WHEN MEMORY SERVES NOT SO WELL

MEMORY ERRORS 30 YEARS LATER

PH.D. THESIS

VICTOR VAN DER VEEN

VRIJE UNIVERSITEIT AMSTERDAM, 2019

Faculty of Science

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

Netherlands Organisation
for Scientific Research

This work is part of the research programme *Cyber Security* with project number 628.001.021, which is financed by the Netherlands Organisation for Scientific Research (NWO)

ISBN 978-94-6361-334-7

Cover design by Victor van der Veen
Printed by Optima Grafische Communicatie
This work was written in Vim, not Emacs

VRIJE UNIVERSITEIT

WHEN MEMORY SERVES NOT SO WELL

MEMORY ERRORS 30 YEARS LATER

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op donderdag 24 oktober 2019 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

VICTOR VAN DER VEEN

geboren te Hoorn

promotor: prof.dr.ir. H. J. Bos

copromotor: dr. C. Giuffrida

Voor Marieke

"First, it is slightly cheaper; and secondly it has the words

DON'T PANIC

inscribed in large friendly letters on its cover"

Douglas Adams on *The Hitchhiker's Guide to the Galaxy*

Acknowledgements

"Haha, het is echt het meest vage projectvoorstel dat ik ooit heb geschreven." This is how Herbert pitched his open PhD position to me, back in 2013. The proposal, indeed, was quite special: "Je zult ook af en toe een tijdje in Californië moeten zitten bij Christopher Kruegel in Santa Barbara" — what a horrible thought. It was only a few months earlier when I finally transitioned from eternal student being to a healthy, happy, functional, well-behaved grown-up. And now I considered going back to that asbestos clinic? What was wrong with me? Clearly, it required a lot of ~~dr~~thinking, discussing, soul searching, and iterating over pros and cons to come to the conclusion that I did not want to ask myself *what if?* 50 years from now. In other words, I did a PhD because I had *fear of missing out.*

And so I entered my four year rollercoaster ride as a PhD student at the Vrije Universiteit, in the systems and network security group — now known as VUSec. I consider myself extremely lucky to have been a member of this group and I am confident that this dissertation would not exist if it wasn't for all the bright people I had around me. First and foremost, I would like to thank my promotor, supervisor, and professor, Herbert Bos, for his guidance and support during my journey. He taught me not only how to write coherent stories, he also forced me to think critical and always look for scientific value when pursuing new ideas. His laid-back attitude, great humor settings and considerable basketball skills were a perfect match that made that I very much enjoyed the PhD experience. I honestly could not have wished for of a better supervisor than Herbert.

I am deeply grateful to my mentor and copromotor, Cristiano Giuffrida. Working with Cristiano was truly inspiring. Combined with his incredible amount of systems knowledge, his ability to come up with clever fixes and genius workarounds saved me numerous times. I will always remember that afternoon in Santa Barbara, where I had to "bring in the troops" — Cristiano came over (he was in the 'neighborhood' anyway) and managed to turn our theoretical attack into something that actually operated. Cristiano taught me to never settle, always be

skeptical, and, most importantly, how to perform sound scientific research. I am indebted.

Next, I would like to thank my reading committee, namely Michael Franz, Stefan Brunthaler, Mathias Payer, Lucas Davi, Stijn Volckaert, and Henri Bal. I cannot think of a better assembly of researchers with broad expertise on the topics covered in this dissertation to assert my claims. I am thankful that they all took time off their busy schedules to review this thesis.

I would like to express my gratitude to two extraordinary people in my life, Ben and Kaveh. I had the pleasure to first work with both, and later elevate our professional relationship into invaluable friendships. I learned a great deal about life, *and* research from both Kaveh and Ben. Thank you for the crazy memories. Our friendship is extremely important to me. You are both superstars.

I am also grateful for the many other people I had the pleasure to spend time with around the university. First, I would like to express my gratitude to Remco. Although our shared time in the group was short, Remco taught me the most important skills for completing a dissertation: prioritize and focus — *whatever you do, think about what and how it contributes to your goal of getting a PhD.* I took his advice to heart. Second, I wish to thank Chen. I still miss our breaks where we would come up with new ways of changing the world. I will never forget that hectic night before the TypeArmor submission where you showed me Chinese efficiency: our Skype call took less than one minute. Shit on the inkscape! Third, I thank Dennis, with whom I worked closely on my first paper. I envy his ability to finish such paper two weeks *before* the deadline, while simultaneously showing me that it is ok to write not-so-long-and-complex sentences — those without em-dashes. I would also like to thank Asia. She was always happy to burn some of her expensive cycles on explaining me the simplest basics of low-level systems. She taught me the fundamentals of writing and presenting — be nice to the color blind! Albeit short, it was a delight to work with her; if only it could have been a bit longer. Finally, I wish to express my thankfulness to Martina. Our collaborations where never boring, just as pretty much every conference we got to attend. Thank you for all the great nights and good talks.

I was fortunate enough to collaborate with many more smart researchers, both local and external to Amsterdam, resulting in a number of co-authored publications. My sincere thanks go to Elias, Alberto, Moritz, Yanick (the 100% premium-quality Italian), Enes, Daniel, Markus, Thorsten, Radhesh, Christopher, Federico, Clémentine, Georg, Matthias, Sebastian, Chris Ouwehand, Andre, Harikrishnan, Christian, Lionel, Martin, Manolis, Sanjay, Giovanni, Lukas, and Edgar. This thesis would not exist if it wasn't for you.

VUSec and the Computer Systems department of the Vrije Universiteit are a fantastic place to meet incredible individuals. I relished many CompSys borrel-takeover sessions, especially those for which we needed a 24-hour access card to get *out* of the building. Thank you Angelos, Alyssa (also for supporting me in San Diego!), Andrei Bacs, Andrei Tatar, Erik van der Kouwe, Istvan, Koen, Koustubha, Lucian, Natalie, Marco, Pietro (my code also has bugs, don't worry), Sanjay, Sebastiaan, Stephan, and Taddeüs. Thank you for making life bearable. Also thank you Pieter (alles goed?), Dirk, Ceriel, and Marc for the fun conversations we had. I also thank my roomies from P456. Erik Bosman, you are the smartest person I ever met and I doubt this will ever change; Arno, your absence was amazing!

Thank you Caroline, not only for shielding me from the university's bureaucracy, but also for being a beacon of light in troubling times. Because of you, I never had to worry about administrative tasks. On the contrary, I secretly hoped things would fail so that I had an excuse to catch up with Caroline. It was good to have a friendly ear to talk about sorrows other than computer systems. Also thank you Mojca, for arranging many things so swiftly. I also express my gratitude to my friends and former colleagues of the VU IT department. Support for Linux printing could be improved, but, if I recall correctly, my purple network connection never let me down.

California became our second home during my PhD. Our stay in Santa Barbara was immense. I thank the UCSB Seclab for hosting me. I am also grateful to Renwei Ge, for inviting me to join the Qualcomm Product Security Initiative in San Diego. I would also like to thank Alex Gantman, Can Acar, David Hartley, and Robert Turner. You made me feel at home, while giving me the opportunity to continue doing what I like most. Also thank you, Robert Buhren, for co-interning in a weird time frame.

Finally, I express my gratitude to my dear friend, Erik-Paul. It was nice to have a companion like you outside of academia, somebody who could keep me connected to the real world — our mid-week gatherings were a fitting way to vent about everything that is wrong with it. I admire your enthusiasm and your incredible pace of coming up with project ideas. I will nominate you as VUSec counselor as soon as they open the position. Also thank you Kasper, for the fun trips we had to Kuala Lumpur and your unhealthy amount of sarcasm. Let's try to keep Heap Heap Hooray alive.

Last but not least, I would like to thank those who are closest to my heart. Pap, mam, bedankt voor alles. Jullie hebben me altijd de vrijheid gegeven om mijn hart te volgen. Pap, jouw perfectionisme en mam, jouw nuchterheid hebben

gemaakt wie ik ben en ik denk dat dit zeker terug te vinden is in dit proefschrift. Casper, bedankt voor de morele support; ik kijk uit naar jouw afstudeerscriptie. Ik hou van jullie. Mannie en Nico, bedankt voor de ontelbare keren dat Lorèn (en Marieke) welkom waren ten tijde van deadlines en conferenties.

Marieke, woorden schieten tekort om te beschrijven hoe dankbaar ik je ben. Dankbaar voor de talloze fijne gesprekken over het leven; ik kan nog steeds veel van je leren. Uiteindelijk was jij het die me overhaalde om voor mezelf te kiezen en te gaan promoveren. Dankbaar voor het overnemen van de dagelijkse beslommeringen als ik weer eens 'druk' was en je er praktisch alleen voor stond. Met een baby — of later een peuter. Dankbaar ook, voor het meereizen naar Californië. Vooral die drie maanden in Santa Barbara, waarbinnen ik zo nodig nóg een artikel wilde schrijven, waren niet altijd even gezellig. Dankbaar dat je me meenam naar de buitenwereld, ook al zat ik soms vaker op mijn telefoon dan met mijn hoofd bij jullie. Ik hou zielsveel van je en ik verheug me op dat wat hierna gaat komen, wat het ook is. Lorèn, je hebt geen idee hoe belangrijk je bent geweest; je aanwezigheid dwong me om niet te blijven hangen in details, maar om projecten af te ronden zodat we snel weer naar Artis konden. Ik hou van jou ook.

Thank you all for the wonderful time I had in my late twenties and early thirties. It was epic.

Victor
Amsterdam, September 2019

Contents

Publications

Parts of this dissertation have been published earlier. The text in this thesis differs from the published versions in minor editorial changes that were made to improve readability. The following publications form the core of this thesis.

V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Słowińska, H. Bos, and C. Giuffrida. **Practical context-sensitive CFI**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.
[Appears in Chapter 2.]

V. van der Veen, E. Göktaş, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. **A tough call: Mitigating advanced code-reuse attacks at the binary level**. In *Proceedings of the* 37^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.
[Appears in Chapter 3.]

A. Pawlowski, V. van der Veen, D. Andriesse, E. van der Kouwe, T. Holz, and C. Giuffrida. **VPS: Excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching**. In *Proceedings of the* 35^{th} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2019.
[Appears in Chapter 4.]

V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. **The dynamics of innocent flesh on the bone: Code reuse ten years later**. In *Proceedings of the* 24^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2017.
[Appears in Chapter 5.]

V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. **Drammer: Deterministic rowhammer attacks on mobile platforms**. In *Proceedings of the* 23^{rd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2016.
[Appears in Chapter 6.]

V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi. **GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM**. In *Proceedings of the* 15^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jun. 2018.
[Appears in Chapter 7.]

The following publications are not included in this dissertation.

S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Schmiedecker, and E. Weippl. **Enter sandbox: Android sandbox comparison**. In *Proceedings of the* 3^{rd} *IEEE Mobile Security Technologies Workshop (MoST)*. May 2014.

M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. **ANDRUBIS - 1,000,000 apps later: A view on current Android malware behaviors**. In *Proceedings of the* 3^{rd} *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. Sep. 2014.

R. K. Konoth, V. van der Veen, and H. Bos. **How anywhere computing just killed your phone-based two-factor authentication**. In *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016.

A. Coletta, V. van der Veen, and F. Maggi. **DroydSeuss: A mobile banking trojan tracker - short paper**. In *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016.

D. Andriesse, X. Chen, V. van der Veen, A. Słowińska, and H. Bos. **An in-depth analysis of disassembly on full-scale x86/x64 binaries**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. **MARX: Uncovering class hierarchies in C++ programs**. In *Proceedings of the* 24^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2017.

1 Introduction

It is 1988. The Internet is at its inception. Sir Tim Berners-Lee needs at least one more year before he will propose the *World Wide Web* — Google does not exist for another decade. Users are mostly well-behaved intellects, primarily academics, who work for universities or research organizations. Among the most popular Internet services is *Usenet*, a worldwide distributed discussion system. Instead of *tweeting*, a verb that is still reserved exclusively for birds, users share information by reading and posting messages to Usenet categories, known as newsgroups. It is November 3 when Peter Yee, a researcher at NASA Ames, posts the following to the `comp.protocols.tcp-ip` group.

> **Internet VIRUS alert.**
>
> We are currently under attack from an Internet VIRUS. It has hit UC Berkeley, UC San Diego, Lawrence Livermore, Stanford, and NASA [...]. This program copies [...] binaries that try to replicate the virus via connections to TELNETD, FTPD, FINGERD, RSHD, and SMTP. [...] For now turning off the above services seems to be the only help.

The warning came too late. In only a couple of hours, thousands of systems got infected and reached a complete halt. Estimates suggest that the virus spread across ten percent of all 60,000 computers attached to the Internet, putting the cost of the damage at the range of $100,000 to $10,000,000. As regional networks disconnected from the backbone while cleaning their machines, normal activity was severely disrupted and connectivity was impeded for days. The Internet just lost its innocence.

The cause of this problem was Robert Morris, at the time a 22-year-old graduate student at Cornell University. Morris, not one of the well-behaved intellects *yet*, started his attempt to "gauge the size of the Internet" on Wednesday evening, November 2, 1988. By exploiting a number of known vulnerabilities in core system services, his code could spread from machine to machine without any user

interaction. While this approach should be considered unorthodox at the very least, it was ultimately a programming error that turned his attempt at running a relatively harmless measurement study into an attack with devestating concequences.

The virus that would later became known as the *Morris Worm*, was an iconic event in Internet history. Morris himself can claim the dishonorable award of being the first person to be tried and convicted under the 1986 Computer Fraud and Abuse Act. More revolutionary though, is that his actions obliterated the complacency of computer security being mostly a theoretical problem. The Morris Worm forced software vendors to anticipate more proactively on reported security flaws in their products. It galvanized the field of security research, creating a demand in both industry and academia. As we will see shortly, computer attacks have been on a resurgence ever since, and many of us computer security 'experts' fighting these, can trace their roots back to the events of November 1988.

Showing his intellectual capabilities, Morris later became a successful entrepreneur, being involved in the selling of Viaweb to Yahoo for $49 million in 1998. He currently holds a position at MIT and is a well-established colleague in the security community.

Memory Errors

One of the vulnerabilities that Morris exploited was a *memory error*, a (software) bug that corrupts the program's internal memory state. At that time, the code for fingerd — a remote user information server — accepted an arbitrary large input from untrusted sources. By providing more bytes than fingerd expected, the Morris Worm triggers a *buffer overflow*. This allows the worm to overwrite memory that it should not have access to, including a *return address*: a data value that the CPU uses as a bookmark to jump back to. By corrupting the return address so that it would point to his own input, Morris was able to *divert control flow* of the program. He dictates the CPU to start executing instructions that *he* provided, instead of those of the actual program. This allowed him to take full control over the program and thus the remote server.

Memory errors are among the oldest classes of software vulnerabilities. To date, the research community has proposed and developed a number of different approaches to eradicate or mitigate memory errors and their exploitation. From safe languages, which remove the vulnerability entirely [67, 166], and bounds checkers, which check for out-of-bounds accesses [6, 69, 117, 149], to countermeasures that prevent certain memory locations to be overwritten [31, 36], de-

tect code injections at early stages [108], or prevent attackers from finding [13, 209], using [159, 72], or executing [205, 234] injected code.

Despite over three decades of independent, academic, and industrial research efforts, memory errors still undermine the security of our systems. This is true even if we consider only classic buffer overflows: this class of memory errors has been lodged in the top-3 of the top 25 most dangerous software errors for years [206, 215]. Experience shows that attackers, motivated nowadays by profit rather than fun [180], have been effective at finding ways to circumvent protective measures [45, 167]. Many attacks today start with a memory corruption that provides an initial foothold for further infection.

To make matters even worse, memory errors are no longer limited to the software domain. In 2014, *Rowhammer* was introduced. This disturbance error is the result of the ever increasing density of memory chips, a necessity to be able to put more and faster DRAM memory in new devices. However, assembling memory cells — tiny capacitors — closer to each other, makes them prone to leaking charge into adjacent cells on memory accesses. By repeatedly accessing, i.e., "hammering", the same cell over and over again, a neighboring cell may lose its charge faster than it should, causing a bit to flip. The Rowhammer bug allows an attacker to flip a bit in memory without requiring access to that memory location [76]. Ever since its discovery, researchers used Rowhammer-based memory corruptions to exploit a variety of ecosystems, including the desktop [216], browser [19, 59], and even the cloud [114].

Figure 1.1 puts this emerging trend into perspective. By searching for certain keywords in the summary description of each Common Vulnerability and Exposure (CVE) entry in the National Vulnerability Database[1], we can distinguish memory errors from other vulnerabilities. In Figure 1.1, we use this to display the *absolute* number of memory errors that are reported every six months, over the course of the last two decades. The plot depicts a substantial influx of memory errors, steadily increasing in numbers over time.

