel (e.g., the timer subsystem and the slab allocator) are called frequently throughout the kernel's execution and handle globally-accessible data structures. Hence, if taint passes into one of these kernel "hot paths", it soon spills into unrelated operations throughout the kernel. In principle, a heavyweight constraint solver could address this (e.g., by only propagating verifiably controllable dataflows to these hot paths). However, in practice, such approaches do not scale to the size and complexity of the Linux kernel.

Therefore, to avoid false positives arising from such dataflows, we adopt an approach from an existing kernel DTA system [207]. Specifically, we clear taint for all operations and accessed memory within various kernel hot paths[3].

Flows across execution contexts The kernel maintains various data structures (e.g., sockets and file descriptors) that persist state from one execution context to another. If taint flows into one of these structures in one execution context (e.g., a timer interrupt), and is used in another execution context (e.g., some syscall), it is often difficult to reproduce the dataflow, as it may rely on non-deterministic (e.g., timing) constraints.

Hence, we maintain a notion of taint *validity*, which enforces same-domain dataflows using the following semantics: (i) An execution context begins with all taint labels marked as *invalid*. (ii) If a taint source (i.e., an errant access) is covered, its taint label is marked as *valid*. (iii) A taint sink only reports vulnerable operations if its taint label is *valid*. As a result, we ensure that the vulnerable operations we identify are accurate and easily-reproducible.

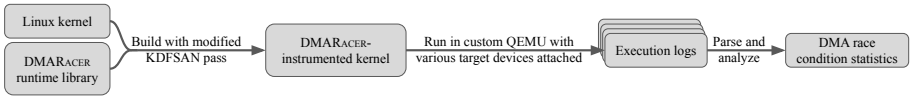


Figure 4.3: The workflow of DMARacer, which takes the Linux kernel, identifies DMA race conditions, and presents statistics to aid developers in applying mitigations.

4.7 Implementation

Figure 4.3 presents the workflow of DMARacer, which: (i) takes the Linux kernel and our runtime library, (ii) builds them with our LLVM instrumentation, and (iii) runs the instrumented kernel as a VM in our custom QEMU hypervisor, where it identifies vulnerable DMA race conditions at runtime.

Runtime library Our runtime library consists of: (i) the KernelDataFlowSanitizer (KDFSAN) [104] runtime library, which provides support for DTA (e.g., by creating taint labels, combining labels, etc.); and (ii) the handlers for DMARacer's various *instruction*-level and *function*-level callbacks, which collectively identify vulnerable DMA race conditions.

Specifically, the DMARacer runtime library handles the following function-level callbacks: First, for *DMA operations*, it tracks the state of DMA regions. Second, for *execution context entries and exits* (e.g., `__enter_from_user_mode()`, `irq_enter_rcu()`), it: (i) clears the local access map and (ii) marks taint labels as invalid. Third, for *assertions* (e.g., `BUG_ON()`), it performs the taint sink for vulnerable asserts.

Furthermore, it handles the following instruction-level callbacks: First, for *memory accesses*, it: (i) records DMA accesses, (ii) performs the taint source for errant accesses, and (iii) performs the taint sink for vulnerable writes. Second, for *backward conditional branches*, it performs the taint sink for vulnerable loops. Third, for *`AND` operations*, it clears the output's taint.

LLVM instrumentation We build the kernel with the KDFSAN pass, which we modified to add the above instruction-level callbacks. First, to hook memory accesses, we reuse KDFSAN's callbacks for `LOAD`, `STORE`, and `MEMTRANSFER` instructions. Second, to hook backward conditional branches, we modify KDFSAN's conditional `BranchInst` visitor to check its direction (via the `PostDominatorTree` API), and if it is backwards, to add our callback. Third, to hook `AND` operations, we modify KDFSAN's `BinaryOperator` visitor to add our callback for `AND` instructions.

[3]The semantics are similar to those of KMSAN's `__no_kmsan_checks` function attribute; indeed, we clear taint in many of the same functions where this attribute is applied.

4.8 Evaluation

In this section, we evaluate DMARacer's ability to detect DMA race conditions.

4.8.1 Setup

Environment We perform our evaluation on a host machine with an AMD Ryzen 9 3950X CPU and 128GB of RAM, running Ubuntu 22.04.4 LTS (kernel v6.8). We instrument the kernel (based on Linux v6.5.8) with a modified KDFSAN pass (based on LLVM v11.0.1), and run the instrumented kernel as a guest VM in QEMU.

Device emulation with QEMU Dynamic error detectors require that the code is executed to be analyzed. In the case of DMARacer, this means that a driver needs to allocate a DMA buffer and then violate one of the invariants. Additionally, the detection of vulnerable operations relies on tainted data from DMA to spread across the kernel.

Device driver code can only be successfully executed if their respective device is connected. Because we do not have access to the multitude of physical devices supported by the Linux Kernel, we instead decided to use the virtual device emulation feature of QEMU for our evaluation. This emulation feature allows QEMU to mimic a connected peripheral which includes the kernel-device communication via DMA. This emulation is only available for QEMU devices for which the QEMU developers implemented a backend. The backend is responsible for communicating with the running kernel in the same way as the real hardware would. Table 4.4 shows a summary of the 147 QEMU devices we used in our evaluation. These 147 devices include all QEMU devices except CPUs and various test devices (e.g., devices that are only used for educational purposes).

Evaluation workloads For each emulated QEMU device we decided to evaluate, we also needed a way to exercise the DMA communication of the device and spread the data from DMA across the kernel. Unfortunately, standard kernel fuzzing techniques such as syzkaller are not suitable for this purpose for two reasons. First, they focus on general code coverage which is not directly relevant for DMARacer. For example, just covering a corner case in a system call handler will never lead to a report from DMARacer unless the code accesses DMA memory or handles tainted data from DMA. Second, some driver logic depends heavily on input from DMA devices itself (e.g. drivers for input devices), and fuzzers like syzkaller focus on the unrelated user-kernel interface.

To overcome this, we instead decided to manually create workloads that are tailored for each device. Table 4.4 provides a summary of each workload. In general, we mostly use shell scripts that randomly invoke shell commands specific to the

Kernel source	Streaming DMA		Coherent DMA				
	Affected regions	Errant accesses	Affected regions	Errant accesses	Vuln. writes	Vuln. loops	Vuln. asserts
block/*	-	-	-	-	32	2	1
drivers/ata/*	3	-	1	2	49	2	-
drivers/hid/hid-core.c	-	-	-	1	-	-	-
drivers/hid/usbhid/hid-core.c	-	-	-	-	1	-	-
drivers/net/ethernet/amd/pcnet32.c	2	-	3	10	-	-	-
drivers/net/ethernet/dec/tulip/*	3	-	1	8	-	-	-
drivers/net/ethernet/intel/e100.c	2	10	2	35	37	1	-
drivers/net/ethernet/intel/e1000/e1000_main.c	3	-	2	7	-	-	-
drivers/net/ethernet/intel/e1000e/netdev.c	-	-	2	8	-	-	-
drivers/net/ethernet/realtek/8139cp.c	-	-	1	10	-	-	-
drivers/net/vmxnet3/*	1	455	5	30	-	-	-
drivers/pci/msi/msi.c	-	4	-	-	-	-	-
drivers/scsi/*	-	-	-	-	46	-	1
drivers/tty/serial/serial_core.c	-	-	-	-	-	1	-
drivers/usb/core/*	28	-	3	-	-	-	-
drivers/usb/host/*	-	8	16	119	18	-	-
fs/ext4/*	-	19	-	-	-	-	-
fs/fs-writeback.c	-	-	-	-	2	-	-
fs/kernfs/dir.c	-	-	-	-	-	1	-
kernel/dma/*	-	4	-	-	16	3	5
kernel/printk/printk.c	-	-	-	-	2	-	-
kernel/sched/*	-	-	-	-	34	5	-
kernel/workqueue.c	-	-	-	-	4	-	-
lib/*	-	8	-	-	58	6	2
mm/dmapool.c	-	-	-	7	-	-	-
net/core/*	-	68	-	-	1	-	-
net/ipv4/*	-	-	-	-	9	-	1
net/ipv6/addrconf.c	-	-	-	-	1	-	-
security/keys/key.c	-	-	-	-	-	1	-
sound/core/*	-	-	1	-	2	-	-
sound/hda/hdac_stream.c	-	-	-	1	-	-	-
sound/pci/hda/hda_controller.c	-	-	-	2	-	-	-
sound/usb/pcm.c	-	-	-	1	-	-	-
Total	42	576	37	241	312	22	10

Table 4.3: DMA race conditions found.

DMARacer

Device kind	Num. devices	Workload
Audio card	13	Playing and recording audio files
GPU	12	Running OpenGL/GPU test software
Mouse/keyboard	36	Random input events
Network adapter	61	Handling random HTTP requests
Storage device	25	Random file system operations

Table 4.4: Tested devices and workloads.

device for about 5 minutes per device. For example, the workload for storage devices consists of a shell script that randomly manipulates files, file contents and folders on the mounted storage device. The workload for human interface devices was implemented by using a custom version of QEMU that produces random mouse, touchpad and keyboard presses. We again use QEMU's virtual device emulation feature for this and emit the same internal input events that QEMU creates when being used with a graphical frontend. The emulation backend then translates these generic QEMU input events into device-specific device-to-kernel communication. For the network devices, we start an HTTP server on the host machine and then send random HTTP requests from the QEMU guest.

4.8.2 Overall Results

Table 4.3 presents the race conditions found by DMARacer, categorized by: (i) the number of DMA regions affected by errant accesses, (ii) the number of errant accesses, and (iii) the number of vulnerable operations. We group DMA regions based on the *calltrace* of their mapping operation, as this reflects the number of unique *objects* a developer may need to address. This grouping is independent of any intermediate DMA mapping APIs (e.g., `dma_pool_alloc()`), which may ultimately perform a single low-level mapping operation (e.g., via `dma_alloc_attrs()`). Conversely, we group errant accesses and vulnerable operations based on the offending *instruction*, representing the number of unique *lines of code* that a developer would need to fix. It should be noted that the data flow from errant accesses can span multiple files, which means that the vulnerable operations in one row are not necessarily caused by the errant accesses in the same row (see Section 4.8.3).

Our first observation is that the sheer number of errant accesses (817) and vulnerable operations (344) highlights that DMA race conditions pose a significant threat. However, this alone does not fully convey the mitigation effort required: if errant accesses are confined to a *single* DMA region, the fix is relatively straightforward; but if they occur across multiple *distinct* DMA regions, remediation becomes

more challenging. By examining the number of DMA regions affected (79), we see that the problem is indeed widespread, indicating that mitigation may be difficult. Our second observation is that the largest count of taint sinks for TOCTOU/TOITOU errors are vulnerable writes. That is, the attacker has at least partial control over the address of a store instruction. As DMARacer is a dynamic analysis tool, it cannot determine all constraints enforced for each write. However, any of these found vulnerable writes is a potential arbitrary write primitive which can corrupt kernel memory and lead to privilege escalation. In Sections 4.8.3 and 4.8.4, we draw more insights by breaking down the numbers based on the source file and type of DMA.