Looking at Figure 1.1, it is tempting to try and identify different eras in memory error history: starting with *the outset*, we see linear growth in the number of CVEs from 1998 to 2008. We then entered *the redemption*, a period of 7 years (2008–2015) where the expansion came to a halt and the trend maybe even went down a bit. A cause may be the enormous increase of web vulnerabilities, as this was the time that the *Web 2.0* really took of [133]. Since 2015, we experience *the resurgence*. We notice an exponential growth of memory errors that is still continuing today: in the first half of 2018 (not plotted in the figure), a massive

[1] https://nvd.nist.gov/

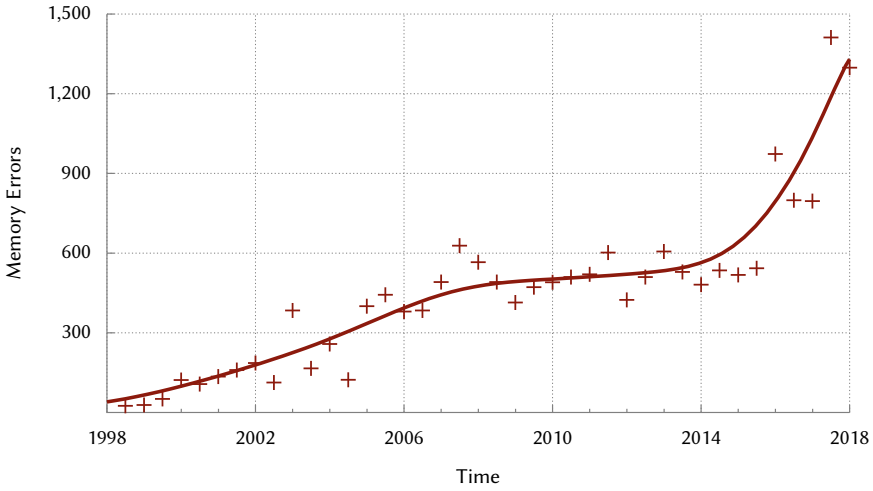


Figure 1.1. Absolute number of memory error vulnerabilities (CVEs), reported per 6 months, over the course of two decades. This plot shows that, after a decade of linear growth, followed by a brief stagnation, we currently experience an exponential increase in memory errors.

number of 1,441 memory errors have been reported. Without proper evaluation of these records, we can only speculate on what causes this. One possibility may be that security research shifted from finding web vulnerabilities to testing Internet of Things (IoT) devices; these appliances are often programmed in low-level programming languages like C and C++, making them more vulnerable to buffer overflows than SQL-injection attacks.

Figure 1.2 shows the relative number of reported memory error vulnerabilities. It also includes the portion of memory error *exploits* — scraped from the exploitdb database.[2] The trend that we witness here is more stable than those for absolute numbers discussed before: for two decades, roughly 20% of all vulnerabilities and exploits are related to memory errors.

Figures 1.1 and 1.2 show that 30 years after Morris released one of the first memory error attacks, these vulnerabilities are still relevant today and that there is no sign that this will change anytime soon. Combined with our knowledge about the Morris Worm — memory errors can have a devastating impact — it justifies academic research on these type of vulnerabilities. If launched today, a Morris Worm 2.0, i.e., an outbreak that dismantles 10% of the Internet, will have consequences that are orders of magnitude worse than those in November 1988.

[2] https://www.exploit-db.com/

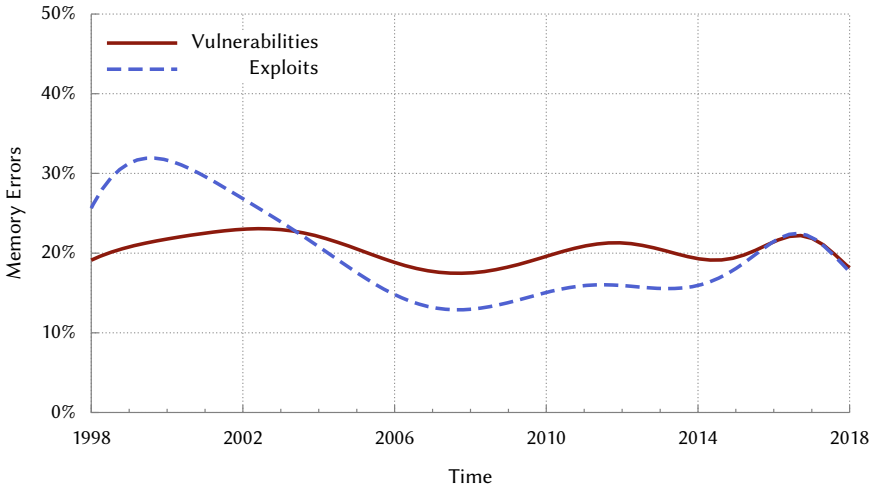


Figure 1.2. Relative number of memory error vulnerabilities (in red) and exploits (in dashed blue), reported per six months, over the course of two decades. This plot shows that memory errors make up around 20% of all vulnerabilities and exploits, ever since we started to keep track of them.

Goals

The goal of this work is to study and advance computer security defenses that primarily focus on preventing memory-error-based exploitation. We approach this from both the software and hardware level.

Software errors. The first part of this thesis dissects and improves the state of the art in defenses against one of the most advanced types of memory-error exploits: *code-reuse attacks*. Code-reuse attacks are the mature version of Morris' fingerd exploit. They are the result of a cat-and-mouse game between attackers and defenders that took place over the past two decades. They are responsible for some of the most impactful outbreaks we witnessed in recent history [172, 133].

Hardware errors. In the second part of this thesis, we study a relatively young hardware-based memory error: *the Rowhammer bug*. We explore how Rowhammer-based attacks and defenses behave on mobile platforms. We are the first to unravel that bit flips on Android devices are possible and that attackers can use these to mount powerful memory-corruption exploits on mobile platforms.

Organization

This dissertation makes several contributions, most with published results in refereed conferences (Page xix). The following provides a brief summary of these and acts an an global outline for the remainder of this thesis. Note that parts of this introduction, specifically the experiments related to Figures 1.1 and 1.2, as well as some of the text on memory errors, are based on prior work that was already published before the start of this dissertation [133].

We study memory errors across two dimensions. First, Chapters 2 to 5 focus on a particular topic in the area of **software-based memory errors**: code-reuse attacks. In contrast, Chapters 6 and 7 concentrate on a specific instance of **hardware-based memory errors** on mobile platforms: the Rowhammer bug.

Chapter 2 presents PATHARMOR, a context-sensitive Control-Flow Integrity (CFI) implementation, aimed at protecting binaries without requiring access to their source code. The key idea is to rely on hardware extensions for recording a trace of recently executed branches. Then, just before the execution of a sensitive endpoint, the recorded "path" leading towards this endpoint is matched against the program's internal control-flow graph, ensuring that no control-flow diversion can take place. PATHARMOR deploys significantly stronger invariants than context-insensitive CFI, without incurring a major performance impact.
Chapter 2 appeared in the *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS 2015)* [131]. PATHARMOR was nominated for the *Dutch Cyber Security best Research Paper (DCSRP) Award* at ICT.OPEN in 2016 [193].

Chapter 3 presents TYPEARMOR, a binary-level CFI implementation that reconstructs a conservative approximation of function and callsite prototypes to reduce the number of allowed targets on the forward edge. For possible targets, TYPEARMOR relies on use-def analysis to compute the *minimum* number of parameters that this function consumes. Similarly, liveness analysis at indirect callsites yields the *maximum* number of arguments that they prepare. The many-to-many relationship between callsites and target callees that we then derive — *a callsite that prepares at most* n *arguments may never invoke a function that consumes at least* $n + 1$ *arguments* — achieves a much higher precision than prior solutions.
Chapter 3 appeared in the *Proceedings of the* 37^{th} *IEEE Symposium on Security and Privacy (S&P 2016)* [135].

Chapter 4 presents VPS, a binary-level defense against vtable hijacking in C++ applications. VPS achieves accurate protection by restricting virtual callsites to

validly created objects. More specifically, VPS ensures that virtual callsites can only use objects that are created at valid object construction sites. VPS prevents wrongly identified virtual callsites from breaking the binary, an issue most previous work do not consider.
Chapter 4 appeared in the *Proceedings of the* 35^{th} *Annual Computer Security Applications Conference (ACSAC 2019).*

Chapter 5 presents NEWTON, a runtime gadget-discovery framework based on constraint-driven dynamic taint analysis. The key insight in NEWTON is that we can model the capabilities of a powerful attacker — one with arbitrary memory read/write primitives — by means of dynamic taint analysis. In particular, we taint all bytes that an attacker can modify with a unique color and then track the flow of taint until we reach code that, given the right values for the tainted bytes, allows the attacker to launch a code-reuse attack. NEWTON can model a broad range of code-reuse defenses by mapping their properties into simple, stackable, reusable constraints, and automatically generate gadgets that comply with these. Using NEWTON, we systematically map and compare state-of-the-art defenses, demonstrating that even simple interactions with popular server programs are adequate for finding gadgets for all state-of-the-art code-reuse defenses.
Chapter 5 appeared in the *Proceedings of the* 24^{th} *ACM Conference on Computer and Communications Security (CCS 2017)* [132]. In 2017, NEWTON won the *Best Presentation Award* (presented by co-author Manolis Stamatogiannakis) at the *Cyber Security Workshop* in the Netherlands. In 2018, NEWTON was nominated for the *Dutch Cyber Security best Research Paper (DCSRP) Award* at ICT.OPEN [195].

Chapter 6 presents DRAMMER, a deterministic Rowhammer attack on mobile platforms. DRAMMER shows that the Rowhammer bug — a DRAM disturbance error that allows an attacker to flip a single bit in memory not under the attacker's control — is prevalent on platforms other than x86. By carefully massaging physical memory, we exploit a single bit flip to mount a deterministic privilege escalation exploit without having to rely on esoteric operating system features, like memory deduplication, as used in prior work.
Chapter 6 appeared in the *Proceedings of the* 23^{rd} *ACM Conference on Computer and Communications Security (CCS 2016)* [134]. As the attack was carried out on an Android device, Google assigned CVE-2016-6728 to refer to DRAMMER. In 2017, DRAMMER gathered international media coverage and won multiple awards: the *Applied Research Best Paper Award* at CSAW, region North America [208], the *Pwnie for Best Privilege Escalation Bug* at Blackhat US [211], and the *Dutch Cyber Security best Research Paper (DCSRP) Award* at ICT.OPEN [194]. DRAMMER was

also nominated for a *Pwnie* in the category *Most Innovative Research* at Blackhat US [210].

Chapter 7 presents RAMPAGE and GUARDION. RAMPAGE elaborates on DRAMMER and shows that the deployed mitigations in Android are not sufficient in stopping mobile Rowhammer attacks. It consists of a set of DMA-based offenses against the latest Android OS, including (1) a renewed root exploit, and (2) a series of app-to-app exploit scenarios. GUARDION is a lightweight defense that prevents DMA-based attacks by isolating physical memory of DMA buffers with guard rows.
Chapter 7 appeared in the *Proceedings of the* 15^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2018)* [136]. Google assigned CVE-2018-9442 to refer to RAMPAGE. In 2018, RAMPAGE and GUARDION gained international media attention and won the *Best Research Award* at the International Conference on Computing Systems (CompSys 2018). RAMPAGE was nominated for a *Pwnie* in the category *Best Privilege Escalation Bug* at Blackhat US 2018 [212].

Chapter 8 concludes this dissertation, summarizes results, analyzes limitations, and discusses future research directions.

2 Practical Context-Sensitive CFI

Current Control-Flow Integrity (CFI) implementations track control edges individually, insensitive to the context of preceding edges. Recent work demonstrates that this leaves sufficient leeway for powerful ROP attacks. Context-sensitive CFI, which can provide enhanced security, is widely considered impractical for real-world adoption. This chapter shows that Context-sensitive CFI (CCFI) for both the backward and forward edge can be implemented efficiently on commodity hardware. We present PATHARMOR, a binary-level CCFI implementation which tracks paths to sensitive program states, and defines the set of valid control edges *within the state context* to yield higher precision than existing CFI implementations. Even with simple context-sensitive policies, PATHARMOR yields significantly stronger CFI invariants than context-insensitive CFI, with similar performance.

2.1 Introduction

Control-Flow Integrity (CFI) [2] has developed into one of the most promising techniques to stop code-reuse attacks against C and C++ programs. Typically, such attacks circumvent common defenses such as DEP/W⊕X by diverting a program's control flow to a set of Return-Oriented Programming (ROP) gadgets [27, 124]. Likewise, they defeat widely deployed ASLR by either targeting gadgets at fixed (non-randomized) addresses [18], or by dynamically disclosing the addresses of randomized gadgets [126]. CFI promises to prevent all such attacks by ensuring that all control transfers conform to the program's original Control Flow Graph (CFG). In theory, CFI is very powerful and, in its purest and ideal form, provably secure against most integrity violations of the control flow [1].

Ten years after the original CFI proposal [2], however, researchers are still working to find practical CFI implementations [30, 181, 83, 102, 130, 154, 157], able to approximate the security of the purest form of CFI with acceptable performance. Common CFI solutions, including state-of-the-art binary-level implementations such as bin-CFI [157] and CCFIR [154], attempt to substantially relax constraints on the set of legal targets for both the backward (e.g., `ret` instructions) and forward (e.g., indirect `call` instructions) control edges. While doing so reduces the performance overhead to a few percent only, it also provides more degrees of freedom for the attackers. Other even more lightweight CFI solutions, such as ROPecker [30] and kBouncer [102], build on heuristics and hardware support to detect anomalous control flows—which resemble ROP gadget chains—and stop many current exploitation attempts at low performance overheads. Unfortunately, a string of recent publications comprehensively shows that it is possible to circumvent all these lightweight CFI solutions with relatively little effort [26, 43, 57, 58, 121].

A key problem with traditional CFI solutions — even recent source-level fine-grained ones [130] — is that they enforce only context-insensitive CFI policies, which examine control edges in isolation and attempt to statically derive the resulting superset of all the possible targets according to the CFG. The lack of context inevitably results in weak CFI invariants, allowing attackers to freely chain edges together and form paths that are even trivially infeasible in the original CFG (e.g., returning to a function never on the active call stack [57]).

Context-sensitive CFI techniques are a promising way to address this problem, since they rely on context-sensitive static analysis to associate CFI invariants to control-flow *paths*—i.e., multiple consecutive edges—in the CFG and enforce such invariants on execution paths at runtime. The stronger security guar-

antees provided by context-sensitive CFI techniques have been acknowledged as early as in the original CFI proposal, but their real-world adoption has been rapidly dismissed as impractical [2].

In this chapter, we demonstrate that Context-sensitive CFI (CCFI) can indeed be implemented in an efficient, reliable, and practical way for real-world applications. We present PATHARMOR[1], the first binary-level CCFI solution which enforces context-sensitive CFI policies on both the backward and forward edges. PATHARMOR relies on hardware support to efficiently and reliably monitor execution paths to sensitive functions which can be used to mount control-flow diversion attacks [102], and uses a carefully optimized binary instrumentation design to enforce CCFI invariants on the monitored paths. PATHARMOR's path invariants are derived by a scalable context-sensitive static analysis performed over the CFG on-demand, which uses caching of path verification steps to achieve high efficiency. Verification itself is also very efficient, since all the CFI checks are batched at sensitive program points.

To show the practicality of our design, we have prototyped two context-sensitive and binary-level CFI policies (for the backward and forward edges, respectively) on top of PATHARMOR. Moreover, our framework can also serve as a general foundation for even stronger CCFI implementations, for instance using context-sensitive data-flow analysis at the source level. Even in the current setup, PATHARMOR provides a comprehensive CCFI protection system with much stronger security guarantees than traditional CFI solutions, while matching or even improving their performance. Moreover, due to its optimized design, PATHARMOR can also serve as an efficient basis for fine-grained context-insensitive CFI ($\overline{\text{C}}$CFI) policies.

Contributions Our contribution is threefold:

- We identify the key challenges towards practical CCFI implementations and investigate opportunities to address these challenges in real-world applications and commodity platforms.
- We present PATHARMOR, a framework to support context-sensitive and context-insensitive CFI policies on commodity platforms. To fulfill its goals, PATHARMOR relies on hardware support, binary instrumentation, and on-demand static analysis to batch even sophisticated CFI checks at the relevant sensitive points in a binary. We complement our solution with fine-grained $\overline{\text{C}}$CFI policies and simple but comprehensive (backward and for-

[1] PATHARMOR is open source, available via https://github.com/vusec/patharmor

ward edge) CCFI policies, making PATHARMOR the first practical end-to-end CCFI implementation.

- We evaluate PATHARMOR using popular server applications and the SPEC CPU2006 benchmarks. Our results show that PATHARMOR can significantly restrict the number of legal control flows compared to traditional CFI solutions (-70% across all our applications, geometric mean), while yielding bounded memory usage (+18-74 MB on our applications) and low runtime performance overhead (3% on SPEC and 8.4% on our applications, geometric mean).

2.2 Context-Sensitive CFI

The general goal of every CFI solution is to allow all the control flows which occur in the interprocedural control-flow graph (CFG) defined by the programmer, and reject the largest possible fraction of the other flows as illegal [2]. This section formalizes the definition of a legal flow adopted in existing practical CFI solutions, contrasts it with the stricter definition adopted in Context-sensitive CFI (CCFI), and details the key challenges towards practical CCFI.

2.2.1 Legal Flows

We model a CFG as a digraph $G = (V, E)$ where V is the set of basic blocks, and E the set of control edges in the CFG defined by the program.

Traditional CFI [2] enforces that each individual (indirect) control transfer taken by the program during the execution must match an edge in the CFG:

Context-insensitive CFI ($\overline{\text{C}}$CFI). *For each control transfer $e_i = (v_x, v_y)$ between basic blocks v_x and v_y, $\overline{\text{C}}$CFI enforces that $e_i \in E$.*

In other words, $\overline{\text{C}}$CFI checks conformance to the current position in the CFG and does not distinguish between different paths in the CFG that lead to a given control transfer. For instance, consider the two code paths that both lead to the execution of a function Z in Figure 2.1. Disregarding the context would allow function Z to return to either B or D. However, we should only allow a return (backward edge) to B, when coming from A (and B). Likewise, we should only allow a return to D if the program got there via C.

We can easily construct a similar example for the CFG's forward edges, for instance by considering callbacks. Suppose B and D both call Z with callback argument cb_B and cb_D, respectively. When Z invokes the callback, $\overline{\text{C}}$CFI would

```
A() {
  indirect call to B();
}

B() {
  indirect call to Z();
}
```

```
C() {
  indirect call to D();
}

D() {
  indirect call to Z();
}
```

Figure 2.1. Two code paths that both lead to the execution of function *Z*.

allow either of the (cb_B and cb_D) targets, while taking the context into consideration would allow us to (rightly) conclude that cb_B is only legal if we reached Z via B.

To mimic context-sensitive behavior to some degree (backward edges only) a number of existing CFI solutions complement their operations with a shadow stack [13, 29, 31, 35, 47, 111, 115, 184, 150]. However, shadow stacks are typically expensive at the binary level [29, 41, 184]. Moreover, unlike CFI techniques, their security relies on the integrity of in-process runtime information, which state-of-the-art implementations typically protect using system-enforced ASLR—with its known security limitations and probabilistic guarantees against arbitrary memory write vulnerabilities. All the other existing CFI solutions, in turn, implement fully context-insensitive ($\overline{\text{C}}$CFI) policies as described above.

In addition, state-of-the-art binary-level CFI solutions, such as CCFIR [154] or binCFI [157], further relax their $\overline{\text{C}}$CFI policies for performance reasons. These context-insensitive implementations group control transfer sources and destinations based on a general definition of type, and enforce that the source and destination type match:

Practical $\overline{\text{C}}$CFI. *For each control transfer $e_i = (v_x, v_y)$ between basic blocks v_x and v_y, practical $\overline{\text{C}}$CFI ensures $x \in sources(type(e_i)) \wedge y \in sinks(type(e_i))$, where $sources(\tau)$ and $sinks(\tau)$ denote the sets of program locations having outbound or inbound edges of type τ, respectively.*

Practical $\overline{\text{C}}$CFI precludes malicious control transfers like jumps into the middle of a function, or returns to non-call sites. Attackers, however, can still successfully mount powerful attacks using gadgets which adhere to the imposed type restrictions [25, 26, 43, 57, 121].

CCFI provides stronger CFI invariants than both practical and ideal $\overline{\text{C}}$CFI. Rather than considering control transfers individually, CCFI examines each transfer in the context of recently executed transfers:

CCFI. *Given a path $p = (e_1, e_2, \ldots, e_n)$ of control transfers leading to a given program point P, CCFI verifies the validity of P by checking that $\forall i \in \{1, 2, \ldots, n\}$, edge e_i is consecutively valid in the context of all preceding CFG edges $e_1, \ldots, e_{i-1}$.*

Since CFI checks are enforced *per path* (rather than *per edge*), CCFI can enable arbitrarily powerful context-sensitive policies on both the backward and forward edges.

2.2.2 Challenges

In this section, we discuss the three fundamental challenges towards practical CCFI, and in the remainder of the chapter, we present PATHARMOR—the first practical binary-level solution to these problems—proving CCFI effective in practice.

C1. Efficient path monitoring A major challenge in implementing a practical CCFI solution is identifying an efficient mechanism to constantly monitor paths of executed control flow transfers at runtime. Other than imposing minimal performance overhead, the path monitoring mechanism should also be reliable, that is neither the program nor the attacker should be able to tamper with the recorded data. All these requirements were considered the key obstacle to the real-world adoption of context-sensitive CFI in the original CFI proposal [2].

To address this challenge, PATHARMOR relies on branch recording features available in modern x86_64 processors to implement an efficient and reliable path monitoring mechanism at runtime.

C2. Efficient path analysis To verify the validity of a path towards a given program point P, CCFI needs to statically analyze the CFG and identify the legal paths towards P in a context-sensitive fashion, validating all the edges in the path. The naive solution—statically enumerating *all* the legal paths towards *all* the relevant program points—cannot scale efficiently to large and complex CFGs, with the number of paths growing exponentially with $|V|$ and $|E|$. This *path explosion* problem is well known in several application domains (symbolic execution, among others [82]). Even focusing our static analysis on a particular program point and sequence of indirect control transfers derived by runtime information only partially eliminates this problem. Path explosion can still occur between any two indirect control edges, especially in presence of loops and long sequences of direct jumps and calls.

To address this challenge, PATHARMOR relies on an on-demand, constraint-driven context-sensitive static analysis over a normalized CFG representation. The constraints, derived by runtime information recorded by our path monitor,

allow our context-sensitive path analysis to efficiently scale to arbitrarily large and complex CFGs.

C3. Efficient path verification To detect control-flow diversion attacks, CCFI needs to carefully select program points to verify the current execution path for validity. To provide strong security guarantees, path verification needs to be performed in all the possible execution states that are potentially harmful. The naive solution—performing path verification after every executed control transfer—is clearly inefficient and scales poorly with the path length.

To address this challenge, PATHARMOR relies on a kernel module to efficiently verify only the paths to well-defined sensitive functions in the program. While the verification still needs to run for each path to these functions encountered during the execution, PATHARMOR aggressively caches verification results to minimize the resulting impact on runtime performance. Since the number of paths to sensitive functions is limited in practice (as shown in Section 2.5.2 for popular server programs), caching is effective in amortizing path verification costs throughout the execution.

2.3 PathArmor

Figure 2.3 presents the high-level overview of PATHARMOR and details its three main components: (1) a kernel module, (2) an on-demand static analyzer, and (3) an instrumentation component.

PATHARMOR relies on a *kernel module* which provides a Branch Record core to support per-thread control transfer monitoring in multi-process and multi-threaded programs. For this purpose, our module uses the 16 *Last Branch Record* (*LBR*) registers available in modern Intel processors and only accessible from ring 0. This strategy allows our module to monitor paths of (16) recently exercised control transfers in an efficient and reliable way (addressing **C1**).

In addition to path monitoring, our kernel module triggers path verification steps upon security-sensitive system calls issued by the program—but also other special sensitive operations, as detailed later. To further improve the performance of path verification, our module also maintains a path cache, which stores hashes of previously verified paths and eliminates the need to enforce more expensive CCFI checks at each verification (addressing **C3**). We discuss our kernel module in more detail in Section 2.3.1.

Once the kernel module is loaded, protected program binaries run with the *dynamic instrumentation component*. This component first starts our *path analyzer*, an external trusted component which runs in the background and waits

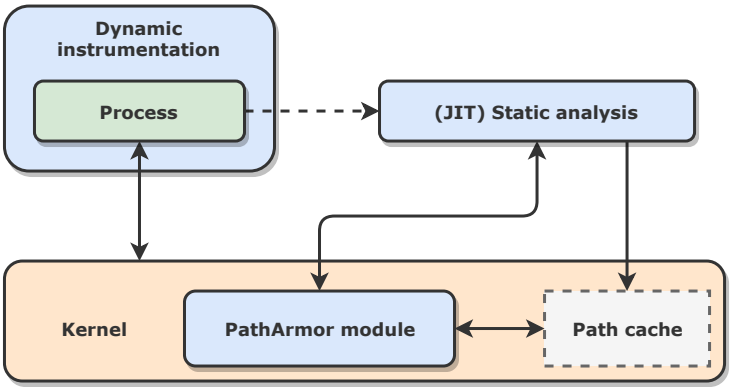


Figure 2.3. High-level overview of PATHARMOR. Before a protected process can execute a sensitive endpoint, our PATHARMOR module instructs a just-in-time analysis to validate whether the most recently executed branches form a legal path in the process' control-flow graph. Valid paths are hashed and stored in a small cache for fast lookups. Dynamic instrumentation provides the process with an in-program communication channel with our module.

for path verification requests from the kernel module via a dedicated upcall interface. To satisfy path verification requests, our analyzer receives all the necessary LBR-based path information—and constraints on indirect but also interprocedural direct control transfers—from our kernel module and performs static analysis on-demand to enforce our CCFI policies. For this purpose, the analyzer needs to reconstruct the CFG of the target binary and preprocess it with a preliminary *CFG reduction* step that prunes all the irrelevant intraprocedural edges from the control-flow graph. This step and our constraint-based strategy eliminate all the intraprocedural and interprocedural (respectively) path explosion threats, ensuring a scalable on-demand path analysis (addressing **C2**). After determining whether a path is valid, our analyzer reports its findings back to the kernel module, which, in response, stops the program (if verification fails) or populates the path cache (otherwise). We elaborate more on our path analyzer in Section 2.3.2.

After initializing PATHARMOR's path analyzer, our *dynamic instrumentation component* sets up an in-program communication channel with the kernel module to enable (and later manage) path monitoring for the target binary. Finally, our instrumentation component instruments the binary according to a predetermined sensitive path termination policy. PATHARMOR can, in principle, be configured to verify either entire paths to sensitive system calls or limit such paths to the library call interface. The current PATHARMOR implementation uses the latter mode of operation by default, given that, on commodity hardware, the

LBR can only record the 16 most recently executed control transfers and allowing branch tracing inside the libraries can potentially "pollute" paths and thus "erase" program context—an observation also made in prior work [102]. The trade off—which can, however, be reconsidered with future hardware extensions—is that PATHARMOR's default configuration can defend against control-flow diversion attacks only in the program, excluding those originating from vulnerabilities in the libraries from the threat model. For completeness, we evaluate the feasibility of future in-library path tracking in Section 2.5.4. We discuss our instrumentation component in more detail in Section 2.3.3.

2.3.1 Kernel Module

As illustrated in Figure 2.3, the kernel module consists of two main components: (1) a system call interceptor that sends validation requests (via a cache) to the on-demand static analyzer, and (2) a Branch Record core (LBR API) that monitors and records branches occurring within the target's main address space.

System call interception As mentioned in Section 2.2.2, PATHARMOR limits verification to a small number of security sensitive path endpoints in order to maintain minimal runtime overhead. In particular, these endpoints consist of a set of dangerous system calls an attacker requires to deploy a meaningful exploit, like `exec` and `mprotect` (and other dedicated sensitive operations, see Section 2.3.3). We refer to them as sensitive calls. Like other work in this area [30], we propose to detect only these dangerous endpoints, rather than every possible library and system call.

To intercept system calls at runtime, the kernel module installs an alternative syscall handler. When our target requests execution of a dangerous system call, we pause execution, collect LBR data, and forward it to the on-demand static analyzer in user space. If the analyzer returns `true` (meaning that the path was found in the CFG and thus is valid), the kernel module stores a hash of the path in a cache data structure before permitting the system call. We use cryptographically secure second-preimage resistant hash algorithms (MD4 in our evaluation) to prevent path crafting attacks, where attackers craft an invalid path with a hash that collides with that of a valid path: For a second-preimage resistant hash algorithm h and input x, it is computationally hard to find a second input $x' \neq x$ such that $h(x) = h(x')$.

If the exact same path is executed a second time, PATHARMOR looks for its hash in the cache, and only sends a request to the on-demand static analyzer if no match was found in the cache. This limits the amount of overhead caused by

traversing the CFG.

In the event that on-demand static analysis returns a negative result (no valid path was found in the CFG), the module stops the program and reports that an attack was detected. With the LBR data still in place, this can also help pinpoint the exact location of the attack.

Branch recording In addition to path verification, the kernel module provides a Branch Recording core that implements support for tracking branches on a per process-thread basis. In addition, it exposes an interface to the instrumented libraries that is used to disable branch recording during library execution. It can do this either using the LBR (the current default) or Intel's Branch Trace Storage (BTS) feature. Although prior work has shown that BTS imposes a significant performance slowdown (typically in the order of 20-40x [127]), its 'unlimited' nature provides a useful means to measure how many LBR registers are required to approach optimal security (Section 2.5).

Ideally, we would configure the Branch Recording core to collect only indirect branches (indirect jumps, indirect calls and returns), as only these branches can be modified by an attacker. However, armed only with information about indirect branches exercised by the program, we cannot eliminate the path explosion problem. To solve this issue, we instruct the Branch Recording core to keep track of direct call instructions as well, which can be used by the on-demand static analysis to eliminate path explosion, rendering PATHARMOR efficient in practice. We elaborate more on this in Section 2.3.2.

To disable branch recording during library execution, we expose two `ioctl()` requests to libraries: `LIB_ENTER` and `LIB_EXIT`. The dynamic instrumentation component detailed in Section 2.3.3 inserts these requests for each used library function by instrumenting their entry and exit points. We discuss related implementation challenges such as how to enable branch recording again for callbacks, in depth in Section 2.4. Note that attackers cannot abuse `ioctl` requests to disable PATHARMOR, as discussed in Section 2.6.3.

2.3.2 Path Analyzer

The role of the path analyzer is to verify (on the static CFG) at runtime if a particular path observed at an endpoint is valid. It consults the CFG of the binary and searches it for the path. We now discuss the three main steps of this analysis: CFG generation, a reduction of the CFG to eliminate the path explosion problem, and path validation.

CFG generation To validate a path, PATHARMOR requires an accurate CFG of the protected binary. To obtain a CFG, we use existing binary analysis frameworks to disassemble and analyze binaries, as detailed in Section 2.4. Previous work has shown that the results are accurate enough in practice [162, 156]. To err on the safe side, PATHARMOR tolerates potential errors by overestimating the CFG when necessary. In the worst case, this may cause PATHARMOR to accept invalid paths, but it will never reject legitimate ones.

In addition, PATHARMOR implements indirect edge resolution policies to augment a CFG walk with indirect edges in a context-sensitive manner. If these policies fail, we resort to a fine-grained context-insensitive policy instead [154, 157].

For **backward edges** (i.e., returns), our policies implement a fully context-sensitive resolution strategy, to which we refer as *call/return matching*. This strategy emulates a runtime call stack by tracking call and return edges as these are encountered.

For **forward edges** (i.e., indirect calls), our current prototype supports a simple context-sensitive strategy which resolves code pointers propagated across caller-callee pairs with no contrived data flow. This policy lets us unambiguously resolve indirect call sites, at which call targets are loaded as constants and passed as a callback argument. However, our path abstraction, in principle, enables much more complex context-sensitive extensions. We evaluate the additional security provided by forward-edge context-sensitivity in Section 2.5. In cases where our current policy is unable to trace a code pointer (e.g., in case of a long-lived code pointer stored on the heap), PATHARMOR resorts to a $\overline{\text{C}}$CFI policy which matches all indirect call sites with all the functions having their address taken. Indirect jumps, in turn, are conservatively resolved by the underlying binary analysis framework.

We also implement a strategy to augment the precision of indirect jumps found in PLT entries. The CFG is updated with data received from the instrumentation component, enabling unambiguous target resolution. We discuss this resolution in more detail in Section 2.3.3.

Addressing path explosion As discussed in Section 2.2, static analysis of large CFGs may lead to a path explosion problem, where the number of paths to explore increases exponentially with the exploration depth. PATHARMOR takes two measures to address the problem and perform efficient path verification.

First, as a preprocessing round, PATHARMOR performs a *CFG reduction* step that significantly prunes the CFG, and preserves reachability relations with re-

spect to indirect edges and interprocedural direct edges. This step finds all possible paths of direct edges between entry and exit points of each function, and then collapses these paths down to a single edge between each entry point and the exit points reachable from it. This makes the subsequent search much faster, as needless (re-)explorations of direct edges can be avoided (e.g., loops).

Second, call/return matching (discussed in Section 2.3.2) allows us to recognize and discard impossible paths, such as paths that call a function from one call site, and subsequently return to another call site. Without call/return matching, the path search would have no way of identifying such mismatches.

Path verification The path analyzer is given a path that must be verified. The path is an LBR state containing direct and indirect calls, indirect jumps, and returns. To verify whether it is valid, the analyzer performs a Depth-First Search (DFS) on the CFG to find a path that contains the provided edges in the same order as they were recorded by the LBR. A recorded path is thus considered valid iff: (1) all edges in the LBR state exist within the CFG, and (2) these edges can be linked together via a valid path of direct edges within the CFG. To ensure that the search terminates quickly if a path does *not* exist (e.g., the LBR state is malicious), the DFS does not follow indirect edges or direct call edges. Following such edges would not make sense, because by definition, such edges would be in the LBR state if they occurred on the path under analysis.