4.8.3 Coherent DMA-based Race Conditions

Our analysis of coherent DMA-based race conditions reveals two notable findings: (i) most vulnerabilities stem from drivers managing complex data structures within DMA regions, which unfortunately makes exploitation easier and mitigation more challenging; and (ii) most dataflows to vulnerable operations occur across kernel subsystems, making them difficult to track without DMARacer's DTA-based approach. To illustrate each finding, we present case studies that share a common theme: a vulnerable high-level API, which implies that any driver that uses the API may also be vulnerable.

TOITOU-vulnerable data structures

Our analysis reveals that although just over half (57%) of the identified errant accesses are TOITOU bugs—as opposed to TOCTOU bugs—the vast majority of the resulting vulnerabilities (99%) depend on TOITOU-loaded data. Upon manual examination, we attribute this disparity to the fact that TOITOU bugs typically stem from the kernel managing critical data structures within DMA regions. Consequently, the kernel performs complex operations on these data structures, many of which can be exploited.

For example, it is common to find entire linked lists—with pointers and all—managed in DMA. We observe such DMA-resident lists throughout the kernel code, e.g.: the UHCI driver maintains a list of "queue headers"; the E100 driver maintains a list of "callbacks"; and the DMA pool API maintains a list of "blocks" (which we will discuss below). Whenever the kernel loads a pointer from these linked lists, it loads an attacker-controllable pointer. Thus, when the kernel dereferences it (e.g., to traverse the list), it performs an attacker-controllable memory access.

In contrast, TOCTOU-based bugs are typically more limited, as they involve simpler, less consequential data structures. For example, a common source of a TOCTOU bug is reading a status flag multiple times. If an attacker corrupts such a status flag, the impact is limited because it usually does not allow control over critical operations, such as the pointer used in a memory write.

Furthermore, we find that TOITOU bugs are more concerning than TOCTOU bugs for two main reasons. First, they are *easier to exploit*, as they involve the corruption of long-lived data. Conversely, exploiting TOCTOU bugs requires corrupting short-lived data within a narrow time window, forcing attackers to employ synchronization tricks to precisely time the corruption [46, 226, 228]. Second, they are more *difficult to mitigate*, as they affect critical data structures, and may therefore require extensive algorithmic rewrites of driver code. Conversely, TOCTOU bugs can often be resolved with minor, localized changes (e.g., combining two reads into a single read).

Case study: DMA pool API A striking example of a TOITOU-vulnerable linked list is in the widely used DMA pool API. A DMA pool aims to reduce the overhead of coherent DMA allocations, which are resource-intensive operations. It achieves this by creating a large coherent DMA buffer—the "pool"—from which smaller buffers are allocated as needed.

Fundamentally, a DMA pool functions like a heap: it is a structure composed of linked memory "blocks", which, in this context, are DMA buffers. When a driver employs a DMA pool, it grants the device access not only to these blocks but also to the pointers linking them. Consequently, similar to traditional heap corruption vulnerabilities—where a malicious program corrupts heap metadata to e.g., hijack control flow [10, 108]—a TOITOU bug allows a malicious device to corrupt DMA pool metadata, which can trivially lead to arbitrary kernel memory corruption from any driver that uses it, as illustrated by Figure 4.4.

Unfortunately, because the DMA pool API is extensively used, this vulnerability is not confined to a single instance. In fact, every usage of the DMA pool API is potentially vulnerable. Each of these instances could lead to arbitrary memory corruption, highlighting the critical importance of addressing this issue.

Cross-subsystem dataflows

Coherent DMA-based race conditions frequently involve dataflows across different kernel subsystems, making them challenging to detect. Typically, both the mapping of a coherent DMA region and any errant accesses to it occur within the same file, usually within the driver. However, the data loaded via DMA often propagates throughout the kernel, and any vulnerable operations that depend on this data may occur in separate files or entirely different subsystems.

This is understandable because drivers manage DMA, but the data they load can flow into other kernel components. Tracking such dataflows between disparate components is difficult for heavyweight analyses (e.g., symbolic execution) because it requires interprocedural and cross-module analysis. In contrast, DMARacer's lightweight DTA engine effectively tracks these dataflows, allowing us to identify many vulnerable operations that might otherwise be missed.

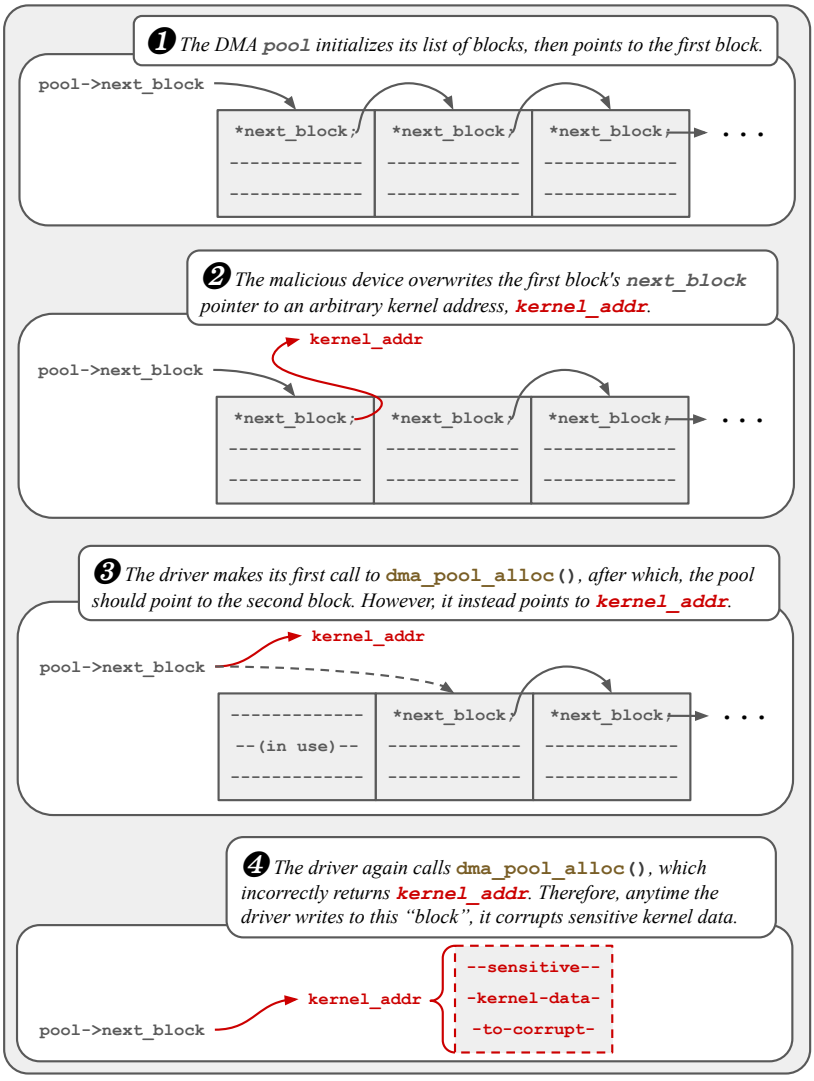


Figure 4.4: DMA pool exploitation: The DMA pool API initializes a linked list in coherent DMA, which a device can exploit to corrupt arbitrary kernel memory.

Case study: Driver-to-swiotlb dataflow An illustrative example of a cross-subsystem dataflow occurs when a driver improperly saves the bus address of a *streaming* DMA mapping into a *coherent* DMA region—thereby making it attacker-controllable—then uses it in an unmapping operation. We observed this scenario in several drivers (e.g., the Intel E100 NIC driver, RealTek 8139C+ NIC driver), and we present one such occurrence in Figure 4.5.

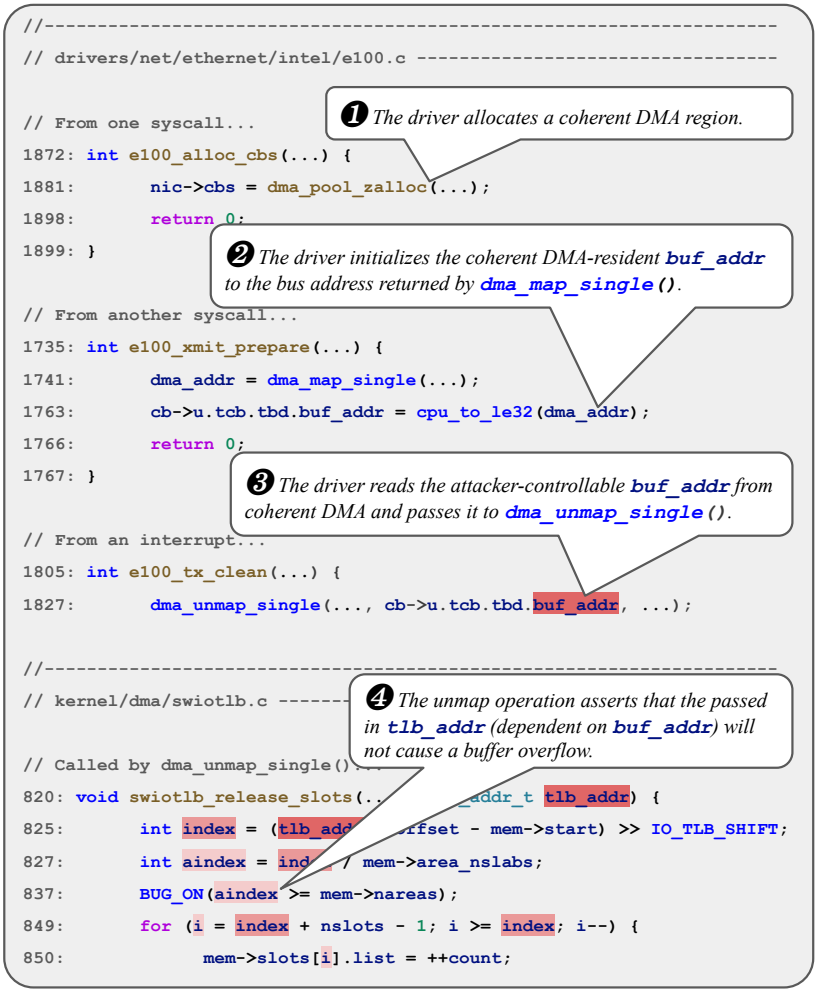


Figure 4.5: Driver-to-swiotlb exploitation: A driver saves a (mapped) bus address to coherent DMA, and later passes it to an unmapping operation with a vulnerable assertion, which a malicious device can exploit to cause a DoS.

This vulnerability is particularly challenging to detect with static analysis due to several factors. First, the three interactions with the coherent DMA region—i.e., the allocation, initialization, and usage—occur in separate syscalls and interrupts, which complicates control flow analysis and requires modeling asynchronous events. Second, the dataflow from the errant access (in `e100_tx_clean()`) to the vulnerable assertion (in `swiotlb_release_slots()`) crosses several subsystems: from the driver, to the mapping API, then to the swiotlb (i.e., "software I/O translation lookaside buffer") API. Even along this single control path, there are multiple in-

termediate indirect calls (e.g., the driver may use custom mapping operations via function pointers), which are notoriously difficult for static analysis to resolve. Consistent with this, our review of SADA's reported findings (Appendix 4.B) suggests that the vulnerable operations it flags are typically contained within a single function, whereas the bug here spans multiple functions and subsystems.

This cross-subsystem dataflow raises a crucial question: *Which subsystem is responsible for mitigation?* On one hand, the driver should not save a bus address in an attacker-controllable DMA region and expect it to remain uncorrupted. On the other hand, the swiotlb API should handle errors more gracefully than by invoking a kernel panic[4]. Given the widespread nature of such vulnerable DMA-based race conditions, we propose that mitigation efforts should occur at both ends.