Note that in the presence of (1) direct call recording and (2) the CFG reduction, the DFS cannot get stuck on cycles within the CFG. Indeed, it first consults the LBR for the oldest recorded branch, from a basic block `A` to a basic block `B`, and then loops over all possible outgoing edges of `B` to see which one to follow. Due to the CFG reduction, direct jump edges are collapsed, so the outgoing edges of `B` are all either indirect edges or direct call edges. For each edge the DFS examines, it checks whether this edge is the next recorded branch. If this does not hold, it tries the next edge, until it finds one that matches the following LBR state. From here, it restarts analysis, starting from this new edge. This process continues until the last edge (the most recently recorded branch) is found.

2.3.3 Dynamic Instrumentation

The instrumentation component consists of a library instrumentation (in the form of a special loader), and dynamic binary instrumentation module. Its main objectives are (1) collecting address offsets (for libraries and the target program) and passing these to the static analysis component, (2) instrumenting libraries so that they disable LBR tracking before their execution starts and re-enable it again

once finished, and (3) starting the actual target process. In addition, the instrumentation component opens a communication channel with the kernel module that the inserted instrumentation snippets use to communicate with the Branch Recording core. We now discuss our instrumentation modules in more detail.

Loader The loader is responsible for setting up the PATHARMOR environment before starting the protected binary. It is implemented as a pre-loaded shared library using `LD_PRELOAD` and instruments the target binary's `main()` function. This hook is then used to open an `ioctl()` interface with the LBR API of the kernel module, which in turn is used by the inserted code snippets to notify the kernel module of specific events (e.g., `LIB_ENTER`).

In addition, the loader collects the program's PLT and GOT entries as well as the base addresses of the different libraries that are in place. This information is then passed via the kernel module to the on-demand static analyzer where it is used to distinguish calls to library functions from branches within the program's main address space. For this to work, the target program is started with `LD_BIND_NOW=1`, which causes the dynamic linker to resolve all symbols at the program startup instead of deferring function call resolution to the point when they are first referenced.

Rewriter In its default configuration, PATHARMOR terminates all sensitive paths at library boundaries. For this purpose, our dynamic instrumentation module uses Dyninst to rewrite all library functions that are used by the program (i.e., those that can be found in the process' PLT table, as well as those dynamically loaded using `dlsym()`). The inserted code snippets ensure that library functions first send an disable request to the LBR API in the kernel module before executing, and finish with an LBR enable request before returning to the program.

Disabling the LBR of course comes at a price: a library function may at some point invoke a callback handler which may or may not reside in the target's address space. If we do not re-enable the LBR again on callbacks, a bug in the callback handler could still be exploited by an attacker as we lose vital information on executed paths. To overcome this problem, we apply another round of dynamic instrumentation, only this time to make sure that whenever such a callback is invoked, LBR tracking is enabled again. We discuss this process in more detail in Section 2.3.3.

The dynamic instrumentation module of the initialization component performs necessary rewriting tasks at load time (when dynamically linked libraries are available) and at runtime (every time a new shared library is dynamically

loaded into memory). Note that we only need to instrument shared libraries. No instrumentation is required in the protected applications, leaving the original target's code space intact.

Callbacks As mentioned above, a second dynamic instrumentation round is required in order to enable branch recording again when a library function invokes a callback that lies in the program's binary (e.g., `qsort()`). Instrumenting callback sites is done by looping over all shared library functions and searching for indirect call instructions. For each indirect call instruction, a short snippet of code is inserted that (1) tests if the target of the indirect call lies in the target program's address space, and (2) if this is the case, wraps the call instruction in two `ioctl()` system calls that notify the kernel module that a callback function is entered or exited: (`CALLBACK_ENTER` and `CALLBACK_EXIT`, respectively).

Whenever the kernel module receives the `CALLBACK_ENTER` request, it pushes the current LBR state (i.e., the content of the LBR registers as seen before the library function that performs the callback) to an internal stack of LBR contexts. When the callback exits (`CALLBACK_EXIT`), the kernel module pops the top of this LBR stack back into the actual registers. To support code that forks within a callback, the kernel module copies the stack of LBR contexts to the newly created process, so that parent and child both safely return to their callback handlers without inconsistent branch records.

Observe that signals are essentially a specialized form of callbacks and can be processed in a similar manner. The only difference is that instead of instrumenting code, we install a hook on the kernel's signal delivery function. This hook is executed before control is returned to the signal handler, allowing us to save the current LBR context so that it can be restored upon the `sigreturn` system call.

This approach of switching LBR contexts at the moment callback handlers are invoked raises a specific security threat where an attacker could install a different handler than normally enforced by the CFG. Consider the example where an attacker exploits a memory corruption bug to install a callback handler that fits his needs. Without applying any additional security measurements, this operation may go unnoticed (control-flow diversion happens indirectly in the kernel or in the libraries). To overcome this situation, PATHARMOR (i) considers signal handler registration and LBR management operations (i.e., push context, pop context) as sensitive operations and (ii) always copies the last branch entry during LBR context switching as the first branch entry for the new context, allowing on-demand static analysis to apply our indirect edge resolution policies on the library-originated indirect call edge before allowing the callback. A symmetric

approach is used to avoid false positives for library-originated function pointers (e.g., returned by `dlsym()`) which are used for indirect call invocations by the program. Our static analyzer resolves the "special" library target in a dedicated way without resorting to more sophisticated modular CFI policies [97].

Special constructs Similarly to our callback support, PATHARMOR supports the `longjmp()` construct by implementing a special handler for this in the kernel module: for each `setjmp()`, the kernel stores the existing LBR contents along with the provided `env` argument. When a `longjmp()` is executed, our module verifies the LBR contents, flushes them and restores the LBR with the appropriate state as stored earlier (matching `env`). Similarly to callbacks, we rely on our dynamic instrumentation component to insert dedicated `SETJMP` and `LONGJMP` `ioctl()` requests for each construct.

2.4 Implementation

We implemented PATHARMOR on Linux v3.13 for x86_64 with support for multi-process and multi-threaded applications. Our kernel module is implemented as a standard loadable module for the Linux kernel in 1,752 LOC. The on-demand static analysis component is implemented as a plugin for the Dyninst binary analysis and rewriting framework [11] in 6,741 LOC overall. The library instrumentation is implemented as another Dyninst plugin in 1,625 LOC.

To intercept sensitive system calls, we install an alternative syscall handler by overwriting the `MSR_LSTAR` register. PATHARMOR will forward most system calls directly to their vanilla implementation, imposing little to zero extra overhead. However, we consider a total of seven system call families as dangerous, and start verification whenever these are encountered: `mprotect` and the `mmap` family (which can be used to disable DEP/W⊕X), and the `exec` family (which can be used to start a malicious command) are obvious choices and have been considered in prior work in the area [30]. To address the challenges related to signal handling as detailed in Section 2.3.3, PATHARMOR also intercepts the `sigaction` and `sigreturn` system calls. PATHARMOR can also be configured to protect I/O system calls, to prevent attacks like data leaks and script injection in (for instance) web servers.

Since Linux does not currently support per-task LBR context management, we implemented it to avoid pollution from other processes. We used the standard preemption notifier functionality (`preempt_notifier_register`) provided by the Linux kernel to install hooks on context-switches. During a context-switch-

out (sched_out), PATHARMOR stores the LBR state of the current process into an LBR process table, to restore it later when the thread is scheduled in again (sched_in). This approach allows PATHARMOR to support binaries that make use of multi-threading.

Our current PATHARMOR prototype is based on the Dyninst binary rewriting framework, and as a consequence does not support C++ exceptions. This limitation is not fundamental to PATHARMOR, and can be addressed in future work with additional engineering effort.

2.5 Evaluation

We evaluated PATHARMOR on a workstation equipped with an Intel i5-2400 CPU 3.10 GHz and 8 GB of RAM. We ran all our tests on an Ubuntu 14.04 installation running Linux kernel 3.13 (x86_64). To measure the performance impact of PATHARMOR for the worst case, we default PATHARMOR to run in non-library operation mode, but we evaluate the effects of enabling in-library tracking in Section 2.5.4.

We focus our evaluation on popular Linux server applications, given that (i) they are widely known and adopted in the research community for evaluation purposes, (ii) they are popular exploitation targets for both local and remote attacks, and (iii) they naturally contain a relevant number of security-sensitive functions that can benefit from the protection guarantees provided by PATHARMOR. Specifically, we evaluated our prototype with three popular FTP servers (namely, vsftpd v1.1.0, ProFTPD v1.3.3, and Pure-FTPd v1.0.36), two popular web servers (nginx v0.8.54 and lighttpd v1.4.28), a popular SSH server (the OpenSSH Daemon v3.5), and a popular email server (Exim v4.69). We also evaluated the performance of PATHARMOR on SPEC CPU2006.

To benchmark our web servers, we used the Apache benchmark [239] configured to issue 25,000 requests with 10 concurrent connections and 10 requests per connection. To benchmark our FTP servers, we relied on the pyftpbench benchmark [242] configured to open 100 connections and request 100 1 KB-sized files per connection. To benchmark OpenSSH and Exim, finally, we used the OpenSSH test suite [243] and a homegrown script which repeatedly launches the sendemail program [245], respectively. We configured all our applications and benchmarks with their default settings. We ran all our experiments 11 times (checking that the CPUs were fully loaded throughout our tests) and report the median with marginal variations observed across runs.

Our evaluation answers 4 key questions: (1) *Security*: Is PATHARMOR effective

in improving the security of existing CFI techniques against control-flow diversion attacks? (2) *Analysis time*: Does PATHARMOR's static analysis complete in reasonable time? (3) *runtime performance*: Does PATHARMOR yield low runtime overhead while protecting a relevant set of sensitive functions? (4) *Memory usage*: How much memory does PATHARMOR require?

2.5.1 Security

To evaluate the security guarantees offered by PATHARMOR and, in particular, the improvements offered by CCFI over existing $\overline{\text{C}}$CFI techniques, we measured the strength of the CFI invariants extracted by our static analysis and enforced by PATHARMOR's runtime verification. For this purpose, we instructed our static analyzer to generate CFI statistics during the execution of our benchmarks and compare the results against fully context-insensitive CFI policies. Note that these statistics (and metrics) are intended only to provide a clear picture of the strength of PATHARMOR's invariants compared to other CFI solutions. As such, the following discussion focuses on a relative comparison across CFI implementations, rather than on absolute numbers.

Table 2.1a presents control-flow statistics aggregated across our applications. The first, second, and third group of columns provide an overview of all the applications analyzed, their sensitive functions, and their interprocedural CFG (or simply CFG) information generated by our analyzer with fully context-insensitive indirect edge resolution policies. As the table shows, the number of sensitive functions as well as the number of nodes and edges in the CFG ($|V|$ and $|E|$, respectively) varies greatly across applications, reflecting their different internal structure.

The fourth group of columns, in turn, reports the fraction of indirect backward edges (*IB*), indirect forward edges (*IF*), and direct forward edges (*DF*) in the LBR averaged across all the sensitive function calls during the execution of our benchmarks. As expected, the overall distribution is relatively stable across applications, with backward edges largely dominating (indirect) forward edges (37% vs. 25% geometric mean). Encouragingly, direct forward edges—which, however necessary to scalably enforce our CCFI policies, also naturally decrease the number of LBR entries subject to CFI enforcement—have a significant but non-dominant impact in practice (37% geometric mean).

Table 2.1b presents averaged gadget statistics for coarse-grained $\overline{\text{C}}$CFI, fine-grained $\overline{\text{C}}$CFI and CCFI policies (respectively). In detail, the $|G|$ column reports the average number of targets (and thus gadgets) allowed by the given CFI policy for each indirect edge observed in the LBR. The $\min[G_{\text{Len}}]$ column, in turn,

Table 2.1. Runtime CFI statistics for the evaluated server programs. Table 2.1a lists control-flow properties for each evaluated server program. Table 2.1b compares permitted control flows in coarse-grained, fine-grained, and context-sensitive CFI.

(a) Control-flow properties. The **Functions** column reports the sensitive endpoints observed at runtime: sa=`sigaction`, sg=`signal`, ra=`raise`, ki=`kill`, mm=`mmap`, m4=`mmap64`, mp=`mprotect`, el=`execl`, ev=`execv`, and ee=`execve`. The **CFG** group reports the number of nodes and edges in the CFG. The **LBR** group reports the average number of indirect backward edges, indirect forward edges, and direct forward edges in the LBR, just before executing a sensitive endpoint.

		CFG		**LBR (Avg.)**		
Server	**Functions**	$\lvert V\rvert$	$\lvert E\rvert$	$\frac{\lvert E_{\text{IB}}\rvert}{\lvert E\rvert}$	$\frac{\lvert E_{\text{IF}}\rvert}{\lvert E\rvert}$	$\frac{\lvert E_{\text{DF}}\rvert}{\lvert E\rvert}$
exim	sa,sg,ki,ev,ee	37,906	167,867	0.34	0.28	0.38
lighttpd	sa,sg,ki,m4,el	7,380	38,006	0.38	0.22	0.40
nginx	sa,ra,ki,m4,ee	26,029	432,829	0.45	0.20	0.35
openssh	sa,sg,mm,el,ev,ee	14,749	63,644	0.38	0.26	0.36
proftpd	sa,sg,ki,mm	29,682	210,489	0.38	0.27	0.35
pure-ftpd	sa,	5,702	19,910	0.32	0.33	0.35
vsftpd	sa,mm,mp	4,052	9,269	0.33	0.23	0.44

(b) Number of gadgets and minimum gadget length for coarse-grained $\overline{\text{C}}$CFI, fine-grained $\overline{\text{C}}$CFI, and CCFI, averaged for each indirect edge observed in the LBR, just before executing a sensitive endpoint.

	$\overline{\textbf{C}}\textbf{CFI}_{\textbf{cg}}$ **(Avg.)**		$\overline{\textbf{C}}\textbf{CFI}_{\textbf{fg}}$ **(Avg.)**		**CCFI (Avg.)**	
Server	$\lvert G\rvert$	$\min[G_{\text{Len}}]$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$
exim	2,589	2.2	25	4.4	11	11.1
lighttpd	561	2.0	3	4.8	1	5.5
nginx	1,482	2.8	23	9.3	15	9.9
openssh	1,725	2.1	16	3.9	4	7.2
proftpd	3,250	2.2	20	4.0	6	7.5
pure-ftpd	404	2.2	5	4.5	2	5.1
vsftpd	543	3.5	3	8.0	1	13.1

provides more qualitative information on the resulting CFI-allowed gadgets, by averaging the minimum allowed gadget length for each edge observed in the LBR. As shown in the table, CCFI yields a significantly lower average number of gadgets compared to coarse-grained and fine-grained $\overline{\text{C}}$CFI (respectively, -99.7% and -61.6% geometric mean). Figure 2.4 also details the CDF of the number of allowed targets for the two applications with most sensitive calls (Exim and ProFTPD). We observed similar trends for the other applications. The CDF confirms that CCFI allows very few targets for the vast majority of control flow trans-

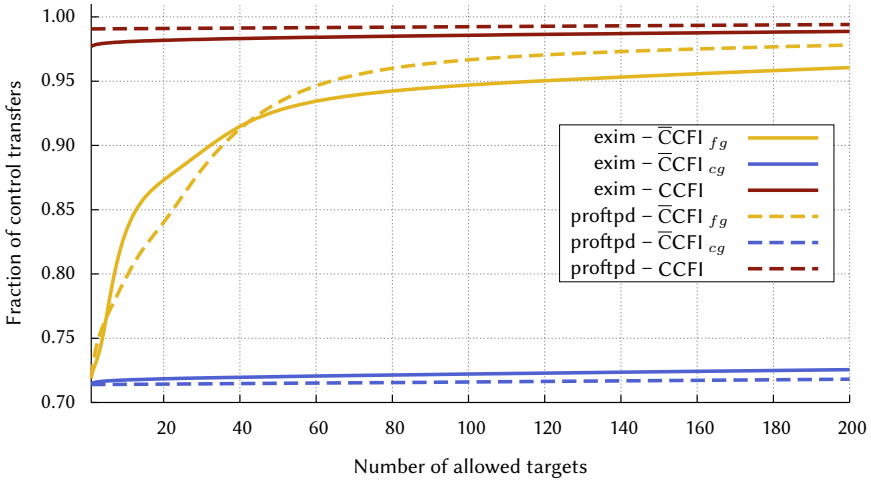


Figure 2.4. Distribution of the number of allowed targets for indirect branches in the LBR when applying coarse-grained $\overline{\text{C}}$CFI, fine-grained $\overline{\text{C}}$CFI, and CCFI on Exim and ProFTPD—two applications with a high number of sensitive endpoints. The plot shows what fraction of control transfers may target how many gadgets. For example, when applying CCFI on ProFTPD, 98% of the control transfers may target less than 40 gadgets, while coarse-grained $\overline{\text{C}}$CFI only enforces that 72% of the indirect branches have less than 40 possible targets.

fers — for instance, on Exim, 98% have less than 13 targets compared to around 86% for fine-grained $\overline{\text{C}}$CFI and 72% for coarse-grained $\overline{\text{C}}$CFI (the common policy for binary-level CFI solutions [154, 157]). This demonstrates the effectiveness of our context-sensitive CFI policies, which can drastically restrict the number of legal targets for most LBR entries.

Our improvements are naturally also reflected in the overall complexity of the gadgets left to the attacker, with the average minimum allowed gadget length ($\min[G_{\text{Len}}]$) substantially increasing compared to the coarse-grained and fine-grained versions of $\overline{\text{C}}$CFI (respectively, $+245\%$ and $+53\%$ geometric mean). In general, shorter gadgets are easier to fit together and are more preferred than longer gadgets for building a ROP chain. By reducing the possible indirect edge targets, the attacker's gadget arsenal gets diminished and the bar for exploitation increased. As an example, Table 2.1 shows that the reduction in the average number of indirect edge targets from 17 to 2.3 for Exim resulted in an increase of the average number of instructions in the shortest allowed gadgets from 4.4 to 11. With CCFI, a deeper gadget analysis also revealed a significant increase in the average number of register accesses in the shortest allowed gadgets com-

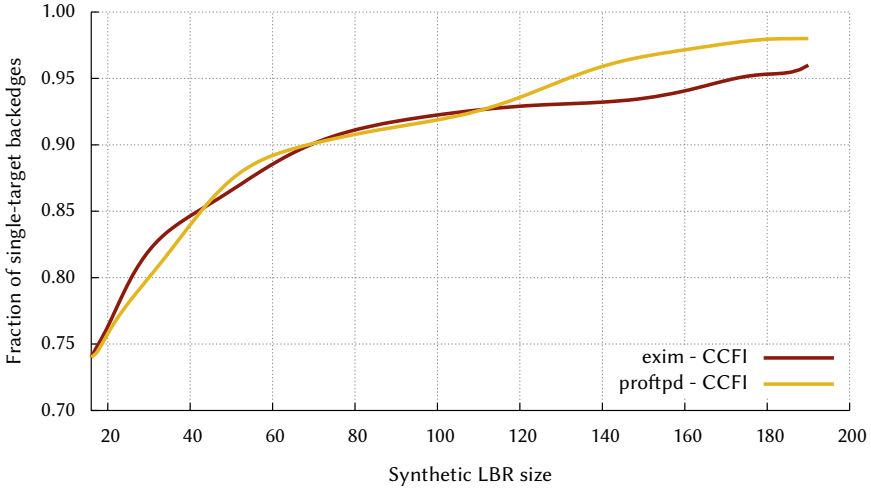


Figure 2.5. Fraction of single-target backedges for CCFI for Exim and ProFTPD—two applications with a high number of sensitive endpoints—when simulating an increasingly large LBR. The plot shows that roughly 90% of the return instructions would have a single valid destination if the LBR can hold 64 branches.

pared to the coarse-grained and fine-grained versions of $\overline{\text{C}}$CFI. The geometric means of these accesses for the coarse-grained $\overline{\text{C}}$CFI, the fine-grained $\overline{\text{C}}$CFI and CCFI are respectively 1.3, 4.5 and 7.7. This further confirms the increased gadget complexity when using CCFI policies.

To evaluate the effectiveness of the particular CCFI techniques implemented in PATHARMOR, we also examined the impact of context sensitivity on both edges in more detail. For this purpose, we first compared our (static) backward-edge CCFI policy with that enforced by a (dynamic) shadow stack, the only known (runtime) solution which mimics context-sensitive control-flow policies—albeit only on the backward edge and using tamper-prone and more heavyweight instrumentation at the binary level. For a fair comparison, we focused our measurements on the fraction of backward edges observed in the LBR which are allowed only one target (in a fully context-sensitive fashion) by our CCFI techniques and also relied on Intel's BTS feature to simulate an LBR of arbitrary size—overcoming the restrictions imposed by commodity hardware.

Figure 2.5 presents our results for increasing LBR sizes and the two applications with most sensitive calls (Exim and ProFTPD). We observed similar trends for the other applications. On commodity hardware (16 LBR entries), PATHARMOR can enforce a single target for nearly 75% of the backward edges observed in the LBR. In the remaining cases, the limited LBR size causes PATHARMOR to lose

Table 2.2. Indirect call reduction and JIT analysis statistics. The **Target reduction** group shows the fraction of legal indirect targets for (ideal binary-level) context-sensitive vs. context-insensitive forward-edge CFI. The **JIT** group shows how much time was spent running static analysis at runtime, while **Cache** presents how many path cache hits and misses occurred during the execution of our benchmarks.

	Target reduction		**JIT (ms)**		**Cache**	
Server	*Indirect calls*	$\frac{targets_{cs}}{targets_{ci}}$	*Total*	*Average*	*# Misses*	*# Hits*
exim	99	0.89	100	3	40	1,871
lighttpd	66	0.84	28	2	13	2
nginx	271	0.82	24	5	5	10
openssh	131	0.82	52	2	22	49
proftpd	120	0.99	140	4	39	2,495
pure-ftpd	11	1.00	56	2	27	1,915
vsftpd	6	0.38	24	3	9	2,283

program context and resort to $\overline{\text{C}}$CFI policies. While the current LBR size limit prevents PATHARMOR from fully reaching the ideal shadow stack performance (100%), these results are still encouraging given the small default LBR size. In addition, Figure 2.5 shows that future hardware extensions can help fill the gap, e.g., enforcing a single target in 90% of cases with 70 LBR entries.

To evaluate the effectiveness of our forward-edge CCFI policy, we examined the reduction in the number of allowed indirect call targets caused by context sensitivity. Due to the very limited number of indirect call entries in the LBR for our test programs (which rarely use indirect calls close to sensitive function points), however, we did not observe any significant reduction in our experiments. To generalize our results and eliminate any application-specific bias, we applied our policy to all the code paths. This still resulted in a relatively small reduction overall (less than 5% in most cases). This is, however, expected, given that our current binary-level forward-edge CCFI policy is very simple—only propagating function pointers passed in call arguments in a straightforward way—and only intended to demonstrate the practicality of implementing arbitrary forward-edge CCFI policies in PATHARMOR. To examine the potential for more sophisticated forward-edge CCFI policies, we approximated an *ideal* binary-level context-sensitive forward-edge analysis using higher-level language semantics—i.e, implemented on top of LLVM 2.9 Data Structure Analysis (DSA) [84].

The **Target reduction** group in Table 2.2 shows the effect of the resulting forward-edge CCFI policy on our set of server programs. In most cases, our context-sensitivity policy causes a reduction of around 10% to 20% for the average number of indirect call targets. The reduction varies depending on the

context-sensitive function pointer resolution accuracy. For vsftpd, we obtain a reduction of 62%, while numbers decrease for applications with more complex pointer resolutions. We believe these results are encouraging, stimulating research on more sophisticated forward-edge CCFI policies—which PATHARMOR can serve as a basis for. Moreover, DSA's flow-insensitive and unification-based design aggressively merges data-flow information, improving speed but also resulting in overly conservative results [84]. In addition, due to implementation limitations, DSA is known to produce even more conservative, and thus pessimistic, results on modern LLVM releases [230]. Thus, an updated version of DSA (or a more precise, but also less scalable analysis) would already likely yield substantially improved forward-edge results.

Overall, our analysis shows that CCFI is effective in generating robust CFI invariants to defend against even sophisticated control-flow diversion attacks. While attacks are still theoretically possible—and they might be even for an ideal CCFI solution—the adoption of context sensitivity sensibly limits the quantity and quality of gadgets available to the attacker. This is in stark contrast, for example, with unrestrictedly allowing simple *call-site gadgets*, which have been used to mount attacks against prior $\overline{\text{C}}$CFI techniques [57].

2.5.2 Analysis Time

PATHARMOR's on-demand path analysis translates to increased application runtime. To evaluate the impact, we measured the time spent in our analyzer—using our CCFI policies—during the execution of our benchmarks. The right-hand side of Table 2.2 presents our results: it details the total and average analysis time measured across all the paths analyzed. As shown in the table, the average time spent in our analyzer to inspect each path—with little time variations across paths—is relatively low (ranging from 2 ms to 5 ms). This demonstrates that our optimizations—pre-normalizing the CFG and recording direct forward edges in the LBR—are effective in implementing a scalable context-sensitive path analysis even for programs with a large and complex CFG. In addition, the total time spent in our analyzer is marginal compared to the total benchmark run time (24 ms to 140 ms vs. several seconds). This shows the effectiveness of our path cache which, as also reported in Table 2.2, was consulted thousands of times with only dozens of misses for most applications. We elaborate on the end-to-end impact of our on-demand path analysis strategy on runtime performance in the next section.

Table 2.3. Runtime performance results and statistics collected at runtime for our set of server programs. The **Normalized runtime** group shows the overhead for a number of incremental configurations: (1) save and restore LBR registers during context-switches, (2) added library instrumentation to disable LBR tracking when executing library code, (3) added callback instrumentation to temporarily re-enable LBR tracking when library code calls back to the program, and (4) added path verification. The **Event statistics** group show details on the number observed library calls, system calls and signals at runtime.

	Normalized runtime				**Event statistics**		
Server	*LBR only*	*+ LInstr*	*+ CBInstr*	*+ PathVer*	*# LCalls*	*# SCalls*	*# Signals*
exim	1.025	1.019	1.036	1.079	67,849	4,149	50
lighttpd	1.097	1.236	1.226	1.275	1,209,081	200,564	0
nginx	1.053	1.178	1.168	1.174	1,500,021	200,002	0
openssh	1.003	1.003	1.031	1.020	24,313	720	8
proftpd	1.000	1.000	1.000	1.000	171,440	48,562	6
pure-ftpd	1.003	1.053	1.031	1.074	115,897	57,843	64
vsftpd	1.000	1.000	1.000	1.000	35,883	42,446	208

2.5.3 Runtime Performance

To evaluate the impact of PATHARMOR's instrumentation and path verification strategy on runtime performance, we measured the time to complete the execution of our benchmarks and compared against the baseline. Table 2.3 presents our results. The **Normalized runtime** group details the runtime overhead for a number of configurations. First, *LBR only* refers to configuring our kernel module to solely save and restore LBR contents during context switches. As shown in the table, this introduces marginal performance impact (0% to 10% in the worst case). The overhead is somewhat more pronounced in the *+ LInstr* and *+ CBInstr* configurations (6.6% and 6.7%, geometric mean), which additively account for our library entry point and callback instrumentation (respectively), but omit the path verification step in our kernel module. The *+ PathVer* configuration, finally, refers to the default PATHARMOR setup, enabling full instrumentation and path verification using our on-demand static analyzer. As shown in the table, our cache-aware path analysis has relatively little impact on runtime performance (+1.7%, geometric mean), resulting in an average runtime overhead of 8.5% (geometric mean).

To shed some light on the key factors contributing to the performance overhead, we also instructed PATHARMOR to report statistics on the runtime events of interest, as shown in the third group of columns in Table 2.3. Our results confirm that library calls (*# LCalls*) are the most prevalent contributing factors in the mean

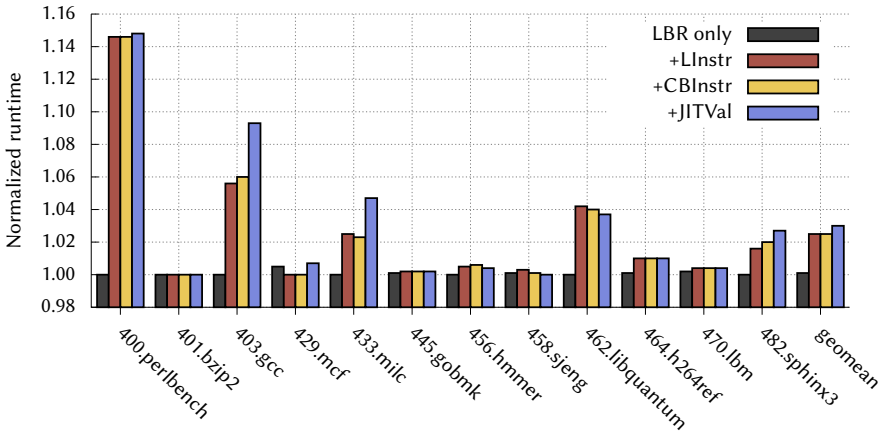


Figure 2.6. Normalized runtime for four incremental PATHARMOR configurations (only LBR tracking, added library instrumentation, added callback instrumentation, and full protection) for all C programs in SPEC CPU2006.

case, also inducing the worst-case performance impact on lighttpd (27.3%). More aggressively instrumented operations like callback invocations (marginal, not reported in table), sensitive function calls (#*SCalls*) and signals (#*Signals*) have a less prominent impact and can thus be better amortized over the execution.

To obtain standard and comparable performance results across PATHARMOR's configurations, we also measured the time to complete all the C programs in SPEC CPU2006 and compared against the baseline. Figure 2.6 presents our findings. Our results confirm the general behavior observed for our server applications, but the performance overhead is generally much lower (3% in PATHARMOR's default configuration, geometric mean). This result stems from the lower number of library and system calls issued by SPEC programs, as expected for standard CPU-intensive (as opposed to syscall-intensive) benchmarks.

Overall, PATHARMOR imposes a relatively low runtime performance impact on all the test programs considered. This confirms that PATHARMOR's lightweight instrumentation and cache-aware path analysis are successful in producing a runtime overhead comparable to the most efficient (source-level and forward-edge only) $\overline{\text{C}}$CFI techniques [130], while enforcing much more advanced context-sensitive CFI policies on both the forward and backward edge and operating entirely at the binary level.

Table 2.4. LBR pollution caused by library calls in SPEC CPU2006. **#Library calls** lists the total number of library calls that were executed at runtime, **#Polluted entries** shows how many LBR entries got polluted by library code in total, and **Fraction** shows how much history is lost per library call on average.

Benchmark	#Library calls	#Polluted entries	Fraction
400.perlbench	60,495,412	253,246,721	26.16
401.bzip2	449	1,284	17.87
403.gcc	3,373,862	13,086,146	24.24
429.mcf	470,597	5,705,524	75.78
433.milc	28,807,387	65,657,612	14.24
445.gobmk	299,877	1,004,581	20.94
456.hmmer	4,098,071	18,395,790	28.06
458.sjeng	11,602	176,683	95.18
462.libquantum	52,609,059	105,222,996	12.50
464.h264ref	2,449,569	12,515,117	31.93
470.lbm	2,626,460	5,263,308	12.52
482.sphinx3	48,625,654	187,711,907	24.13

2.5.4 LBR Pollution

As discussed in Section 2.3, PATHARMOR's design supports two modes of operation: (1) stop tracking branches at the library level, or (2) continue tracking within libraries. The current implementation of PATHARMOR uses the first mode by default, effectively increasing the control flow context of the protected binary during path verification. To also protect against control flow diversion triggered within library code, PATHARMOR can be configured with the second mode of operation. When running in this mode, branch tracking is never disabled at the cost of (partially) "polluting" the LBR from (self-contained) library code.

To evaluate the LBR pollution cost of running in full-library mode, we configured PATHARMOR to compare LBR contents right before and right after each library call and reran the SPEC CPU2006 benchmark. Table 2.4 shows the results. The average pollution rate of 25.68% overall (geometric mean) is likely acceptable in environments where untrusted, potentially vulnerable libraries are in place.

Tracking inside libraries leads to better performance, as this removes the jump to kernel during program-library transitions. Thus, as mentioned earlier, the results provided in this section show worst-case performance. As discussed above, the trade-off of in-library tracking is increased LBR pollution, which, however, can also be mitigated with complementary techniques, such as inlining library code or using hardware that provides a larger branch record.

2.5.5 Memory Usage

PATHARMOR instrumentation increases memory usage at runtime. To evaluate this impact, we measured the physical memory used by instrumented applications compared to the baseline. Deploying our kernel module alone has a constant and marginal memory usage impact (+1 MB). Our static analyzer, in turn, yields a memory usage impact proportional to the size of the CFGs under active analysis, resulting in an increase of +18-74 MB across all our applications.

More important is to assess the memory usage impact of our path caching strategy, given that caching static analysis results is important to minimize the performance impact on instrumented applications. Encouragingly, our measurements indicate a very small memory usage impact induced by our in-kernel path cache, resulting in a worst-case increase of only 2 KB across all our applications during the execution of our benchmarks. This suggests that our path caching strategy is practical even for applications which periodically issue several different sensitive function calls, and even provides evidence that deploying a system-wide path cache that persists across application restarts (thus eliminating cache warmup-phase penalties for applications with strong real-time guarantees) may be a realistic option.

2.6 Discussion

This chapter outlined and evaluated the design decisions made in PATHARMOR. We now discuss evasion techniques an attacker may employ to bypass PATHARMOR, analyzing their impact and the limitations of our current solution.

2.6.1 History-Flushing Attacks

An attacker may attempt to mount a *history-flushing attack* to clear any traces of a ROP chain from the LBR. History-flushing attacks previously described in the literature first execute 16 innocuous *NOP-like gadgets* followed by a long *termination gadget* that restores argument registers and ultimately performs a security-sensitive system call [26]. The long termination gadget bypasses heuristics used in prior LBR-based solutions such as kBouncer [102] and ROPecker [30], which rely on weak security invariants based on gadget size (which they assume to be small) and frequency.