4.8.4 Streaming DMA-based Race Conditions

Our analysis of streaming DMA-based errant accesses reveals two significant findings: (i) many are caused by developers seemingly misunderstanding the DMA mapping API; and (ii) many involve improper synchronization across entirely different kernel subsystems, making them difficult to detect without DMARacer's dynamic approach. Additionally, we present a case study to highlight the first finding, which also demonstrates a critical insight: a single incorrect synchronization can result in hundreds of errors.

Mapping API misuse

As explained in Section 4.6.2, an inconsistent access bug occurs when the accessed region is immediately preceded by either a device-synchronizing operation or a mapping operation. We found that among all errant accesses identified by DMARacer, only 10 were preceded by a device-synchronizing operation; the remaining 569 were preceded by a mapping operation and lacked any subsequent synchronization before the errant access. Evidence from prior work and commit history indicates that many of these issues stem from developers being unaware of the rules for DMA buffer synchronization—particularly, the assumption that accessing a streaming DMA buffer immediately after mapping it is safe. These findings align with SADA's analysis of inconsistent accesses [11]. Given the widespread use of the streaming DMA and the evident misunderstanding of it, we conclude that misuse of the API is a systemic problem.

Case study: VMXNET3 driver A striking example of DMA mapping API misuse is found in the VMXNET3 driver, a high-performance NIC driver from VMware. Figure 4.6 illustrates how a single incorrect synchronization in this driver leads to hundreds of inconsistent access bugs.

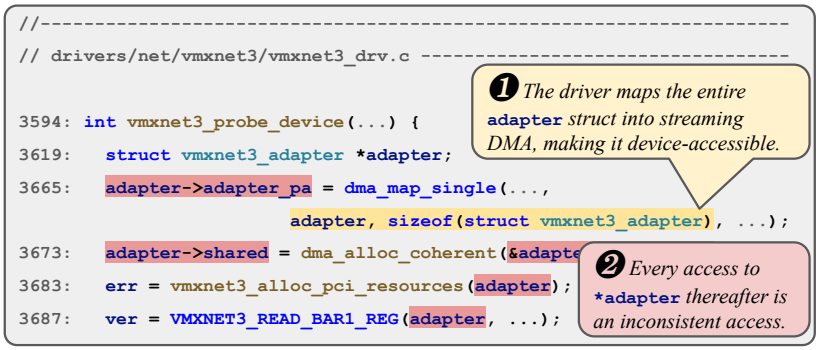


Figure 4.6: A misunderstanding of the rules regarding DMA synchronization leads to hundreds of inconsistent access in the VMXNET3 NIC driver.

During boot time, the driver initializes its data structures and maps the `struct vmxnet3_adapter *adapter` structure—which includes critical fields such as the transmit and receive queues—into a streaming DMA region. Immediately after the mapping operation, on the same line of code, it stores the returned value into the `adapter->adapter_pa` field, thereby committing an inconsistent access. This example highlights developers' apparent misunderstandings of DMA synchronization rules—the bug occurs on the same line as the mapping operation.

Subsequently, within the same function, the driver performs 52 additional inconsistent accesses to this object while initializing its various fields. Proper mitigation would involve deferring the mapping operation until after initialization and adding synchronization operations when the object transitions between kernel and device access. However, such a mitigation is non-trivial, as it requires defining synchronization points in a driver not originally designed with these considerations. We are currently collaborating with the developers to address these issue, and mitigation efforts are underway.

Cross-subsystem synchronization

Similar to the cross-subsystem dataflows observed in coherent DMA-based race conditions, streaming DMA-based race conditions often involve cross-subsystem dependencies—though in this case, the issue revolves around synchronization rather than data propagation. In streaming DMA, the mapping of the DMA region and any errant accesses often occur in different files or subsystems.

Specifically, apart from those in the VMXNET3 and E100 drivers, all errant accesses involve DMA regions that are mapped in a driver (e.g., the ATA, E1000, and USB drivers), whereas the errant access occurs in a completely different subsystem (e.g., the filesystem or networking subsystems). This is perhaps unsurprising, as drivers are typically responsible for mapping and synchronizing DMA buffers,

whereas higher layers of the stack simply access these buffers. Consequently, if a driver fails to synchronize DMA correctly, it affects accesses elsewhere in the kernel.

Because control transfers between these disparate parts of the kernel occur via indirect branches, hardware interrupts, and similar mechanisms, static analysis approaches can struggle to identify such errant accesses. In particular, SADA [11] reports that it does not analyze function pointer calls when building call graphs; consequently, it cannot construct a complete call graph and would miss the kind of cross-subsystem synchronization bugs we observe here. In contrast, DMARacer's dynamic approach effectively tracks DMA state through these control transfers, thereby identifying these synchronization issues.

4.8.5 Accuracy, Performance, & Coverage

Accuracy

We manually inspected results to estimate DMARacer's false positive (FP) rate. Unfortunately, full inspection is infeasible due to two challenges: First, some errant accesses span multiple execution contexts (e.g., across syscalls or interrupts), making complete tracing impractical. Second, due to a limited taint space (256 colors), taint labels are assigned per instruction rather than per calltrace. As a result, frequently used instructions (e.g., in `memcpy()`) may conflate independent execution paths.

Instead, we manually reviewed the 179 errant accesses and 56 vulnerable operations that we could definitively assess. We found a 0% FP rate for errant accesses and a 9% FP rate for vulnerable operations—both acceptably low. Errant accesses are precise by design, as DMARacer tracks concrete memory accesses to real DMA regions. False positives in vulnerable operations arise from not modeling dataflow constraints.

For instance, in the ALSA audio driver, the kernel reads a previously-initialized index via DMA and returns it to the PCM subsystem. While this appears attacker-controllable, the PCM layer bounds-checks the index and resets it to zero if it exceeds the buffer size. Thus, although the data is tainted, the actual access is safe—demonstrating a FP due to an unmodeled constraint. However, such cases are uncommon in practice: they require explicit checks against known-safe values, which most DMA-handling code does not perform. We further discuss false positives and false negatives in Section 4.9.

Performance

We evaluated the performance of DMARacer using the OS subset of LMBench [141] and measured a mean runtime overhead of 402% across 20 iterations. Additionally, we performed an ablation study using LMBench to understand the overhead

Enabled component	Mean		Geomean	
	Overhead	Δ	Overhead	Δ
Default Kernel (Baseline)	0%		0%	
+ KDFSan (Taint Tracking)	194%	+194%	232%	+232%
+ DMA Region Tracking	194%	≈ 0%	232%	≈ 0%
+ Load Monitoring	333%	+139%	488%	+257%
+ Store Monitoring	349%	+16%	508%	+20%
+ Compare Monitoring	400%	+51%	584%	+76%
+ Reporting (DMARacer)	401%	+1%	588%	+4%

Table 4.5: Runtime overhead of DMARacer's components.

incurred by each component. Table 4.5 shows the incremental and total overhead of each component in LMBench. According to our measurements, the taint tracking logic and load instruction monitoring are the largest contributors to runtime overhead.

In the context of other widely used sanitizers, the overhead of DMARacer is higher than AddressSanitizer [187] (73% to 100% [35]), comparable to that of MemorySanitizer [199] (between 200% to 400% [36]) and faster than ThreadSanitizer (between 400% to 1400% [37]).

Coverage

The findings of this evaluation depend on the DMA-specific code we covered in the kernel. The more DMA-related code we exercise, the more DMA race conditions we can detect. In this section, we analyze how much of the kernel's DMA code we reached with our evaluation workloads.

Table 4.6 presents: (i) the total number of DMA mapping sites in the *entire kernel*; (ii) the number of DMA mapping sites we *covered*; and (iii) the number of DMA mapping sites we covered that are *affected* by a race condition. Note that, unlike Table 4.3, which groups DMA regions by mapping *calltraces*, we count mapping *callsites* here to enable direct comparison with the total number of callsites in the kernel. Additionally, the total number of DMA mapping sites is obtained by compiling the kernel with `allyesconfig` and counting calls to DMA mapping operations in the resulting object files. As a result, this total also includes mapping sites that are potentially unreachable with the specific devices and drivers used in our evaluation setup.

Our first observation is that approximately half of the covered mapping sites (37 out of 68) are affected by a DMA race condition. This high proportion indicates that DMA race conditions are prevalent among the code we exercised. However,

Kernel source	Maps	Maps covered	Aff. maps covered
`drivers/ata/`	13	2	2
`drivers/net/`	589	48	23
`drivers/usb/`	58	16	11
`drivers/` (other dirs.)	932	1	-
`sound/core/`	3	1	1
`sound/` (other dirs.)	7	-	-
`/` (other dirs.)	29	-	-
Total	1631	68	37

Table 4.6: DMA mapping (or allocation) sites.

this figure represents an upper bound: some mapping sites may be used to map multiple DMA regions, and not all of these regions are necessarily affected—only at least one is.

Our second observation is that we covered only a fraction of the total mapping sites in the kernel (68 out of 1631). We attribute this mainly to the relatively low number of devices that can be emulated by QEMU (see also Section 4.9). Additionally, several of 147 devices we emulated with QEMU use the same driver and therefore share a mapping site. The reason for this is that they are either similar devices (e.g., the `i82557a` and `i82557b` network adapters) or follow a standard communication protocol (such as USB human interface devices).

4.9 Discussion & Limitations

Completeness DMARacer may incur false negatives for two reasons. First, since KDFSAN does not track implicit data flows, DMARacer may miss vulnerabilities that rely on them. We consider this preferable to the alternative approach of approximating implicit flows, which can be a source of over-tainting and thus false-positives [109]. Second, our taint policies aimed at reducing false positives (Section 4.6.3) may occasionally result in false negatives. For instance, a vulnerability dependent on bit-level dataflows might be missed due to our policy of clearing taint for `AND` instructions.

A possible alternative to removing taint is to use bit-precise taint tracking. This removes the need to completely clear taint after bitwise operations, and reduces false-positives caused by overtainting due to DMARacer's *byte*-accurate tainting approach.

Soundness DMARacer may also produce false positives where DMA race conditions exist, but the code explicitly defends against abuse. This is primarily because, like any DTA system, DMARacer does not fully model all dataflow constraints. For example, if a driver verifies each value it loads from DMA against an expected value, DMARacer would not capture this constraint. However, manual inspection reveals that such cases contribute to only a few false positives. Addressing this inherent limitation would require heavyweight constraint solving, which we intentionally avoided to keep our prototype scalable and practical, encouraging its real-world adoption.

Access to devices Unlike static analyzers, DMARacer requires that each driver can be executed and perform its normal communication with its respective device. This also means that the device has to be emulated or physically connected to the system. The rise of device emulation with tools such as QEMU alleviates this issue to some degree. However, creating these drivers requires in-depth understanding of how each device works, as the kernel-device communication has to be recreated exactly in the emulation layer. It is therefore possible that some devices can never be emulated by QEMU, and therefore DMARacer is unable to find DMA errors in their respective drivers unless the physical device is available.

Comparison to static analysis Compared to a static analysis approaches such as SADA [11], DMARacer can detect DMA misuse across long execution traces. DMARacer can also analyze traces that involve asynchronous completions, indirect function calls or control flow based on hardware-triggers (see Sections 4.8.3 and 4.8.3).

4.10 Related Work

DMA errors Previous work investigated the security implications of communicating with peripheral devices using DMA. [140] and [7] characterize possible DMA attack scenarios in the presence of various IOMMU protections. This also includes the accidental exposure of OS-internal data to the device, which is a superset of the TOITOU bugs that are detected by DMARacer. SADA [11] proposes using static analysis to search code for various DMA bugs, which includes the TOCTOU bugs and other errant accesses detectable by DMARacer. Other work is concerned with dynamic detection of race conditions in the DMA memory controller itself [116, 176]. These race conditions are a superset of the ones detectable by DMARacer.

Double fetches Existing work has studied double fetches and proposed detection approaches by e.g., using static analysis [131, 221, 222, 231]. Other approaches are based on dynamic analysis of memory accesses. DECAF [183] is a dynamic detection

tool using modern CPU features and hardware side-channels to detect memory accesses that indicate double fetch bugs. Bochspwn [107] is a dynamic detection tool that simulates the target OS and searches the simulated memory accesses for indications of double fetches. It should be noted that the attack vector studied by previous work on double fetches is mainly concerned with the system call interface. There are no DMA-based memory accesses in this scenario and instead the memory is manipulated by a malicious thread running in user space. However, the general issue of double fetching from untrusted memory is the same in DMA-based drivers and system call handlers. The mitigations proposed for double fetches in system handlers [16, 63] are in theory also applicable to the DMA interface in drivers. However, whether their overhead is practical in the context of DMA-based device drivers is unclear.

Race conditions The problem of detecting race conditions in concurrent programs has a significant body of research covering it. This includes approaches based on static [65, 164] and dynamic analysis [72, 178, 188] which verify the proper use of synchronization primitives within a program. However, there are no viable synchronization primitives when interacting with coherent DMA, so the applicability of these approaches to DMA data races is limited.

Kernel race detectors Other previous work focuses on detecting data races within OS kernels [66, 206] by observing values read from memory locations via memory watchpoints. The concurrent read of different values from the same address is here used as an indication of a race condition. Because these tools do not rely on observing synchronization primitives, they can detect when peripheral devices and the kernel concurrently access memory via DMA. However, as the information is limited to a single point in time within a thread's execution, this information is not suitable to detect TOCTOU errors.

Kernel fuzzing Sanitizers like DMARacer provide reliable bug detection for fuzzers which drive the dynamic analysis. Previous work has proposed a wide variety of solutions for performing random testing on device driver code. This includes fuzzing approaches specific to USB [101, 161, 233], mobile devices [165, 204] or peripherals using DMA [142]. Other fuzzing methods improve the generating coverage by more accurately simulating the real interface behavior of peripheral devices [42, 69, 136, 241].

4.11 Conclusion

Modern OS kernels use direct memory access (DMA) to efficiently communicate with untrusted peripheral devices. However, when DMA is used incorrectly, it can lead to potentially dangerous race conditions within the kernel. In this chapter, we propose a dynamic detection approach named DMARacer that detects race conditions caused by the incorrect usage of DMA buffers in kernel drivers. DMARacer tracks all DMA-based memory accesses and checks them for various kinds of DMA-specific race conditions. DMARacer can also estimate the security impact of the detected bugs. This is done by taint tracking the data from problematic memory reads and pinpointing vulnerable code that operates on this data. We applied DMARacer to the drivers in the Linux kernel and found 817 problematic memory accesses and 344 vulnerable operations. This suggests that DMA-based race conditions in driver code are a systemic issue.

Appendix 4.A LMBench Performance Data

Table 4.7 shows the per-benchmark LMBench results of DMARacer compared to an unsanitized kernel.

Appendix 4.B SADA Findings

There is no publicly available code or artifact of SADA, so we could not run a direct comparison against DMARacer. Instead, we provide an analysis of the issues reported by SADA. We found these issues by searching the Linux kernel mailing list for reports and patches that are referencing the paper's authors. The list of found issues is to our knowledge complete except for privately reported and unpatched bugs.

Table 4.8 shows the 20 patches/reports we found. Some reports describe multiple issues affecting several DMA memory regions. We indicated if this is the case using the third table column. In total, 24 unique bugs were reported that involve a problematic DMA memory access.

The reports do not contain the actual trace that SADA analyzed. We therefore approximated the detection logic using the bug explanation in each report. Specifically, we tried to determine how often SADA finds issues involving long and complex traces.

Column 4 shows whether the vulnerable DMA operations that needed to be analyzed occurred in the same function. For a bad streaming DMA mapping, the involved code piece are the to-device mapping operation and the subsequent memory access. For a TOCTOU error, the vulnerable operations are the check and use. Unchecked DMA access only involve one read operation, so we omitted them in this

Benchmark	Baseline time	DMARacer		
		Time	Overhead	Increase
Simple syscall	0.25μs	0.84μs	0.59μs	+234%
Simple read	0.32μs	1.60μs	1.27μs	+392%
Simple write	0.32μs	1.31μs	0.99μs	+315%
Simple stat	0.62μs	5.58μs	4.96μs	+797%
Simple fstat	0.37μs	1.67μs	1.30μs	+353%
Simple open/close	1.21μs	10.75μs	9.54μs	+788%
Select on 10 fd's	0.41μs	1.76μs	1.35μs	+331%
Select on 100 fd's	1.36μs	13.81μs	12.45μs	+917%
Select on 250 fd's	2.90μs	33.53μs	30.63μs	+1055%
Select on 500 fd's	5.53μs	66.52μs	61.00μs	+1104%
Select on 10 tcp fd's	0.49μs	2.41μs	1.92μs	+393%
Select on 100 tcp fd's	3.72μs	40.58μs	36.85μs	+990%
Select on 250 tcp fd's	9.00μs	104.91μs	95.91μs	+1066%
Select on 500 tcp fd's	18.07μs	212.22μs	194.15μs	+1074%
Signal handler installation	0.31μs	1.02μs	0.71μs	+227%
Signal handler overhead	0.76μs	4.95μs	4.19μs	+548%
Protection fault	0.47μs	0.92μs	0.45μs	+96%
Pipe latency	2.61μs	24.31μs	21.70μs	+831%
UNIX sock stream latency	3.17μs	29.94μs	26.77μs	+843%
Process fork+exit	69.86μs	354.08μs	284.22μs	+407%
Process fork+execve	215.01μs	961.15μs	746.14μs	+347%
Process fork+/bin/sh -c	465.10μs	1919.48μs	1454.38μs	+313%
UDP latency	4.05μs	52.10μs	48.05μs	+1187%
TCP latency	5.40μs	87.20μs	81.80μs	+1514%
TCP/IP connection cost	21.18μs	238.17μs	216.99μs	+1024%

Table 4.7: Runtime overhead of DMARacer.

column. In summary, all found issues had their vulnerable operations contained within a single function. If we consider all DMA operations including the DMA allocation call itself, then 12 of the 20 reports (16 of 24 bugs) have traces that only spanned a single function (Column 5).

The last column indicates whether a patch has been merged or the respective bug was fixed by a maintainer. In total, 10 reports containing 14 bugs were fixed.

Subject line on mailing list	Issue class	Num. DMA regions	Single func. contains...		Patch merged?
			Vuln. ops.?	All ops.?	
rtlwifi: rtl8723ae: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
rtlwifi: rtl8192de: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
rtlwifi: rtl8192ce: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
rtlwifi: rtl8188ee: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
p54: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
atm: idt77252: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
atm: eni: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✓
net: vmxnet3: avoid accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✗
crypto: hisilicon: accessing the data mapped to streaming DMA	Incons.	4	✓	✓	✓
crypto: qat: accessing the data mapped to streaming DMA	Incons.	2	✓	✓	✗
net: rocker: accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✗
scsi: wd719x: accessing the data mapped to streaming DMA	Incons.	1	✓	✓	✗
media: venus: fix possible buffer overlow casued bad DMA value in venus_sfr_print()	TOCTOU	1	✓	✗	✗
media: pci: ttpci: av7110: avoid compiler optimization of reading data[0] in debiirq()	TOCTOU	1	✓	✗	✓
scsi: esas2r: fix possible buffer overflow caused by bad DMA value in...	TOCTOU	1	✓	✗	✗
net: sfc: fix possible buffer overflow caused by bad DMA value in efx_siena_sriov_vfdi()	TOCTOU	1	✓	✗	✗
usb: cdns3: fix possible buffer overflow caused by bad DMA value	Val. Check	1	—	✗	✗
input: tablet: aiptek: fix possible buffer overflow caused by bad DMA value in aiptek_irq()	Val. Check	1	—	✗	✗
usb: storage: alauda: fix possible buffer overflow casued by bad DMA value in...	Val. Check	1	—	✗	✗
net: vmxnet3: fix possible buffer overflow caused by bad DMA value in vmxnet3_get_rss()	Val. Check	1	—	✗	✓

Table 4.8: DMA errors detected by SADA. In column two, “Incons.” denotes a bad streaming DMA mapping, and “Val. Check” a missing DMA value check.

Appendix 4.C Artifact Appendix

4.C.1 Abstract

This artifact contains the LLVM-based compiler and Linux Kernel fork of DMARacer. Additionally, we provide the evaluation scripts as well as the QEMU fork needed for the experiments in the paper.

4.C.2 Description & Requirements

Security, privacy, and ethical concerns

The evaluation runs fuzzing scripts which could consume excessive CPU, memory or disk space resources.

How to access

The artifact is available at `https://zenodo.org/records/17010802`. The source code repository is also available at `https://github.com/vusec/DMARacer`.

Hardware dependencies

This artifact requires an x86-64 CPU as the implementation is specific to this architecture. It also requires at least 16GiB of available RAM which is what the fuzzing script assign each QEMU instance. All source and build artifacts combined can use up to 35 GiB of available disk space.

Software dependencies

The artifact is within a Docker container that contains all software dependencies. You only need to have Python 3 and Docker (`https://docs.docker.com/engine/install/`) installed on your host system.

Note: Your host system still has to be a Linux distribution, due to the low-level nature of the experiments. You also should be in the `docker` group, so the scripts can access Docker without superuser privileges (see `https://docs.docker.com/engine/install/linux-postinstall/` for details).

Benchmarks

This artifact does **not** require any manual provisioning of benchmarks.

Note that we use `lmbench` for measuring OS performance. This benchmark was originally created in 1996 and no longer compiles with a modern C compiler. The evaluation setup will automatically download a fixed version from `https://github.com/vusec/lm-bench-fixed`. This file is already downloaded in the artifact.

4.C.3 Set Up

Installation

The artifact runs within a Docker container. You can start the docker container by running the following command in the root directory of the artifact:

```
$ ./start
```

This script will open a shell within docker and all further commands should be executed within this shell. Note that the host and the docker container share the base repository folder, and you can edit files within the container's ~/mnt folder by editing the contents of the artifact folder.

Your build files are stored in this shared folder, so they are **not** lost when you close the docker shell. You can freely close and open the shell without losing build files, but it will terminate any running experiments or build processes. You can also open multiple shells in the same container at the same time and access the same files. However, the artifact does not support running any of the experiments concurrently.

> **Note:** If `./start` fails due to a permission error or being unable to connect to the docker socket, you are probably not in the `docker` group on your system. See Section 4.C.2. If you are unable to add yourself to the docker group, but you have sudo access, you can try using `./start --sudo` as a workaround.