PATHARMOR is not vulnerable to this simple attack, as history flushing in PATHARMOR is equivalent to the attacker crafting a valid CCFI-permitted path of 16 NOP-like gadgets (using direct calls or indirect branches). This is much

more difficult than chaining arbitrary and CFG-agnostic gadgets. In other words, the notion of a path in PATHARMOR is stronger than that of regular (context-insensitive) CFI and much stronger than that of kBouncer and ROPecker. Hence, while history-flushing attacks remain of concern, PATHARMOR's stronger invariants significantly raise the bar for the attacker. For example, we have shown in Section 2.5.1 that it is generally much harder to maintain register states over that many branches.

A related attack is to force context switches to clear the LBR and indirectly mount a history-flushing attack. This attack is also ineffective against PATHARMOR, given that, as outlined in Section 2.4, PATHARMOR stores and restores LBR states during context switches on a per-thread basis.

2.6.2 Non-Control Data Attacks

An attacker may attempt to mount a non-control data attack to indirectly influence the execution of existing security-sensitive functions in the program without directly diverting control flow. For example, an attacker can exploit an arbitrary memory write vulnerability to overwrite sensitive function arguments that are maintained in a data region. Similarly to all the existing (and even ideal) CFI solutions, PATHARMOR cannot protect against these and other data-only attacks. Unlike existing whole-program CFI solutions, however, PATHARMOR's history-based strategy would also allow an attacker to craft a ROP-based memory write primitive before jumping to the beginning of a valid execution path leading to a security-sensitive function. Nevertheless, since ROP is not necessary to perform an attacker-controlled memory write and arbitrary memory write vulnerabilities are actually very common, we do not believe this is a limiting factor within our threat model. We also note that binary-level defenses against non-control data attacks are explored in orthogonal work [125].

2.6.3 Endpoint-Pruning Attacks

An attacker may attempt to evade detection by avoiding calls to sensitive endpoints recognized by PATHARMOR. This is because, similarly to prior endpoint-driven solutions [30, 102], PATHARMOR enforces security invariants only at predetermined sensitive function calls. Assuming PATHARMOR's default configuration, such *endpoint-pruning* attacks require the attacker to find alternative means to affect the system environment without relying on system calls such as `exec`, and `mprotect`. While this is generally of concern depending on the goals of the attacker, PATHARMOR allows users to configure the list of sensitive endpoints ac-

cording to their needs. For programs in which our default configuration is not sufficient to provide the required guarantees, users can custom tune the list of endpoints and balance security and runtime performance.

Nevertheless, we believe that PATHARMOR's default configuration alone drastically reduces the freedom of an attacker. Although ROP may still be used to perform arbitrary Turing-complete computations, without the ability to execute core security-sensitive system calls, the impact on the system remains limited.

2.6.4 Instrumentation-Tampering Attacks

An attacker may attempt to abuse the instrumentation employed by PATHARMOR's default mode of operation (which disables branch tracking in library code) to alter the branch record. Nevertheless, this attack would still fail to circumvent PATHARMOR's detection strategy. Consider the scenario wherein an attacker sets up a ROP chain that invokes the `ioctl` system call with a dedicated PATHARMOR-specific argument to tamper with the branch-tracking instrumentation. Depending on the request type, this attack will result in two possible outcomes. In the case of a `CALLBACK_EXIT` request, PATHARMOR's kernel module will immediately verify the current LBR state (see Section 2.3.3) and detect CCFI invariants violations caused by the originating ROP-based control flow. In the case of a `LIB_ENTER` request, in turn, PATHARMOR's kernel module will immediately return control to userland after disabling branch tracking, allowing the attack to resume in LBR-free execution. As soon as the attacker invokes a security-sensitive function, however, PATHARMOR's kernel module will perform verification as normal. At that point, the LBR state will still reflect the branch record generated by the attacker's original ROP chain (leading to the previously issued `ioctl` system call), resulting, again, in PATHARMOR detecting the attack. Note that an attacker can also attempt to later re-enable branch tracking via a `LIB_EXIT` operation, but a PATHARMOR-legal path of 16 indirect branches is then required to clear any traces of the original ROP attack—essentially equivalent to the history-flushing attacks discussed earlier.

2.7 Related Work

CFI was originally proposed by Abadi et al. [2]. The original (strict) CFI proposal incurs high overheads. This has lead to a myriad of proposals for practical CFI implementations which realize better performance by strategically trading off security guarantees. There are two broad branches of CFI implementations: (1) Control-Flow Graph-based (CFG-based) CFI, and (2) Heuristic-based CFI.

CFG-based CFI focuses on enforcing properties of the CFG. Compiler-based approaches inherently require source to resolve (indirect) control transfers that are considered legitimate [2, 5, 16, 40, 47, 96, 141, 151]. Due to the availability of source information, these approaches are usually able to derive accurate CFGs. Binary-based approaches, while potentially less accurate (i.e., based on an over-approximated CFG), have the advantage of being applicable to legacy programs where the source code is not available [77, 145, 154, 156, 157]. Recently, modular CFI approaches have also been proposed. These are a variant of CFG-based approaches, which resolve part of the CFG at runtime, providing greater flexibility for dynamically computed targets [97, 98, 105].

In contrast to CFG-based CFI, heuristic-based CFI does not require a CFG to enforce integrity. Such approaches include kBouncer [102] and ROPecker [30], which seek to detect anomalous control patterns at sensitive program points. Such approaches are easy to deploy, but are also relatively easy to circumvent, due to their heuristics [57].

Prior work explored devastating attacks against both prior CFG-based and heuristic-based CFI, using combinations of individually legal control transfers [26, 43, 57]. PATHARMOR enables stronger defenses against such attacks by efficiently enabling context-sensitive CFI policies over paths to sensitive functions and disallowing many unnecessary forward and backward edges permitted by prior context-insensitive CFI policies (e.g., backward edges to arbitrary call-site gadgets, a common attack target [57]).

In prior fine-grained CFI techniques, context-sensitive policies have been explored only for backward edges and only using shadow stacks [13, 29, 31, 35, 41, 47, 111, 115, 184, 150].

In contrast to the runtime shadow stack approach, PATHARMOR resolves backward edges using a hardware-supported context-sensitive static analysis over the interprocedural CFG and caches the results at sensitive points in the program, yielding improved performance and security against tampering attacks. Static context-sensitive backward edge resolution strategies have been explored before for security, but only to improve the accuracy of IDS models based on syscall sequences [138]. PATHARMOR, in contrast, shows that enforcing context-sensitive CFG-based policies both on the forward and backward edge at a much finer level of granularity (i.e., control-flow transfers for CFI) is a realistic and efficient option thanks to emerging hardware features. This result contrasts claims in prior work, which, while acknowledging their security advantages, generally dismissed context-sensitive CFI policies as impractical for real-world adoption [2].

Other approaches rely on hardware-supported branch tracing to improve CFI

performance. Similar to PATHARMOR, kBouncer [102] and ROPecker [30] rely on Intel's LBR to efficiently implement branch tracing, but only to enforce heuristic CFI policies which can be easily circumvented [26]. CFIMon [145] can enforce hardware-supported CFG-based CFI policies, but relies on the significantly slower Intel BTS [102] and yields high detection latencies, potentially missing attacks [30]. Unlike PATHARMOR, none of these approaches attempt to enforce context-sensitive policies over hardware-monitored control transfers.

Concurrently with our work, Schuster et. al. have developed the COOP attack [120], and show that CFI solutions that do not precisely consider object-oriented semantics in C++ programs can generally be bypassed. While our work mainly focuses on C rather than C++ programs, we believe CCFI can strengthen forward-edge invariants (subject to the precision of the underlying data-flow analysis) in modern vtable protection techniques in mainstream compilers [130], raising the bar against COOP-like attacks.

The recent Control-Flow Bending (CFB) [25] evaluates the general effectiveness of even ideal (context-insensitive) CFI solutions and evidences their limitations against sophisticated CFG-aware attacks. Compared to regular CFI, CCFI makes such attacks harder, given that entire paths (rather than individual CFG edges) are checked for validity. CFB attacks have already been shown to be more difficult against CFI solutions that are complemented by a shadow stack [25]. Compared to such solutions, CCFI does not rely on in-process runtime information and can enforce context-sensitive invariants on both forward and backward edges, thereby providing improved defenses against CFB attacks.

2.8 Conclusion

While Context-sensitive CFI (CCFI) can significantly enhance the security of state-of-the-art defenses against control-flow diversion attacks, it has long been perceived as inefficient and impractical for real-world adoption. This chapter has shown that the three fundamental challenges towards fast and practical CCFI—efficient path monitoring, analysis, and verification—can indeed be effectively addressed in a realistic way on commodity platforms.

To substantiate our claims, we implemented PATHARMOR, the first binary-level CCFI solution that efficiently enforces context-sensitive CFI policies on both backward and forward edges. PATHARMOR addresses all the CCFI fundamental challenges using low-overhead hardware registers to track control edges, a scalable on-demand and constraint-driven context-sensitive static analysis, and a path cache accessed at sensitive program points. PATHARMOR yields compa-

rable or better performance than prior context-insensitive CFI solutions, while enforcing much stronger context-sensitive invariants and providing a general framework to implement arbitrarily sophisticated CCFI policies.

Shared Authorship

I share first authorship on PATHARMOR with Dennis Andriesse. Dennis is the main author of the verification/static analysis component, while my focus was on the kernel module and runtime implementation.

3 A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level

Binary-level Control-Flow Integrity (CFI) techniques are weak in determining valid targets for indirect control flow transfers on the forward edge. The lack of source code forces existing techniques to resort to a conservative address-taken policy that results in overapproximation. Source-level solutions can accurately infer the targets of indirect callsites and thus detect malicious control-flow transfers more precisely. Since source code is not always available, however, offering similar quality of protection at the binary level is important, but also more challenging than ever: recent work demonstrates powerful attacks, such as *Counterfeit Object-oriented Programming* (COOP), which made the community believe that mitigating control-flow diversion attacks at the binary level is impossible.

In this chapter, we propose binary-level analysis techniques to significantly reduce the number of possible targets for indirect callsites. More specifically, we reconstruct a conservative approximation of target function prototypes by means of use-def analysis at possible callees. We then couple this with liveness analysis at each indirect callsite to derive a many-to-many relationship between callsites and target callees with a much higher precision compared to solutions.

Experiments on server applications and SPEC CPU2006 show that TypeArmor, a prototype implementation of our approach, is efficient—with a runtime overhead of less than 3%. We evaluate to what extent TypeArmor can mitigate COOP and other advanced attacks and show that our approach reduces the number of targets on the forward edge. Moreover, we show that TypeArmor stops published COOP exploits, providing concrete evidence that strict binary-level CFI can still mitigate advanced attacks, despite the absence of source information or C++ semantics.

3.1 Introduction

Control-Flow Integrity (CFI) [2] is one of the most promising ways to stop advanced code-reuse attacks. Unfortunately, enforcing it without access to source code is hard in practice. The reason is that it requires an accurate Control-Flow Graph (CFG) and extracting such CFG from binary code is an undecidable problem. As a result, most existing binary-level CFI implementations base their invariants on an *approximation* of the CFG which leaves enough wiggle room for an attacker to launch successful exploits [26, 27, 43, 57, 58, 121].

While it may be possible to stop some advanced attacks using a perfect shadow stack implementation [25], there is one class of attacks for which there is no existing defense at the binary level whatsoever. This class of *function-reuse* attacks, exemplified by Counterfeit Object-Oriented Programming (COOP) [120], chains together calls to *existing* functions through *legitimate* callsites. This strategy preserves the integrity of the shadow stack, while abusing the overapproximation of the extracted CFG to call the *wrong* functions from these callsites. This attack is powerful since it allows for exploits that integrate smoothly with legitimate code execution. Unless there is deep knowledge of the C++ class hierarchy semantics, which we can only extract if we have the source code [39], it is hard to tell a COOP exploit apart from a legitimate code sequence [120]. Lacking a handle on the functions that a callsite may target leaves all the existing binary-level CFI solutions unable to stop COOP attacks.

In this paper, we revisit binary-level protection in the face of COOP attacks and follow-up improvements [39]. We explore to what extent we can narrow down the set of possible targets for indirect callsites and stop exploitation at the binary level. Our conclusion is *not* that all possible attacks can be stopped: even the tightest CFI solutions *with* access to source code are unable to guarantee perfect protection against all possible attacks [25]. Nevertheless, we demonstrate that TYPEARMOR, our binary-level protection prototype, can stop all COOP attacks published to date and significantly raise the bar for an adversary. Moreover, TYPEARMOR provides strong mitigation for many types of code-reuse attacks (CRAs) for programs binaries, without requiring access to source code. As researchers have shown that it is easy to bypass existing binary-level CFI defenses [26, 27, 43, 57, 58, 121], this is a significant improvement.

TYPEARMOR incorporates a forward-edge CFI strategy that relies on conservatively reconstructing both callee prototypes and callsite signatures and then uses this information to enforce that each callsite *strictly* targets matching functions only. For example, TYPEARMOR disallows an indirect call that prepares fewer ar-

guments than the target callee consumes. Additionally, TYPEARMOR incorporates a novel protection policy, namely CFC (*Control-Flow Containment*), which further reduces the possible target set of callees for each callsite. CFC is based on the observation that, if binary programs adhere to standard calling conventions for indirect calls, undefined arguments at the callsite are not used by *any* callee by design. TYPEARMOR trashes these so-called *spurious arguments* and thus breaks all published COOP and improved COOP-like exploits. These exploits all chain virtual method calls that disrespect calling conventions to achieve convenient data flows between gadgets [39]. CFC eliminates these data flows via unused argument registers and thus stops such exploitation attempts.

Current binary-level solutions enforce "loose" forward-edge CFI policies, often allowing control transfers from any valid callsite to any valid referenced entry point [154, 157]. In the best case, existing policies only reduce the target set by removing all entry points of other modules unless they were explicitly exported or observed at runtime [105]. In contrast, TYPEARMOR matches up indirect callsites with a more precise target set in a many-to-many relationship. It relies on use-def analysis at all possible callees to approximate the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This effectively leads to a more precise CFG of the binary program in question, which could also be used by existing mitigation systems to amplify their (context-insensitive) invariants (e.g., PathArmor [131]).

Can TypeArmor defend against any exploit? No. TYPEARMOR protects only forward edges at the binary level. As shown by previous work, a backward-edge protection mechanism (e.g., a shadow stack [41] or context-sensitive CFI [131]) is still essential to ensure the integrity of return addresses at runtime [25, 57]. In this paper, we assume an ideal backward-edge protection mechanism such as a shadow stack with no design faults [34]. TYPEARMOR complements such backward-edge protection by countering attacks that take place without violating the integrity of the return path. Specifically, TYPEARMOR provides strong (but not infallible—given also the fundamental CFI limitations [25]) protection against COOP exploits as well as improved COOP-like exploits [39] and similar advanced attacks such as Control Jujutsu [49].

Is TypeArmor superior to approaches like IFCC/VTV and CPI? No. IFCC/VTV and CPI [83, 130] are strong source-level defenses which produce binaries that can resist control-flow hijacking attacks. Source-based techniques are more precise in using fine-grained program constructs (such as the C++ class hierarchy or generic data types) for mitigation purposes. However, there are still important reasons to study and improve binary-level defenses. First, the source code

for many off-the-shelf programs is not always available. Second, real-world programs rely on a plethora of shared libraries and recompiling all shared libraries is not always possible. This is true even for purely open-source projects. For example, in VTV [130], the authors evaluate their system on ChromeOS, which includes legacy libraries. The authors had to manually whitelist these libraries, a task which is not trivial (certain code has to be annotated) and does not scale. Third, even if the source code of the libraries is available, recompiling big projects with dynamic dependencies is, again, a demanding task. Even state-of-the art defenses that enforce CFI policies at the source level such as SAFEDISPATCH [66] do not support dynamic libraries. Note that this is not a minor issue: mixing CFI-protected with non-protected code is impossible. If CFI is applied in just a portion of the CFG, crashes due to legitimate execution are possible. In contrast, with a binary-level solution, we can offer strong protection even if the source code is not available or when recompilation is not feasible (or desirable).

In summary, we make the following contributions in this paper:

- We introduce techniques to recover callsite signatures and callee prototypes for security enforcement purposes. Our techniques yield binary-level control-flow invariants which approximate the type-based invariants used in source-level solutions [130] and are thus much more precise than those used in prior binary-level CFI solutions [105, 154, 157].
- We demonstrate that fully-precise, static forward-edge CFI is inherently hard to achieve in a conservative fashion, due to the unavoidable precision loss when deriving traditional CFI-style target-oriented invariants at the binary level. To compensate for the precision loss, we complement our CFI strategy with a new technique termed *Control-Flow Containment* (CFC). CFC relies only on our callsite analysis to effectively *contain* code-reuse attacks. This approach improves the quality of control-flow invariants of traditional target-based approaches, overall resulting in a strict binary-level CFI strategy.
- We implement and evaluate TYPEARMOR, a new strict CFI solution for x86_-64 binaries. Our experimental results demonstrate that TYPEARMOR can enforce much stronger forward-edge invariants than all the existing binary-level CFI solutions, while, at the same time, introducing realistic runtime performance overhead ($< 3\%$ on SPEC).
- We show that our strict binary-level CFI strategy can mitigate advanced attacks in complete absence of source information or C++ semantics. For

example, TYPEARMOR can stop all published COOP [120] exploits and their improvements [39].

The remainder of this paper is organized as follows. We start with a more detailed discussion of our main goal: mitigating COOP-like attacks at the binary level. Section 3.2 provides a short introduction of how COOP works and Section 3.3 presents an overview of how TYPEARMOR is designed to mitigate COOP attacks. Section 3.4 and 3.5 present TYPEARMOR internals. Section 3.6, Section 3.7, and Section 3.8 evaluate TYPEARMOR's performance and security. Finally, Section 3.9 surveys related work and Section 3.10 concludes the paper.

3.2 Motivation: Key Requirements for COOP

Counterfeit Object-Oriented Programming (COOP) is a novel attack technique that belongs to the class of code-reuse attacks (CRAs) [120]. While the core ideas have general applicability, the attack strategy described in [120] relies on Object Oriented Programming (OOP) principles and mainly targets C++ applications. In contrast to many proposed CRAs, COOP makes the exploit's control flow more akin to a benign execution flow. In this section, we summarize the technique with a focus on its key requirements: the ability to target unrelated virtual functions from an indirect callsite, and especially to pass data from one COOP gadget to another. In the next section, we show how TYPEARMOR impacts the attacker's possibilities to satisfy these requirements.

By exploiting a memory corruption bug, COOP diverts execution flow to a chain of *existing* virtual function calls (so called *vfgadgets*) via an *initial vfgadget*. In practice, an attacker can control said virtual function calls by injecting multiple, attacker-controlled *counterfeit objects* that reuse *existing* vtables in the binary. By choosing the correct object layout and *overlapping* multiple objects, an attacker can ensure intended data flows between different gadgets.

The original COOP paper [120], along with its improvement [39], proposes two main types of initial vfgadgets: (1) the main-loop gadget (ML-G) and (2) the recursive gadget (REC-G). Such gadgets are responsible for dispatching the vfgadget chain using virtual function calls. The former depicts "[a] virtual function that iterates over a container [...] of pointers to C++ objects and invokes a virtual function of these objects" [120]. The latter, in turn, requires at least two consecutive virtual function calls on distinct (counterfeit) objects. The first call dispatches a vfgadget, whereas the last recurses into (any) REC-G.

Proper use of object overlapping may enable an attacker to pass data through object fields, if applicable. For example, one vfgadget may write to and another

gadget then reads from, the same object field. In this paper, we refer to this strategy to pass data between vfgadgets as an *explicit* data flow. Schuster et al. found that cases that allow for explicit data flows are "rare in practice." [120]. Other approaches focus on the calling convention assumed by the indirect call that dispatches the vfgadgets. The ability to pass data to a vfgadget then depends on the choice of the ML-G, or REC-G, respectively. In the case of x86_64 calling conventions, the first six arguments are passed through registers (assuming System V ABI). These registers are *scratch* registers that are not preserved by a function. Consequently, if the ML-G or REC-G does not destructively update one of these registers in between virtual function calls, changes made to such a register by a vfgadget are implicitly passed to the next gadget. In other words, they represent an *implicit* data flow. Similar approaches for other platforms exist as well, for which we refer the reader to the original paper [120].

3.3 Overview

In this section, we first outline the threat model and assumptions under which TYPEARMOR operates. We then give a high-level overview of TYPEARMOR and discuss the impact of TYPEARMOR's measures on COOP exploits.

3.3.1 Threat Model and Assumptions

We assume a common threat model where an attacker can read/write the data section and read/execute the code section of a vulnerable program. The program does not contain self-modifying code, $W \otimes X$ is in place, and the attacker is able to hijack the program's control flow by means of a memory-corruption vulnerability. We seek to defend against attacks with a binary-level version of (forward-edge) Control-Flow Integrity (CFI) [2]. In other words, our solution should support legacy binaries without access to source or debug symbols. In doing so, we focus on 64-bit binaries and analyze only function parameters that are passed via registers (those passed on the stack are conservatively handled). Depending on the ABI, this gives TYPEARMOR the capability to track at most 4 (in the case of Microsoft's x64 calling convention) or 6 (System V ABI) arguments. For simplicity, our implementation currently does not take floating-point arguments passed via `xmm` registers into consideration; future work may improve TYPEARMOR by extending static analysis to also include these registers. Nevertheless, as we show in Section 3.6, this still gives us enough information to stop even state-of-the art code-reuse attacks.

Obfuscated or hand-crafted binaries are out of scope and we assume an orig-

inating compiler that generally adheres to one of the standard calling conventions (to allow our static analysis to derive meaningful invariants), but can also occasionally resort to custom calling conventions for functions which are not externally visible due to standard compiler optimizations (which our analysis can conservatively handle). We discuss compiler optimizations in more detail in Section 3.4.2, illustrating how TYPEARMOR can support optimizations from standard compilers and how it can be also extended to support optimizations from non-standard (or future) compilers. We stress that the current TYPEARMOR prototype works on stripped binaries that have been compiled using different optimization levels (namely `-O0`, `-O1`, `-O2`, and `-O3`).

3.3.2 TypeArmor: Invariants for Targets and Callsites

TYPEARMOR deploys a combination of two type-based control-flow invariants, resulting in a strict forward-edge protection strategy: *target-oriented invariants* and *callsite-oriented invariants*. Target-oriented invariants are based on traditional CFI policies [2], but callsite-oriented invariants have not been explored for binaries before. Specifically, TYPEARMOR enforces callsite-oriented invariants through a novel containment technique which we term Control-Flow Containment (CFC). As noted above, extracting complete function and callsite type information at the binary level is hard in practice, and impossible in the general case. Therefore, TYPEARMOR relies on a *relaxed* form of type information (argument count and return value use), and enforces a many-to-many type-based matching strategy between callsites and targets. TYPEARMOR applies such type-based invariants, inspired by source-level CFI techniques [130], at the binary level for the first time.

In particular, TYPEARMOR ensures that indirect callsites that set at most max arguments cannot target functions that use more than max arguments. For instance, if TYPEARMOR finds a callsite that prepares at most 2 arguments, it ensures that the callsite can never jump to a function that consumes 3 targets or more. Additionally, TYPEARMOR ensures that indirect callsites that expect a return value (non-void callsites) can never jump to a callee that does not prepare such value (void functions). Enforcing such invariants at the binary level is challenging and subject to the precision of argument count and return use information derived by static analysis at *both* the callsite and at the target function.

While CFI's target-oriented invariants seek to identify the target set for each callsite, CFC follows a completely target-agnostic approach and thus is subject to the precision of argument count information *only* at the callsite. CFC relies on callsite-oriented invariants to scramble all the unused function arguments at

every callsite, so that illegal (type-unsafe) function targets are not inadvertently exposed to stale (and potentially attacker-controlled) arguments. Similarly, at the callee, CFC is caller-agnostic and relies on liveness analysis to detect void functions. For these, TYPEARMOR scrambles unused return registers before the function returns. This strategy disrupts many type-unsafe function argument reuse attempts, which are required by existing COOP exploits.

Note that, in order to be conservative and support existing program functionality, TYPEARMOR's callsite analysis is may only report an *overestimation* of the number of prepared arguments, while the callee analysis should report only *underestimation*. As an example, consider a callsite cs that prepares 3 arguments and a callee f that consumes 3 arguments. TYPEARMOR may detect that cs prepares 4 arguments and f only uses 2 arguments. TYPEARMOR's invariants dictate that, in this scenario, cs is still allowed to call f. Examples of how callsite overestimation and callee underestimation occur are further discussed in Section 3.4.

TYPEARMOR uses static analysis results to enforce control-flow invariants at runtime. The enforcement component relies on binary rewriting supported by the Dyninst binary analysis framework [11] to enforce CFI, CFC, or both (default configuration).

CFI TYPEARMOR relies on the caller-to-callee mapping derived by our target-oriented invariants analysis. For this purpose, TYPEARMOR instruments each function according to its type and each indirect callsite to check if it calls the appropriately typed function. In contrast to source-level type-based CFI solutions [130], which benefit from one-to-one (i.e., precise function signature) mappings to detect type-incompatible targets, TYPEARMOR relies on a many-to-many mapping to sidestep the problem of identifying precise function signatures at the binary level—infeasible in general [110]. This strategy effectively results in a hierarchical function type structure when checking target-oriented invariants, as exemplified in Figure 3.1. As shown in the figure, the first callsite (at the top) passes 3 arguments to the callee, which thus belongs to set T_3 (also including sets T_0, T_1 and T_2). The second callsite (at the bottom), in contrast, passes only 1 argument to the callee and thus requires that the callee belongs to the set T_1 (also including T_0).

Note that the invariant that a non-void callsite cannot call a void function (omitted from Figure 3.1 for simplicity) doubles the number of function types: the set of functions that a particular non-void callsite may target is a subset of the possible targets for void callsites. This is because, at the binary level, it is only possible to determine potential non-void callsites. If our analysis finds that

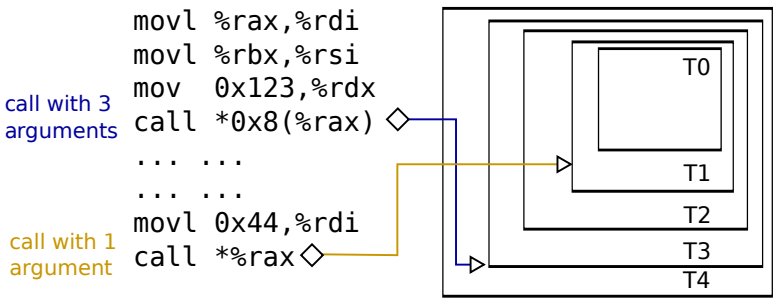


Figure 3.1. Hierarchical structure of binary-level function types. T_i is a set of functions that take i or less arguments. The first indirect call instruction (with 3 prepared arguments) may call functions in T_3 (which also include functions that are in T_2, T_1, and T_0), while the second indirect call instruction (preparing only 1 argument) may only target functions that are in T_1 (which includes those that are in T_0).

a callsite is not non-void, it cannot guarantee that this is a void callsite (the caller may call a non-void function, but never use its return value).

CFC TypeArmor relies on the caller-to-type mapping derived by the callsite-oriented invariants analysis. For this purpose, it instruments each indirect callsite to scramble unused arguments before transferring control to the callee and instruments each void function to scramble unused return arguments before transferring control back to the caller.

A thorough analysis of TypeArmor's static analysis is presented in Section 3.4, while we discuss the runtime component in Section 3.5.

3.3.3 TypeArmor's Impact on COOP

TypeArmor's CFC enforces a maximum number of arguments prepared at a callsite and scrambles the unused registers. This severely impacts the ability of an attacker to enable data flow between gadgets.

As discussed in the original COOP paper [120], data flow via object fields is hard to achieve in practice due to a lack of useful gadgets. Instead, in the case of the x86_64 System V ABI, Schuster et al. suggest using unused argument registers to achieve data flow between indirect calls in the ML-G or REC-G, respectively. This only works if the invoking gadget does not update the register destructively. However, CFC is *explicitly* designed to introduce destructive updates of unused argument registers before an indirect call and mitigates this data-passing strategy. Furthermore, TypeArmor's CFI implementation reduces the target set of the virtual function calls by the main-loop and recursive gadgets considerably. It

prohibits any forward edges to functions that expect more arguments than the callsite prepares.

Needless to say, both aspects rely on the accuracy of TYPEARMOR in terms of callsite coverage in general and argument count identification for both callsites and target functions. Hence, implementing TYPEARMOR at the binary level is challenging from a research point of view and never as accurate as source-level solutions. However, we will show that it is effective in practice. In the next two sections, we look at the static analysis and dynamic enforcement of TYPEARMOR's invariants.

3.4 Static Analysis

Static analysis in TYPEARMOR seeks to detect (1) the maximum number of prepared arguments at indirect callsites, (2) the minimum number of consumed arguments at possible callees, and (3) the preparation (callees) and expectation (callsites) of return values. Since TYPEARMOR targets binaries, the analysis works on disassembled code. For this purpose, we leverage the Dyninst binary analysis framework which is capable of constructing Control-Flow Graphs (CFGs) for both program binaries and libraries [11].

3.4.1 Callee Analysis

We use static analysis to determine the argument count at the callee side. Given a set of address-taken (AT) functions[1], TYPEARMOR iterates over each function and performs inter-procedural *liveness* analysis [171]. The analysis focuses on collecting state information on registers to determine if they are used for passing arguments or not. For a given path of instructions or basic blocks according the CFG, a register can be in one of the following states: *read-before-write* (R) (data are always read from this register before new data are written to it), *write-before-read* (W) (this register is always written to before it is read), or *clear/untouched* (C) (this register is never read or written to). The state of a particular basic block contains the combined register state for all argument registers. The analysis starts at the entry block of an AT function and iterates over the instructions to determine the usage of registers. If all argument registers are either R or W, the analysis terminates. However, if at least one register is in a C state, a recursive *forward analysis* starts until the block has no outgoing edges. Note that the analysis takes special care about variadic functions, which we discuss below.

[1] A function f is defined to have its address taken if there are one or multiple instructions in the binary that load the entry point of f into memory. By definition, indirect calls can only target AT functions.

Forward analysis A recursive analysis loops over all outgoing edges of the basic block to get a pointer to the next basic block to analyze. We distinguish between direct calls, indirect calls, return instructions, and regular outgoing edges (e.g., jump instructions). Depending on the edge type, different operations are performed.

For **direct calls**, the next basic block to analyze is the entry block of the target function. We also retrieve the *fall-through* basic block for this instruction, which is the block to be executed after the direct call returns. For each direct call, we push the fall-through block on a stack that TYPEARMOR maintains, which we later use to analyze return instructions (see below). In the case of direct calls that never return (e.g., calls to functions that exit), we do not retrieve a fall-through block. We detect such calls by checking whether they target a known function that exits (e.g., `exit@plt`). This analysis is again recursive so that we can correctly wrappers around `exit` as non-returning functions.

The analysis cannot statically infer the target of the **indirect calls** and we thus have to be conservative. We assume that the target writes all arguments and stop the recursion, transforming all remaining *clear* registers into a *write-before-read* state.

For **return instructions**, we pop a fall-through basic block from the stack and use it as the next basic block in the analysis. An empty stack indicates the end of the analyzed function and terminates the recursive analysis.

We handle **other edge types** (including indirect jumps, for which we rely on Dyninst to resolve its targets) in the same way: the targets of the edge are set as the next basic blocks in the analysis.

Finally, to avoid loops during the analysis, we keep track of all blocks analyzed so far. When the analysis is about to recurse, we check whether we already analyzed the next basic block, and if so, continue with the next edge. In addition, we use a cache to avoid multiple analysis passes on the same basic block. Notice that the latter is just an optimization for speeding up the analysis (which is offline), and it does not affect the accuracy of the results.

Merging paths The value returned by TYPEARMOR's recursive forward static analysis for a basic block B, which has n outgoing edges, provides us with a set of states S_i $(i = 1, 2, \ldots n)$. These states represent argument usage information for each path following edge i. Each state is represented by a vector composed by the state of each one of the six argument registers. TYPEARMOR combines these states into a *superstate* S that denotes the argument liveness for any path following B. For this purpose, we use a conservative policy that mandates that

the state for argument register c in S can only be R if the state for c is R for all states S_i $(i = 1, 2, \ldots n)$ following B. In other words, states W and C always supersede R, but both (W and C) are neutral with each other. After computing S, TYPEARMOR combines it with S_B, the state information for B. The merging policy here is slightly different in that states other than C in S_B always supersede states in S. This is because B is executed *before* any of its following basic blocks. For an actual example of how path merging works, please refer to Figure 3.3 on page 54.

Argument count Once the recursive analysis converges to a definite state for the entry basic block of a function, the argument count is set using the highest argument register that is marked as R. For instance, the System V ABI uses `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, as arguments registers. Therefore, if `r9` has a read-before-write state, we conclude that this particular function expects *at least* 6 arguments. If `r9` is W or C, then `r8` is examined, and so on.

Variadic functions Since variadic functions can take any number of arguments and thus may use all argument registers, variadic arguments may end up being passed in both CPU registers and memory (via the stack). To support easy manipulation of variadic arguments, modern compilers tend to move all the variadic arguments onto the stack in successive order upon entry of a variadic function. To make sure that the forward static analysis does not erroneously interpret the moving of argument registers to the stack as read-before-write operations (and conclude that this function expects more arguments than are defined), TYPEARMOR identifies variadic functions by means of pattern matching.

A function is labeled to contain n possible variadic arguments iff (1) a series of n argument registers, starting from the last argument register (`r9` for the System V ABI), are marked R, (2) these reads occur in the same basic block (and in the appropriate order), and (3) the arguments are written on the stack. If TYPEARMOR finds that a function contains n argument registers, it limits the maximum number of arguments for this function as computed by our forward analysis to $max - n$, where max is defined to be the maximum numbers of arguments that can be passed via registers (6 for the System V ABI). Figure 3.2 illustrates the operation of TYPEARMOR's variadic function detection mechanism using as an example the `ngx_snprintf` function.