> ! **All commands** except `./start` **must be run within the docker shell** opened by `./start`! The correct shell to run the commands in has a prompt starting with `dmaracer@DMARacer`, and we prefix all commands with this prompt for clarity.

Once you are within the docker shell, run the following command to build all parts of DMARacer.

```
dmaracer@DMARacer$ task build
```

This will build the modified version of the Linux Kernel, DMARacer's implementation of LLVM and QEMU, as well as a bootable Debian version using this kernel. The full build can take up to 24 hours depending on your hardware.

Mounting errors There is a chance the build will fail with the error shown below when trying to mount the built Linux image. This seems to an issue on some host systems, and the exact cause is unclear to us. However, the problem is usually resolved by restarting the Docker container.

```
+ sudo mkdir -p /mnt/bullseye
+ sudo mount -o loop bullseye.img /mnt/bullseye
mount:  /mnt/bullseye:  failed to setup loop device for
/home/dmaracer/mnt/out/syzkaller-image/bullseye.img.
task:  Failed to run task "syzkaller:create-image":  exit status
32
task:  Failed to run task "build":  exit status 201
```

> **Note:** You should have at least 1 GiB of free RAM per logical core for the build process. If that is not the case, a compilation or link step might fail/crash due your system being low on memory. If you do not have enough available RAM, you can lower the number of build jobs by modifying the `NPROC` line in `Taskfile.yml`. E.g., if you only have 4 GiB of free RAM, change the line to the following: `NPROC: { sh:  echo 4 }`

Basic test

You can perform a simple full system test by running the following command.

```
dmaracer@DMARacer$ task qemu:test
```

The expected output will contain a lot of test reports. The tests are all successful if the output ends with the following lines.

```
KDFSan: Cleaning up streaming DMA test...
KDFSan: KDFSan policies tests complete (NNNN cycles elapsed)
KDFSan: Post boot done.
[...]
 expect successful!
```

4.C.4 Evaluation Workflow

Major claims

(C1): DMARacer finds 817 errant accesses in the tested Linux Kernel. This is proven by experiment (E1) in Section 8.1 of the paper, and the detailed results are shown in Table 3 ("DMA race conditions found") of the paper. Because the results are based on a fuzzing campaign, these numbers can be slightly higher or lower due to the randomness of the testing process.

(C2): These errant accesses lead to the three case studies presented in the paper (See Sections 8.3.1, 8.3.2, and 8.4.1 in the paper). These results are filtered from the data of experiment E1, and we refer to this as experiment E2 in the next section.

(C3): DMARacer has an overhead of 401% as shown by Section 8.5.2 and Table 5 ("Runtime Overhead of DMARacer's components") in the paper. This is shown using experiment E3.

Experiments

(E1): Fuzzing Campaign [10 person-minutes + 12 compute-hours]: Performs fuzzing of all supported devices.
Preparation: Start the docker container (Section 4.C.3).
Execution: Run the following command to start the fuzzing campaign.

```
dmaracer@DMARacer$ task qemu:fuzz-all
```

Note that the campaign will sometimes show errors in the following form.

```
Error:  Fuzzing campaign for audio device XXX finished before it
was supposed to!
```

You can ignore these errors as they just indicate that a device driver has crashed or failed to load during the fuzzing campaign. Not all devices are fully supported in QEMU, so this is expected behavior. However, if you see this error for **every** device, then your fuzzing setup is broken.
Once the fuzzing campaign is done, you can collect all reports and turn them into a human-readable format using the following command.

```
dmaracer@DMARacer$ task reports:load-current
```

This command will print various statistics and the output should end with the following lines. The exact values in each line might be slightly different due to fuzzing randomness.

```
[...]
virtio_sound_pci 16 18   2   1   1  - 15
virtio_tablet_pci  14 16   3   1   1  - 15
virtio_vga       14 16   2   1   1  - 15
vmware_svga      14 16   3   1   1  - 15
vmxnet3   24 34 539   1   1  - 15
```

> **Note:** You can ignore all warnings/errors referencing `mongo dump: not found`, `SyntaxWarning`, and warnings that reference `Discarding a part of backtrace [...]`.

Results: Once the campaign and report collection is done, you can print out the textual representation of Table 3 in the paper using the following command.

```
dmaracer@DMARacer$ task results-table
```

The sum of the two errant access columns should be 817, with each of the two columns listing 576 and 241 errant accesses. Because the results are based on a fuzzing campaign, these numbers can be slightly higher or lower due to the randomness of the testing process.

(E2): Case Studies [10 person-minutes + 1 compute-hour]:
Preparation: Run experiment E1 before starting this experiment.
Execution: You can view all bug reports in detail using the following command. You will see about 7500 reports and the exact number is also subject to fuzzing noise.

```
dmaracer@DMARacer$ task reports:inspect
```

The case studies can be found in this (very long) list of reports. You can navigate all reports using the control options described at the bottom of the report viewer. You can leave the bug viewer using `Ctrl+C`.
To avoid having to search all several thousand reports, we implemented shortcuts that should filter out all reports except the ones needed for each case study. We describe these shortcuts and the expected results below.

DMA pool API case study The following command will show the report for the case study described in Section 4.8.3.

```
dmaracer@DMARacer$ task case-study-dmapool
```

The resulting bug viewer should contain an information about the found bug location as shown below.

```
Next VULN report IDs:          []
RIP: dfs$uhci_start+0x845/0x1830
File: drivers/usb/host/uhci-hcd.c
Line:
 [...]/drivers/usb/host/uhci-q.c:255:17 -- uhci_alloc_qh
```

You can open that respective file and should see that the found issue is coming from an allocation created by the DMA pool (i.e., by a function that is called `dma_pool_zalloc`). See below for an example of a reported line and the nearby allocation site.

```
// DMA pool alloc site.
qh = dma_pool_zalloc(uhci->qh_pool, GFP_ATOMIC, &dma_handle);
if (!qh)
  return NULL;
qh->dma_handle = dma_handle; // Reported Line.
```

swiotlb dataflow case study The following command will show the report for the case study described in Section 4.8.3.

```
dmaracer@DMARacer$ task case-study-swiotlb
```

The results should contain reports coming from `swiotlb`. We provide an example report below. Note that one of the reports should reference the `e100` driver (in `e100.c`), which is shown in the paper as part of Figure 5. In the example below, the driver file `e100.c` is referenced in frame 4 and 5.

```
Next report IDs:               []
Next VULN report IDs:          []
RIP: dfs$swiotlb_tbl_unmap_single+0x3d4/0xa10
File: kernel/dma/swiotlb.c
Line:
  [...]swiotlb.c:837:2 -- swiotlb_release_slots
  [...]swiotlb.c:883:2 -- dfs$swiotlb_tbl_unmap_single
Backtrace:
  9. [...]dma/swiotlb.c:837:2 -- swiotlb_release_slots
  8. [...]dma/swiotlb.c:883:2 -- dfs$swiotlb_tbl_unmap_single
  7. [...]dma/direct.h:124:3  -- dma_direct_unmap_page
  6. [...]dma/mapping.c:182:3 -- dfs$dma_unmap_page_attrs
  5. [...]e100.c:1831:26 -- dfs$e100_tx_clean
  4. [...]e100.c:2228:16 -- dfs$e100_poll
  3. [...]dev.c:6464:10  -- dfs$__napi_poll
  2. [...]dev.c:6531:9   -- napi_poll
  1. [...]dev.c:6664:13  -- dfs$net_rx_action
  0. [...]softirq.c:553:3  -- dfs$__do_softirq
```

VMXNET3 case study The following command will show to the report for the case study described in Section 4.8.4.

```
dmaracer@DMARacer$ task case-study-vmxnet
```

You have to enter 'd' in the bug viewer and press ENTER to show the respective allocation site. The backtrace for the allocation should point out the lines below. This matches the code locations of Figure 6 in the paper.

```
Region alloc backtrace:
24.  [...]/linux/dma-mapping.h:424:9 -- dma_alloc_coherent
23.  [...]/vmxnet3/vmxnet3_drv.c:3673:20 -- (inlined by)
dfs$vmxnet3_probe_device
```

Results: Reports for all three case studies should be found using the described steps.

(E3): Performance Evaluation [10 person-minute + about 100 compute-hours]:
Preparation: Start the docker container (Section 4.C.3).
Execution: You can run the full benchmark using the following command.

```
dmaracer@DMARacer$ task full-benchmark
```

This script will rebuild the kernel for each ablation layer and run the `lm-bench` benchmarks.

> **Note:** The benchmark will pause for a few minutes when syncing files to the QEMU guest and the last output before this process is the error: `mount: /sys/kernel/debug: none already mounted or mount point busy.` This is **not** an actual error and just a side effect of the ablation study disabling various DMARacer functionality on the lower layers.

Results: Once the command is done, the benchmark results should be printed in the form of a LaTeX table to the console. We list one possible output below. The numbers should resemble the numbers found in Table 5 of the paper. They can slightly differ depending on whether the used machine has other workloads running or different hardware specifications.

```
################ ABLATION ###################
+ KDFSan (Taint Tracking)& 195\% & +195\% & 231\% & +231\%\\
+ DMA Region Tracking    & 195\% & 0\% & 232\% & +1\%\\
+ Load Monitoring        & 334\% & +139\% & 489\% & +257\%\\
+ Store Monitoring       & 349\% & +15\%  & 508\% & +19\%\\
+ Compare Monitoring     & 400\% & +51\%  & 585\% & +77\%\\
+ Reporting (DMARacer)   & 402\% & +2\%   & 589\% & +4\%\
```

Note that if you accidentially cleared the console, you can redisplay this table without having to rerun all benchmarks using the command below.

```
dmaracer@DMARacer$ task ablation-table
```

! If you abort the benchmark, you will have a kernel build that only has some layers of DMARacer active. If you need to revisit experiment E1/E2 after aborting the benchmark, you need to run `task rebuild-dmaracer-kernel` before. Without this, you are running an incomplete version of DMARacer.

4.C.5 Notes on Reusability

In this section, we give an overview where certain components are implemented and how to extend them.

Tasks

The `task` commands are implemented using Task (`https://taskfile.dev/`). You can see and modify the shell commands for each task by looking at the `Taskfile.yml` in the root folder and the other files in the `taskfiles/` folder.

Runtime

The implementation of the DMARacer's kernel runtime can be found in the folder `kdfsan-df-linux/mm/kdfsan/`. This runtime manages the actual taint labels, the error reporting as well as the tracking of DMA regions. It also maintains the shadow map that is used by the taint propagation to efficiently propagate taint.

Compiler instrumentation

The compiler instrumentation logic of DMARacer can be found in the `DataFlowSanitizer.cpp` file in the following folder:
`kdfsan-llvm-project/llvm/lib/Transforms/Instrumentation`
It mainly injects the taint propagation logic, but also the checking for taint and injects callback for memory accesses. To add a new taint sink, the function `addEdgeCallback` is a good template.

Fuzzing

The fuzzing scripts can be found in the folder `scripts/fuzzing/`. The subfolders have the scripts that run **within QEMU** to create fuzzing workloads.

New devices can be added by adding their respective QEMU arguments that emulate it to `scripts/fuzzing/devs.py`. The device type lists which fuzzing workload should be executed to test the device. The type matches the name of the subfolder with the workload script. The fuzzing campaign task `task qemu:fuzz-all` will automatically fuzz it and the reports viewer will show the found issues.

Fuzzing workloads can be updated/changed by updating the `run.sh` in the respective subfolder of `scripts/fuzzing/`. The fuzzing scripts often call existing programs that exercise drivers (e.g., a desktop environment so mouse/touchpad inputs are processed). To install more programs into the QEMU image, the list of Debian packages in `taskfiles/TasksSyzkaller.yml` can be updated. Afterwards, the QEMU image has to be rebuilt with this command.

```
dmaracer@DMARacer$ syzkaller:create-image
```

Ablation study

The ablation study is implemented as a series of patches. Each patch is applied and then the kernel is rebuild and benchmarked. The patch for each layer can be found in the folder `ablation-patches/`.

Kernel update

DMARacer should work with a newer Linux Kernel version as long as the DMARacer-runtime and some other minor changes are ported. The runtime itself is for the most part isolated in `kdfsan-df-linux/mm/kdfsan/`. There are some changes that are in the rest of the kernel source that may lead to conflicts when porting DMARacer.

- DMA hooks which notify DMARacer of DMA mappings (e.g., `kernel/dma/mapping.c`).
- Some low-level operations that copy memory and which need to be updated to retain taint
  (e.g., `arch/x86/include/asm/page.h`)
- Some parts of the kernel cannot be instrumented with a sanitizer for various reasons. These can usually be identified because they are already handled specially with the existing KMSan sanitizer. The same workarounds that are needed for KMSan usually need to be also copied for DMARacer. See `arch/x86/include/asm/thread_info.h` for an example.

4.C.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926.

Author and Contributor Statement

Conceptualization, Software, Validation, Investigation, Writing - Original Draft: Brian Johannesmeyer, Raphael Isemann; *Visualization*: Brian Johannesmeyer; *Methodology, Writing - Review & Editing*: Brian Johannesmeyer, Raphael Isemann, Cristiano Giuffrida, Herbert Bos; *Supervision*: Cristiano Giuffrida, Herbert Bos.

5 | Conclusion

Modern attacks frequently involve difficult-to-track dataflows—e.g., dataflows that cross nonstandard interfaces, proceed in multiple stages, and exploit quirks in both the hardware and software. Therefore, this dissertation argues that *explicit vulnerability modeling* is a natural evolution of Dynamic Taint Analysis (DTA), as it allows security engineers to identify and mitigate these vulnerabilities, which traditional "source-to-sink" DTA approaches struggle to address.

In this chapter, we will justify our argument by addressing the Research Questions that we posed in Chapter 1. Then, we will discuss promising future directions that are possible with this new approach to DTA.

5.1 Revisiting Research Questions

In this section, we will support our argument that explicit vulnerability modeling can tackle modern vulnerabilities (RQ1), while maintaining a practical overhead (RQ2) and acceptable false positive/negative rates (RQ3), and can generalize to future vulnerabilities (RQ4). Specifically, we will illustrate how our three main case studies—Kasper, Einstein, and DMARacer—demonstrate the feasibility of these goals. Each of the following subsections discusses one research question and explains the role of our prototype systems in addressing it.

5.1.1 RQ1: Systematically Modeling Modern Vulnerabilities with DTA

Our solution for modeling modern exploits is to first decompose them into their essential sub-steps, then construct specialized DTA policies for each step.

Transient execution gadgets We decomposed transient execution attacks into their three essential steps in Figure 2.1, where an attacker: (i) injects data into transient execution, (ii) forces the execution to access a secret, and (iii) forces the execution to leak the secret.

Each of these steps can be accomplished by exploiting a variety of software or hardware bugs. For example, step (i) can be accomplished either through direct attacker data (e.g., the kernel not checking data copied from userspace), memory-massaging, or LVI. Moreover, step (ii) can be accomplished via a variety of transient memory errors (e.g., out-of-bounds and use-after-free accesses). Lastly, step (iii) can be accomplished via cache interference, MDS, and port contention. In Figure 2.3, we summarized KASPER's taint policies, which model these bugs and track the flow of data from "attacker injection" to "secret leakage".

Data-only attack chains We decomposed data-only attacks into their essential steps in Figure 3.1a, where an attacker: (i) exploits a memory write vulnerability, (ii) steers the execution to a gadget's entry point, and (iii) executes some payload via a gadget chain.

Each of these steps can be accomplished in a variety of ways, so we deliberately designed EINSTEIN to generalize each step, and therefore generate diverse data-only attacks. For example, for step (i), we target an *arbitrary* memory write bug by tainting all possible attacker-corruptible data. Next, for step (ii), we send *arbitrary* workloads to the victim, so it covers a range of gadgets all on its own. Finally, for step (iii), we employ taint policies, as summarized in Table 3.2, to chain attacker state across *arbitrary* syscalls.

Vulnerable DMA race conditions We defined three possible race conditions that can arise out of improper DMA usage in Section 4.4. These consist of: (i) classic time-of-check to time-of-use races, (ii) time-of-*initialization* to time-of-use races (a novel kind of race condition), and (iii) inconsistent streaming DMA accesses.

Because these race conditions can manifest in a variety of ways, DMARACER checks for various runtime invariants that, if violated, indicate the existence of a race condition, thereby making the identification of such issues a tractable problem. Then, as summarized in Table 4.1, it employs taint policies to track attacker data from the race condition to any dependent vulnerable operations.

In summary, we found that a purely DTA-based method can indeed represent the complicated logic of three vastly different classes of exploits. This suggests that the relative lack of DTA solutions for modern vulnerabilities does not reflect a fundamental limitation—just a shortcoming in policy design.

5.1.2 RQ2: Tracking Data Without Prohibitive Overhead

Recall from Section 1.3 that "prohibitive overhead" refers to overhead or constraints so severe that the analysis becomes impractical—whether due to crashing, restricting coverage, or inaccurately tracking dataflows. With that in mind, we address performance from multiple angles:

Low-overhead instrumentation KASPER and DMARACER use an LLVM-based approach, limiting overhead by hooking only the minimal set of relevant instructions in an optimized program—thereby avoiding the cost of full-system emulation [166].

Low-complexity analysis In Section 3.5.3, we introduced the concept of "identity dataflows"—i.e., dataflows where the data at the (untrusted) source is identical to the data at the (sensitive) sink. By targeting such dataflows, EINSTEIN demonstrates that heavyweight constraint solving [92, 100, 163, 182] is not necessary for automatically building powerful exploits, as there is plenty of simple, "low-hanging fruit" vulnerabilities at an attacker's disposal.

Taint engine optimizations In Section 3.5.1, we discussed the novel optimizations that we applied to EINSTEIN's taint engine. In particular, we used:

- *Constrained shadow memory* to enable fast shadow memory accesses (unlike a page table-based approach [111, 112, 207, 214]), while maintaining memory-efficiency (unlike typical shadow memory [56, 199]).
- *Hardware-assisted initialization* based on Linux's `userfaultfd` mechanism to enable fast hardware checks (unlike slow software-based checks [198, 214]), while tainting unbounded data (unlike previous fast checks [56, 112, 187, 199, 207]).
- *Dynamic array-based tags* to enable fast tag-combining (unlike an unbounded set-based tag [198, 214]), while supporting unbounded taint colors (unlike a static array-based tag [56, 111, 199, 207]).

Overall, our prototypes ran on large real-world codebases (e.g., the Linux kernel) under typical workloads, with overhead generally manageable for security testing scenarios. Still, overhead is context-dependent: for example, in a heavily simulated environment (e.g., a cycle-accurate CPU simulator), the cost of DTA might be overshadowed by the simulator's baseline overhead, whereas in performance-critical settings (e.g., kernel-based network processing), optimizations such as ours become necessary to keep runtime overhead within acceptable bounds.

5.1.3 RQ3: Reducing False Negatives and False Positives

By explicitly modeling each attack step—rather than employing a simple catch-all policy—we effectively identified more vulnerabilities (reduced FNs) and reduced false alarms (reduced FPs).

Reducing false negatives Our approach reduces FNs relative to a non-stepwise approach primarily due to its ability to *detect novel attack variants*. In particular, by modeling the attacks steps abstractly, we outperform tools that tailor their analysis to a specific attack. For example:

- Before KASPER, previous work [153, 166] tailored their analyses specifically to the Spectre "bounds check bypass" pattern, and, as we demonstrated, this leads to 40–60% FNs. In contrast, because we decompose transient execution attacks into their essential steps, KASPER easily models different combinations of vulnerabilities at each step (e.g., LVI, SMoTherSpectre, etc.), thereby avoiding these FNs.
- Before EINSTEIN, previous worked [92, 94, 163] tailored their analyses to specific, concrete memory write bugs. In contrast, because we define data-only attacks as stemming from *arbitrary* memory write bugs, EINSTEIN builds data-only attacks even for yet-undiscovered bugs.
- Before DMARACER, previous work [11] analyzed DMA accesses with only a vague notion of how they may interact with other DMA accesses. In contrast, because we explicitly analyze all DMA accesses collectively, DMARACER uncovers an entirely new class of vulnerabilities: DMA race conditions.

Reducing false positives Moreover, by employing policies around each precise attack step, our approach reduces FPs relative to non-stepwise approaches because it allows us to avoid the pitfalls of: (i) *pattern-matching*, and (ii) *taint explosion*.

First, our approach advances beyond pattern-matching. For example, previous work to KASPER identifies transient execution gadget *patterns* such as speculative out-of-bounds accesses [153], leading to 99% FPs. Moreover, previous work to EINSTEIN identifies data-only attack *patterns* such as attacker-controllable loop conditions [94], also leading to FPs. In contrast, because we actually identify each necessary component of an attack, we avoid such sources of FPs.

Second, our approach allows us to employ targeted "taint clear" policies around different attacks steps, thereby avoiding FPs due to taint explosion. Such policies allow us to: (i) model a program pacifying attacker data and declassifying secret data, and (ii) filter out dataflows that are challenging to exploit. For these reasons, KASPER washes taint: (i) at the `array_index_nospec()` macro, and (ii) between

testcases. Moreover, DMARacer washes taint: (i) at AND instructions, (ii) kernel hotpaths, and (iii) upon execution context entry. As a result, we effectively reduce FPs in a way that non-stepwise approaches cannot.

Limitations However, throughout our analyses (i.e., in Sections 2.10, 3.9, and 4.9) we acknowledge the limitations of DTA, which can be distilled into its inability to: (i) *solve dataflow constraints*, and (ii) *achieve perfect coverage*.

First, while DTA can indeed track dataflows, it cannot solve dataflow constraints. As a result, it incurs both FNs and FPs. FNs, because it cannot model all types of vulnerabilities, such as those in a cryptographic algorithm. And FPs, because it cannot model vulnerabilities with perfect accuracy. For example, Kasper cannot easily determine whether a gadget can leak only a *one-byte* secret versus *arbitrary* secrets. Therefore, in Section 5.2, we list ways that our approach can be integrated with techniques such as symbolic execution, which can indeed solve dataflow constraints.

Second, unlike static analysis, dynamic analysis (e.g., DTA) struggles with covering all possible program states, thereby incurring FNs. For example, it is non-trivial for a pure DTA solution to execute *all code* and track *all implicit dataflows*. Therefore, in Section 5.2, we list ways that our approach can be integrated with techniques such as directed fuzzing and fault injection to improve its coverage.

5.1.4 RQ4: Generalizing to Future Exploit Techniques

Just as we have shown in this thesis for current state-of-the-art exploits, we argue that future exploits may similarly be modeled by DTA. Specifically, by:

- *Using the same generic approach*—i.e., decomposing new exploits into their essential steps—we can model next-generation hardware or software vulnerabilities. In Section 5.2, we list examples of potential next-generation exploits that may benefit from DTA-based identification.
- *Applying similar optimizations*—e.g., to the taint engine or the policies—we can improve the scalability of future analyses.
- *Providing similar advantages*—e.g., detecting novel variants, avoiding taint explosion—we can enable the practical adoption of future tools.

In conclusion, explicit vulnerability-centric modeling offers an extensible and relatively future-proof approach, provided we continue refining its performance and making use of domain-specific knowledge.

5.2 Future Directions

While our approach to DTA identifies state-of-the-art vulnerabilities, we see several paths for extension.

5.2.1 Integration with other Techniques

Previous work has integrated traditional DTA with program analysis techniques such as fuzzing [87, 171, 200], symbolic execution [87, 200, 223], and fault injection [83, 85]. Similarly, by combining our stepwise DTA with such techniques, we can improve its coverage and accuracy.

Fuzzing We can improve our coverage with fuzzing by either: (i) making our taint policies *fuzzing-aware*, or (ii) making our fuzzing workloads *taint-aware*.

In particular, for the first case, we can add fuzzing sources directly into the different vulnerability steps. For example, currently, when KASPER detects an invalid speculative load (e.g., an LVI- or memory massaged-load), it simply loads the data in the KASAN redzone (e.g., `0xffff`). However, instead, by fuzzing the loaded value (e.g., by loading `0x0`, or `0x1234`, etc.), we can improve coverage at this attack step.

Second, we can direct our fuzzing campaign to cover specific attack steps. For example, currently, DMARACER executes generic device-to-kernel workloads to cover DMA race conditions. However, instead, by rewarding workloads that generate more taint—and thereby, more race conditions—we can cover more vulnerabilities.

Fault injection While DTA can effectively track attacker data, fault injection can effectively track *the effect* of attacker data. In particular, by corrupting attacker-tainted operations at runtime, we can improve coverage.

For example, currently, EINSTEIN builds syscall-centric data-only attacks by tracking the flow of attacker data into syscall arguments. However, many other types of data-only attacks are possible—e.g., those that corrupt conditional branches [94, 236]. Therefore, we can extend EINSTEIN to capture such attacks by diverting attacker-tainted branches at runtime, thereby improving its coverage.

Symbolic execution By employing symbolic execution, we can solve dataflow constraints at specific steps of an attack, thereby improving our accuracy. For example, we can integrate KASPER with recent work [229] to determine the exploitability of gadgets found. Moreover, we can integrate DMARACER with related work [11] to determine whether a driver indeed checks the data it loads from DMA. Each of these additions would eliminate a fundamental source of FPs.

5.2.2 Modeling Advanced Exploits

As novel vulnerabilities emerge, our generic approach—decomposing each exploit into its necessary steps and assigning targeted taint policies—can be extended to new domains.

Beyond Spectre-PHT KASPER currently focuses on Spectre-PHT, i.e., CPU speculative execution driven by the pattern history table (PHT). Yet processors also speculate via other hardware structures, such as the branch target buffer (BTB) [118] and return stack buffer (RSB) [121, 138], which can likewise give rise to transient execution vulnerabilities. We therefore propose extending KASPER to model these additional speculation sources, thus capturing a broader set of gadgets. One key challenge is that Spectre-BTB/RSB attacks exploit indirect branches, which can have many potential targets (unlike direct branches, which have only one alternative). A promising solution is to fully model BTB/RSB functionality by recording prior indirect branch targets and speculatively diverting to those at runtime.

Browser exploits EINSTEIN tracks attacker-corruptible data in server applications, effectively modeling exploits based on a generic write primitive. Similarly, modern web browsers process large amounts of untrusted input and often suffer from memory safety bugs [34, 91, 143]. Hence, extending EINSTEIN to track attacker data within a browser could reveal diverse vulnerabilities (e.g., in its JIT engine [51], autofill [52], and UI [53]). A main challenge here is accurately tracking data through the browser's complex JavaScript engine, which generates and executes code on the fly. Fortunately, prior dataflow tracking work on JavaScript engines [110, 227] can help overcome this hurdle.

Remote Rowhammer Rowhammer [114] is a hardware vulnerability that induces bit flips through rapid DRAM row accesses, and it can (in theory) be triggered remotely [127, 205]. Such attacks, however, are considered rare and more difficult because the attacker cannot run arbitrary code on the target system. Nevertheless, by extending EINSTEIN to track how attacker data influences memory access patterns, we can potentially identify remote Rowhammer-exploitable codepaths automatically. A key challenge, however, is forcing the victim to execute highly memory-intensive code. To address this, we could integrate directed fuzzing (as discussed in Section 5.2.1), rewarding input workloads that generate frequent, attacker-influenced memory operations—thus increasing the likelihood of triggering Rowhammer bit flips.

In summary, all these directions leverage the same fundamental insight: stepwise, explicit vulnerability modeling can handle the complexity of modern exploits. By integrating with alternate techniques and extending to next-generation vulnerabilities, our approach can generalize to mitigate future attacks.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *TISSEC*, 2009.

[2] Marshall Abrams and Joe Weiss. Malicious Control System Cyber Security Attack Case Study–Maroochy Water Services, Australia. *The MITRE Corporation*, 2008.

[3] Advanced Micro Devices, Inc. IOMMU Architectural Specification, 2007.

[4] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N Asokan, and Danfeng Daphne Yao. Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks. *USENIX Security*, 2023.

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. *S&P*, 2008.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. *USENIX Security*, 2009.

[7] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU. *EuroSys*, 2021.

[8] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. Solar Winds Hack: In-Depth Analysis and Countermeasures. *ICCCNT*, 2021.

[9] Starr Andersen and Vincent Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. `http://technet.microsoft.com/en-us/library/bb457155.aspx`, 2004.

[10] Anonymous. Once upon a free(). *Phrack*, 2001.

[11] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static Detection of Unsafe DMA Accesses in Device Drivers. *USENIX Security*, 2021.

[12] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. Hardware Assisted Randomization of Data. *RAID*, 2018.

[13] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. Deterministic Process Groups in dOS. *OSDI*, 2010.

[14] Sandeep Bhatkar and R Sekar. Data Space Randomization. *DIMVA*, 2008.

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

[16] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic Kernel TOCTTOU Protection. *USENIX Security*, 2022.

[17] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. *S&P*, 2014.

[18] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. *AsiaCCS*, 2011.

[19] Erik Bosman and Herbert Bos. Framing Signals——A Return to Portable Shellcode. *S&P*, 2014.

[20] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data Randomization. Technical report, Technical Report TR-2008-120, Microsoft Research., 2008.

[21] 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. *CCS*, 2019.

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

[23] Scott A Carr and Mathias Payer. DataShield: Configurable Data Confidentiality and Integrity. *AsiaCCS*, 2017.

[24] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. *OSDI*, 2006.

[25] James Cervini, Aviel Rubin, and Lanier Watkins. Don't Drink the Cyber: Extrapolating the Possibilities of Oldsmar's Water Treatment Cyberattack. *ICCWS*, 2022.

[26] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. *CCS*, 2010.

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

[28] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-Control-Data Attacks Are Realistic Threats. *USENIX Security*, 2005.

[29] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. *USENIX Security*, 2020.

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

[31] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. *TOPS*, 2021.

[32] Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation Techniques and Defenses for Data-Oriented Attacks. *SecDev*, 2019.

[33] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. *NDSS*, 2014.

[34] Chromium. Memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety/`.

[35] Clang 19.1.0 documentation - AddressSanitizer. `https://releases.llvm.org/19.1.0/tools/clang/docs/AddressSanitizer.html`.

[36] Clang 19.1.0 documentation - MemorySanitizer. `https://releases.llvm.org/19.1.0/tools/clang/docs/MemorySanitizer.html`.

[37] Clang 19.1.0 documentation - ThreadSanitizer. `https://releases.llvm.org/19.1.0/tools/clang/docs/ThreadSanitizer.html`.

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

[39] Jonathan Corbet. Supervisor mode access prevention. `https://lwn.net/Articles/517475/`, 2012.

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

[41] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 2005.

[42] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. *CCS*, 2017.

[43] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities. *USENIX Security*, 2003.

[44] CVE-2007-4727. https://nvd.nist.gov/vuln/detail/CVE-2007-4727.

[45] CVE-2013-2028. https://nvd.nist.gov/vuln/detail/CVE-2013-2028.

[46] CVE-2016-5195. https://nvd.nist.gov/vuln/detail/CVE-2016-5195.

[47] CVE-2021-32027. https://nvd.nist.gov/vuln/detail/CVE-2021-32027.

[48] CVE-2022-23943. https://nvd.nist.gov/vuln/detail/CVE-2022-23943.

[49] CVE-2022-41741. https://nvd.nist.gov/vuln/detail/CVE-2022-41741.

[50] CVE-2023-36824. https://nvd.nist.gov/vuln/detail/CVE-2023-36824.

[51] CVE-2024-8904. https://nvd.nist.gov/vuln/detail/CVE-2024-8904.

[52] CVE-2024-8908. https://nvd.nist.gov/vuln/detail/CVE-2024-8908.

[53] CVE-2024-8909. https://nvd.nist.gov/vuln/detail/CVE-2024-8909.

[54] Thurston HY Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. *AsiaCCS*, 2015.

[55] DataFlowSanitizer (Clang 12 documentation). https://releases.llvm.org/12.0.0/tools/clang/docs/DataFlowSanitizerDesign.html.

[56] DataFlowSanitizer (Clang 13 documentation). https://releases.llvm.org/13.0.0/tools/clang/docs/DataFlowSanitizerDesign.html.

[57] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the Server-Side Event-Driven Architecture. *EuroSys*, 2017.

[58] Dorothy E Denning. A Lattice Model of Secure Information Flow. *CACM*, 1976.

[59] Dorothy E Denning and Peter J Denning. Certification of Programs for Secure Information Flow. *CACM*, 1977.

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

[61] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. *USENIX Security*, 2017.

[62] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The Matter of Heartbleed. *IMC*, 2014.

[63] Victor Duta, Mitchel Josephus Aloserij, and Cristiano Giuffrida. SafeFetch: Practical Double-Fetch Protection with Kernel-Fetch Caching. *USENIX Security*, 2024.

[64] 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.

[65] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SOSP*, 2003.

[66] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. *OSDI*, 2010.

[67] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. *MICRO*, 2016.

[68] James P Farwell and Rafal Rohozinski. Stuxnet and the Future of Cyber War. *Survival*, 2011.

[69] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. *USENIX Security*, 2020.

[70] Jeffrey Stewart Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, 1973.

[71] Jeffrey Stewart Fenton. Memoryless Subsystems. *Comput. J.*, 1974.

[72] Cormac Flanagan and Stephen N Freund. FastTrack: Efficient and Precise Dynamic Race Detection. *PLDI*, 2009.

[73] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. *S&P*, 2018.

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

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

[76] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2C: Fine-grained Configuration-driven System Call Filtering. *CCS*, 2022.

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

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

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

[80] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating Memory Error Detection using the Floating Point Unit. *USENIX Security*, 2023.

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

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

[83] Qiang Guan, Xunchao Hu, Terence Grove, Bo Fang, Hailong Jiang, Heng Yin, and Nathan DeBadeleben. Chaser: An Enhanced Fault Injection Tool for Tracing Soft Errors in MPI Applications. *DSN*, 2020.

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

[85] Luanzheng Guo, Dong Li, Ignacio Laguna, and Martin Schulz. FlipTracker: Understanding Natural Error Resilience in HPC Applications. *SC*, 2018.

[86] William GJ Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. *FSE*, 2006.

[87] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for {Overflows}: A Guided Fuzzer to Find Buffer Boundary Violations. *USENIX Security*, 2013.

[88] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? *PASTE*, 2001.

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

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

[91] Diane Hosfelt. Implications of Rewriting a Browser Component in Rust. `https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/`, 2019.

[92] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. *USENIX Security*, 2015.

[93] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. *CCS*, 2018.

[94] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. *S&P*, 2016.

[95] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. *S&P*, 2013.

[96] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. *ISSRE*, 2019.

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

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

[99] 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.

[100] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. *CCS*, 2018.

[101] Jisoo Jang, Minsuk Kang, and Dokyung Song. ReUSB: Replay-Guided USB Driver Fuzzing. *USENIX Security*, 2023.

[102] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. Protect the System Call, Protect (most of) the World with BASTION. *ASPLOS*, 2023.

[103] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. *ATC*, 2002.

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

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

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

[107] Mateusz Jurczyk and Gynvael Coldwind. Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns, 2013.

[108] Michel Kaempf. Vudo malloc tricks. *Phrack*, 2001.

[109] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. *NDSS*, 2011.

[110] Rezwana Karim, Frank Tip, Alena Sochrková, and Koushik Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *TSE*, 2018.

[111] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. *VEE*, 2012.

[112] KernelDataFlowSanitizer. `https://github.com/vusec/kdfsan-linux/`.

[113] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. *S&P*, 2022.

[114] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *Comput. Archit. News*, 2014.

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

[116] Michael Kistler and Daniel Brokenshire. Detecting race conditions in asynchronous dma operations with full system simulation. *ISPASS*, 2011.

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

[118] 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. *S&P*, 2019.

[119] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization. *DSN*, 2016.

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

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

[122] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. *TrustCom*, 2015.

[123] Chris Lattner and Vikram S Adve. Transparent Pointer Compression for Linked Data Structures. *MSP*, 2005.

[124] Damien Le Moal. I/O Latency Optimization with Polling. *Vault*, 2017.

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

[126] 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. *USENIX Security*, 2018.

[127] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing Rowhammer Faults through Network Requests. *EuroS&PW*, 2020.

[128] Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu Zhang. TMDFI: Tagged Memory Assisted for Fine-grained Data-Flow Integrity towards Embedded Systems against Software Exploitation. *TrustCom*, 2018.

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

[130] 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. *USENIX Security*, 2021.

[131] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. Untrusted Hardware Causes Double-Fetch Problems in the I/O Memory. *JCST*, 2018.

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

[133] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. *CCS*, 2015.

[134] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *PLDI*, 2005.

[135] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. DFTinker: Detecting and Fixing Double-fetch Bugs in an Automated Way. *WASA*, 2018.

[136] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation. *ISSTA*, 2022.

[137] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. *USENIX Security*, 2017.

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

[139] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *TSE*, 2019.

[140] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. *NDSS*, 2019.

[141] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. *ATC*, 1996.

[142] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. *S&P*, 2021.

[143] Matt Miller. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. *BlueHat*, 2019.

[144] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks. *EuroS&P*, 2018.

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

[146] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler-Enforced Temporal Safety for C. *ISMM*, 2010.

[147] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *PLDI*, 2009.

[148] Robert HB Netzer and Barton P Miller. What Are Race Conditions? Some Issues and Formalizations. *LOPLAS*, 1992.

[149] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *NDSS*, 2005.

[150] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. *CCS*, 2015.

[151] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. *USENIX Security*, 2016.

[152] 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*, 2018.

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

[154] OpenBSD. pledge(2). `https://man.openbsd.org/pledge.2`, 2016.

[155] Elliott I Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972.

[156] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating Data Leakage by Protecting Memory-resident Sensitive Data. *ACSAC*, 2019.

[157] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. *S&P*, 2021.

[158] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. *USENIX Security*, 2013.

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

[160] PaX Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[161] Hui Peng and Mathias Payer. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. *USENIX Security*, 2020.

[162] Colin Percival. Cache missing for fun and profit. *BSDCan Ottawa*, 2005.

[163] Jannik Pewny, Philipp Koppe, and Thorsten Holz. Steroids for DOPed Applications: A Compiler for Automated Data-Oriented Programming. *EuroS&P*, 2019.

[164] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. *PLDI*, 2006.

[165] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for Android device drivers. *S&P*, 2020.

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

[167] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. *USENIX Security*, 2018.

[168] Razvan Raducu, Ricardo J Rodríguez, and Pedro Álvarez. Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review. *IEEE Access*, 2022.

[169] 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. *USENIX Security*, 2021.

[170] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. *AsiaCCS*, 2020.

[171] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. *NDSS*, 2017.

[172] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). *ACSAC*, 2009.

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

[174] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. *JCVHT*, 2018.

[175] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from Building Static Analysis Tools at Google. *CACM*, 2018.

[176] Selma Saidi and Ylies Falcone. Dynamic Detection and Mitigation of DMA Races in MPSoCs. *DSD*, 2015.

[177] Jerome H Saltzer. Protection and the Control of Information Sharing in Multics. *CACM*, 1974.

[178] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *TOCS*, 1997.

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

[180] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). *S&P*, 2010.

[181] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. *USENIX Security*, 2011.

[182] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. *CCS*, 2020.

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

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

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

[186] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. *CCS*, 2014.

[187] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. *ATC*, 2012.

[188] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic Race Detection with LLVM Compiler: Compile-time instrumentation for ThreadSanitizer. *RV*, 2011.

[189] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. *ATC*, 2005.

[190] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). *CCS*, 2007.

[191] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. *CCS*, 2004.

[192] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. *USENIX Security*, 2022.

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

[194] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. *NDSS*, 2016.

[195] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-flow Isolation. *S&P*, 2016.

[196] 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. *NDSS*, 2019.

[197] Eugene H Spafford. The Internet Worm Program: An Analysis. *SIGCOMM CCR*, 1989.

[198] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation. *IPAW*, 2015.

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

[200] N Stephens. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *NDSS*, 2016.

[201] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. *ASPLOS*, 2004.

[202] 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.

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

[204] 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. *USENIX Security*, 2018.

[205] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. *ATC*, 2018.

[206] The Linux Kernel documentation: Kernel Concurrency Sanitizer (KCSAN). https://docs.kernel.org/dev-tools/kcsan.html.

[207] The Linux Kernel documentation: Kernel Memory Sanitizer (KMSAN). https://docs.kernel.org/dev-tools/kmsan.html.

[208] The Linux Kernel documentation: MDS - Microarchitectural Data Sampling. https://docs.kernel.org/admin-guide/hw-vuln/mds.html.

[209] The Linux Kernel documentation: Spectre Side Channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

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

[211] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. *S&P*, 2020.

[212] Victor Van der Veen, Dennis Andriesse, Enes Gökta, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. *CCS*, 2015.

[213] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. *S&P*, 2016.

[214] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. *CCS*, 2017.

[215] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. *S&P*, 2019.

[216] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. Lightweight Memory Checkpointing. *DSN*, 2015.

[217] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and Efficient Application Monitoring and Replication. *ATC*, 2016.

[218] David Wagner and R Dean. Intrusion Detection via Static Analysis. *S&P*, 2000.

[219] 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.

[220] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. *TSE*, 2021.

[221] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. *USENIX Security*, 2017.

[222] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. DFTracker: Detecting Double-fetch Bugs by Multi-taint Parallel Tracking. *FCS*, 2019.

[223] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. *S&P*, 2010.

[224] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. *OSDI*, 2012.

[225] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. PatchRNN: A Deep Learning-Based System for Security Patch Identification. *MILCOM*, 2021.

[226] Robert NM Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *WOOT*, 2007.

[227] Shiyi Wei and Barbara G Ryder. Practical Blended Taint Analysis for JavaScript. *ISSTA*, 2013.

[228] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. *ESORICS*, 2016.

[229] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2. *USENIX Security*, 2024.

[230] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. DevFuzz: Automatic Device Model-Guided Device Driver Fuzzing. *S&P*, 2023.

[231] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. *S&P*, 2018.

[232] 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. *CCS*, 2015.

[233] Yiru Xu, Hao Sun, Jianzhong Liu, Yuheng Shen, and Yu Jiang. Saturn: Host-Gadget Synergistic USB Driver Fuzzing. *S&P*, 2024.

[234] Babak Yadegari and Saumya Debray. Bit-Level Taint Analysis. *SCAM*, 2014.

[235] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. *USENIX Security*, 2014.

[236] Hengkai Ye, Song Liu, Zhechang Zhang, and Hong Hu. Viper: Spotting Syscall-Guard Variables for Data-Only Attacks. *USENIX Security*, 2023.

[237] Suan Hsi Yong and Susan Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. *ESEC*, 2003.

[238] William D Young and John Mchugh. Coding for a believable specification to implementation mapping. *S&P*, 1987.

[239] 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. *MICRO*, 2019.

[240] Mingwei Zhang and R Sekar. Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks. *ACSAC*, 2015.

[241] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators. *NDSS*, 2022.

[242] Peter Zijlstra. Add static_call(). `https://lwn.net/Articles/824406`, 2020.

[243] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. *USENIX Security*, 2021.

[244] Yixin Zou, Abraham H Mhaidli, Austin McCall, and Florian Schaub. "I've Got Nothing to Lose": Consumers' Risk Perceptions and Protective Actions after the Equifax Data Breach. *SOUPS 2018*, 2018.

Summary

Modern computer exploits often involve complex dataflows. That is, they involve dangerous data (e.g., attacker data or secret data) traversing unconventional, multi-stage paths and subverting both software and hardware logic. Unfortunately, conventional methods based on Dynamic Taint Analysis (DTA) to identify such exploits—which use a monolithic "source-to-sink" policy to *implicitly* model simpler vulnerabilities—can miss such intricate dataflows.

To address this gap, this dissertation introduces *explicit vulnerability modeling*. By decomposing modern exploits into their essential sub-steps and assigning targeted taint policies for each, we can precisely capture the underlying dataflows. We validated this approach through three case studies: (i) finding *transient execution gadgets* by simulating branch mispredictions and the interplay between hardware and software bugs, (ii) constructing *data-only attack chains* by tracking attacker data into vulnerable syscall arguments and the resulting cross-syscall dependencies, and (iii) detecting *vulnerable DMA race conditions* by monitoring for hardware-level concurrency bugs and following attacker data to sensitive operations.

We evaluated each tool on large-scale, high-value codebases (e.g., the Linux kernel), discovering thousands of vulnerabilities and constructing hundreds of exploits. These results showcase the practicality of vulnerability-centric DTA in identifying and mitigating modern attacks, while also opening the door to new research directions.