We tested our variadic function detection mechanism against binaries compiled with both clang and gcc and found zero cases where a variadic function was mistakenly detected as a regular one. We did, however, observe a handful of

ngx_snprintf

```
u_char * ngx_cdecl
ngx_snprintf(u_char *buf, size_t max, const char *fmt, ...)
{
    u_char   *p;
    va_list   args;

    va_start(args, fmt);
    p = ngx_vslprintf(buf, buf + max, fmt, args);
    va_end(args);
    return p;
}
```

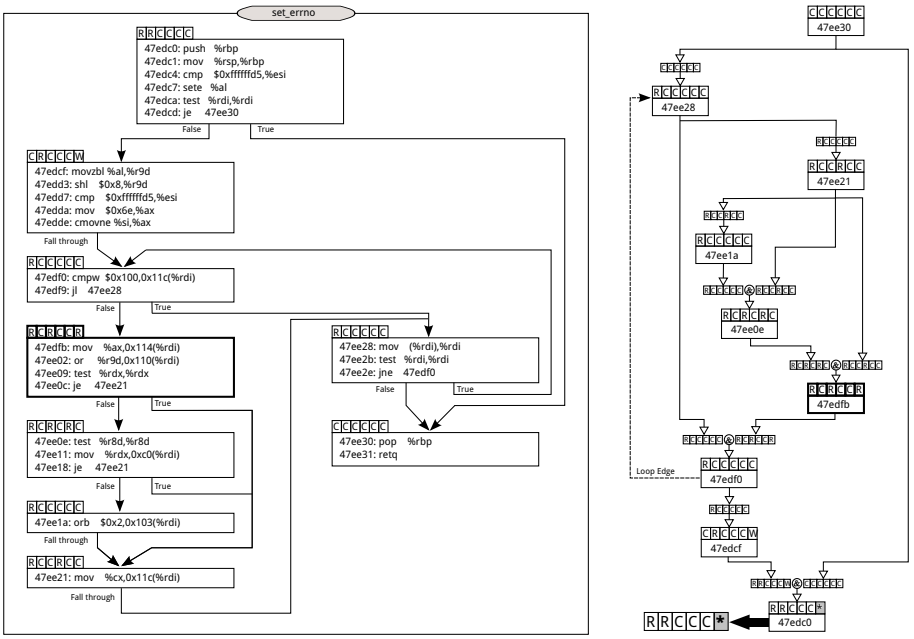
ngx_snprintf

```
40f310: push   %rbp                  |    40f34c: mov   %r8,-0xb0(%rbp)
40f311: mov    %rsp,%rbp             |    40f353: mov   %rcx,-0xb8(%rbp)
40f314: sub    $0xd0,%rsp            |    40f35a: lea   -0xd0(%rbp),%rax
40f31b: test   %al,%al               |    40f361: mov   %rax,-0x10(%rbp)
40f31d: je     40f345                |    40f365: lea   0x10(%rbp),%rax
40f31f: movaps %xmm0,-0xa0(%rbp)     |    40f369: mov   %rax,-0x18(%rbp)
40f326: movaps %xmm1,-0x90(%rbp)     |    40f36d: movl  $0x30,-0x1c(%rbp)
40f32d: movaps %xmm2,-0x80(%rbp)     |    40f374: movl  $0x18,-0x20(%rbp)
40f331: movaps %xmm3,-0x70(%rbp)     |    40f37b: add   %rdi,%rsi
40f335: movaps %xmm4,-0x60(%rbp)     |    40f37e: lea   -0x20(%rbp),%rcx
40f339: movaps %xmm5,-0x50(%rbp)     |    40f382: callq 40eb90
40f33d: movaps %xmm6,-0x40(%rbp)     |    40f387: add   $0xd0,%rsp
40f341: movaps %xmm7,-0x30(%rbp)     |    40f38e: pop   %rbp
40f345: mov   %r9,-0xa8(%rbp)        |    40f38f: retq
```

Figure 3.2. Variadic function detection searches for a basic block that performs read-before-write operations on a series of argument registers in consecutive order (either from lowest to highest in the case of gcc, or vice versa for clang) without other instructions in between. Observe in this particular example a group of instructions near the address `0x40f345`. The last three argument registers, namely `r9`, `r8`, and `rcx`, are moved (read-before-write) to the stack through instructions contained in a single basic block and in a specific order. Thus, this is a variadic function that uses its last three argument registers to hold variadic arguments.

cases where a function was wrongly detected as accepting a variadic number of arguments, leading to an underestimation of the number of arguments used (see also Section 3.8).

Conservativeness A key property of the analysis performed by TYPEARMOR at the callee is that it is conservative and therefore underestimation of the argument count is possible. Some interesting cases are: (i) instructions that perform a read and write on the same register (e.g., `xor %rdi,%rdi` or `neg %r9`), (ii) underestimated callees deriving from functions mistakenly detected as variadic, (iii) functions with *many* arguments (some of them passed through the stack), (iv) analyzed paths that contain further indirect calls, and (v) callbacks that do not



(a) CFG (b) Register merging

Figure 3.3. Callee analysis for `void set_errno(address_item *addrlist, int errno_value, uschar *msg, int rc, BOOL pass_message)` (in Exim). Observe how merging paths works. The basic block starting at `0x47edfb` (emphasized in bold) has state $S_B = (R, C, R, C, C, R)$, since `rdi`, `rdx`, and `r9` are read. There are two incoming states to this block, namely $S_1 = (R, C, R, C, R, C)$ and $S_2 = (R, C, C, R, C, C)$, which are combined to a superstate $S = (R, C, C, C, C, C)$ (C always supersedes). Finally, the superstate is combined with the block state, but this time R supersedes and hence the output state is $(R, C, C, C, C, C) \wedge (R, C, R, C, C, R) = (R, C, R, C, C, R)$. The final state of all analyzed blocks is $(R, R, C, C, C, *)$, where the $*$ denotes that C does not supersedes W or vice versa.

actually use *all* arguments. We stress that TYPEARMOR correctly handles case (i) and assigns the register either the state R or W depending on the used instruction (e.g., `xor %rdi,%rdi` is W and `neg %r9` is R). For (v), TYPEARMOR yields better results than a source-level analysis. As an example, consider a generic signal handler implementation, where the signal number is always passed—and, thus, at least one argument is expected to be passed to the callee—but not necessarily used by the handler, something TYPEARMOR can accurately infer.

Example of operation To illustrate how the analysis at the callee works, consider the `set_errno` function (taken from Exim) in Figure 3.3. The entry basic

block contains read operations on the first two argument registers (`rdi` and `esi`). At this point, the analysis cannot infer other possible arguments, but it can certainly proceed further. Based on the outcome of the conditional operation at address `0x47edcd` there are two available paths. In case the conditional check is *False*, the fall-through basic block at offset `0x47edcf` should be followed, otherwise, control should be transferred to address `0x47ee30`. The latter path simply returns and thus ends the function without any additional read-before-write operations. Since the analysis is conservative, this short path is sufficient to conclude that a *minimum* of two arguments are used by this function.

To illustrate TYPEARMOR's forward merging process, we include a complete merge graph for the `set_errno` function in Figure 3.3(b). Since merging is a backward process, the figure shows the CFG "up-side-down". As an example merging step, consider the basic block starting at `0x47edfb` which has $S_B = (R, C, R, C, C, R)$ (`rdi`, `rdx`, and `r9` are read). There are two incoming states to this block, namely $S_1 = (R, C, R, C, R, C)$ and $S_2 = (R, C, C, R, C, C)$, which are combined to a superstate $S = (R, C, C, C, C, C)$ (notice that C always supersedes). Finally, the superstate is combined with the block state, but this time R supersedes, and hence the output state is $(R, C, C, C, C, C) \wedge (R, C, R, C, C, R) = (R, C, R, C, C, R)$.

3.4.2 Callsite Analysis

TYPEARMOR iterates over each indirect callsite and performs a backward static analysis—a variant of classical *reaching definition* analysis [171]—to detect the prepared argument count at a particular callsite. The backward static analysis collects state information on all possible argument registers, but unlike our forward static analysis (Section 3.4.1), it only accounts for registers that are either set (S) or not (T, *trashed*). In particular, TYPEARMOR starts the analysis at the basic block that contains the indirect call, and iterates over preceding instructions for determining whether argument registers are S or T. If all argument registers are S, TYPEARMOR stops the analysis and assumes that the callsite uses the maximum number of arguments. If some arguments cannot be considered either S or T and the basic block has incoming edges, TYPEARMOR starts a recursive backward analysis.

Backward analysis Direct calls, returns, and other incoming edges are distinguished in the same fashion as in the callee analysis (see Section 3.4.1). For direct calls, the preceding basic block to analyze next is the basic block where the direct call originated. This means that if the backward analysis reached the entry block

of the function containing the inspected callsite, an inter-procedural backward analysis at all the callers of this function is initiated. Return edges during backward analysis indicate that the currently analyzed basic block has a predecessor that performs a function call. Thus, at this point, traversing further in this path is stopped and all remaining argument registers are marked as T: we assume that argument registers are always reset between two calls. This means that analysis is terminated and the state of this basic block is returned. Note that since indirect call targets cannot be resolved statically, there are no indirect call edges.

Merging paths Path merging for the backward analysis is relatively straightforward: for all collected states of the incoming basic blocks, T always supersedes S (arguments must be set on *all* paths). Similar to the forward analysis, once the recursive analysis is finished, the number of prepared arguments is set based on the states of the last write operations.

As an example, consider an indirect callsite cs that is reachable by two basic blocks b_1 and b_2, both of which are preceded by another indirect call instruction. If the backward analysis finds that b_1 writes to (sets) arguments register `rdi`, `rsi`, and `rdx` (the first three argument registers), while b_2 only sets `rdi`, TYPEARMOR concludes that cs prepares *at most* one argument.

Compiler optimizations TYPEARMOR's current implementation of backward static analysis may yield false conclusions (underestimation of number of prepared arguments) if the compiler deploys (inter-procedural) redundant argument register write elimination. Two examples of such optimization, which the compiler may perform at code generation time, are shown in Figure 3.4. The example in the left-hand side of Figure 3.4 shows how an inter-procedural write elimination pass may omit the second `mov $0x1,%rdi` instruction (depicted in red), since `rdi` has already been set to the same constant value in `foo`. The example on the right in Figure 3.4 shows a similar optimization instance, however eliminating writes across functions.

After analyzing the source code of two popular compilers (clang and gcc), we found no evidence of the presence of above optimization. Moreover, as we show in Section 3.8, a thorough comparison of clang-generated binaries against LLVM ground truth, across different optimization levels, did not reveal any underestimation of prepared arguments. These results confirm that the assumption that standard compilers always explicitly (re)set argument registers after a direct (and not only indirect) call is safe. For nonstandard or future compilers that may deploy inter-procedural write elimination optimizations, a possible solution is

```
1  foo(void)
2    mov $0x1, %rdi
3
4  bar(int arg1)
5    ...
6
7  main(void)
8    call foo
9    mov $0x1, %rdi
10   call bar
11
```

```
1  bar(int arg1)
2    ...
3
4  foo(int arg1)
5    ...
6
7  main(void)
8    mov $0x1, %rdi
9    call foo
10   mov $0x1, %rdi
11   call bar
```

Figure 3.4. Two variants of (inter-procedural) redundant argument register write elimination. In both code snippets, a compiler optimization pass may omit the second `mov $0x1,%rdi` instruction (highlighted in red). In the snippet on the left, `%rdi` was set to 1 in `foo` already and is never modified before the call to `bar`, making the second `mov` operation redundant. A similar scenario occurs in the snippet on the right-hand side: the compiler may conclude that `rdi` has been set before the call to `foo`, and was never modified before the second `mov` instruction.

to continue backward analysis from indirect callsite cs_2 until another indirect call cs_1 is found instead of a direct call: since the compiler does not know the target of cs_1 (or else it would have been a direct call), it shall reset all required arguments for cs_2, making our backward analysis between indirect calls safe by design.

Conservativeness As with the callee analysis, TYPEARMOR's callsite analysis should be conservative and therefore only allow for overestimation of the argument count. An interesting case to consider is how the analysis performs for indirect callsites inside *wrapper* functions. Such functions may not need to reset all argument registers, but simply 'pass them through' directly from its caller. However, if the wrapper has its address taken, and is only called through indirect functions, our backward analysis fails to find any incoming edges to the basic blocks and must give up. In order to be conservative, TYPEARMOR then decides that the callsite inside the wrapper prepares the maximum number of arguments.

To improve static analysis results for callsites, we complement TYPEARMOR to accept profiling data to improve its CFG. Consider above scenario of an indirect callsite inside a wrapper function. If a profile run finds an edge from another callsite to the wrapper, our static analysis can continue its backward analysis and possibly reduce the number of allowed arguments.

pr_response_flush

```
#define RESPONSE_WRITE_NUM_STR(strm, fmt, numeric, msg)              \
    pr_trace_msg(trace_channel, 1, (fmt), (numeric), (msg));        \
    if (resp_handler_cb)                                            \
        pr_netio_printf((strm), "%s", resp_handler_cb(resp_pool,(fmt),\
                        (numeric), (msg)));                         \
    else                                                            \
        pr_netio_printf((strm), (fmt), (numeric), (msg));

void pr_response_flush(pr_response_t **head) {
    ... ...
    RESPONSE_WRITE_NUM_STR(session.c->outstrm,  "%s %s\r\n",
                           last_numeric, resp->msg)
    ... ...
```

pr_response_flush

```
... ...                                  |    ... ...
426d51: callq 434960 <pr_trace_msg>      |    426e65: mov   0x5ea4c(%rip),%rdi
426d56: mov   0x5eb73(%rip),%r8          |    426e6c: mov   0x8(%rbx),%rdx
426d5d: test  %r8,%r8                    |    426e70: mov   0x10(%rbx),%rcx
426d60: mov   0x67c79(%rip),%rax         |    426e74: mov   $0x463a00,%esi
426d67: mov   0x38(%rax),%r15            |    426e79: xor   %eax,%eax
426d6b: jne   426e65                     |    426e7b: callq *%r8
                                         |    ... ...
```

Figure 3.6. Partial disassembly of the `pr_response_flush` function in ProFTPD that illustrates the working of our callsite analysis. The indirect call instruction at offset `0x426e7b` maps to the call to `resp_handler_cb`. TYPEARMOR's backward analysis finds that the basic block that ends with this indirect call writes to the first four arguments and thus continues analysis at incoming basic blocks. Only one such block exists and it performs a write operation on the fifth argument register (`test %r8,%r8`). Since the path that leads to this block ends with a call to `pr_trace_msg`, TYPEARMOR concludes that the indirect call `callq *%r8` prepares *at most* 5 arguments.

Example of operation Consider `pr_response_flush` from ProFTPD, which is depicted in Figure 3.6. Notice the indirect call located at offset `0x426e7b` which maps to `resp_handler_cb`; a variadic function that takes two fixed arguments. By analyzing the basic block, we infer that at least four argument registers are *live* (due to the 4 `mov` instructions). Since there is no information for the two additional argument registers (`r8` and `r9`), through recursive analysis, TYPEARMOR discovers all basic blocks directly pointing to `0x426e65`. For this particular scenario, one such block exists, starting at `0x426d56`. This block contains an instruction that moves a value into register `r8`, therefore this callsite is marked to hold a fifth argument. For inferring if `r9` is used as well, the analysis further proceeds and finds one basic block pointing to `0x426d56`. This block contains a return edge from `pr_trace_msg`, thus `r9` cannot be used as an argument register. As a result of the backward analysis, TYPEARMOR concludes that the callsite

`0x426e7b` prepares *at most* five arguments, one more than the actual number of prepared arguments (`strm`, `fmt`, `numeric` and `msg`).

3.4.3 Return Values

Adding information about return value usage improves the precision of TypeArmor's CFI implementation: if we find a callsite that expects a return value (a non-void callsite), it should never target a callee that does not prepare a return value (void functions). Extracting return usage information from callsites and callees is similar to the previously described callee and callsite analysis and is again conservative: a void callsite is allowed to target both void and non-void callees.

Non-void callsites The detection of non-void callsites (i.e., callsites that expect a return value), is done by searching for *read-before-write* operations on the the register that holds return values (`rax` for the System V ABI). In essence, we apply the forward analysis as used by our callee analysis, but now starting from the callsite, and only for `rax`. The difference is that we keep the analysis intraprocedural in order to remain conservative.

Void callees We detect void functions by applying the previously described backward analysis at the *exit points* of a function (exit points are basic blocks that end with a `ret` instruction). The backward analysis only searches for write operations on `rax` which may indicate a set return value.

In order not to break programs, our non-void callsite analysis is conservative and marks a callsite as void (allowing it to target both void and non-void functions) if no read-before-write on return registers is found (the callsite may pass the return value to a caller directly). Similarly, conservativeness at the callee results in an underestimation of the number of void functions: the compiler may use return registers as scratch registers, which we cannot detect by looking at disassembled instructions only. We describe the precision of our return value analysis in Section 3.8.

Note that if a particular ABI specifies that multiple registers may be used to hold return values (like the System V ABI allows callees to use the register pair `rax:rdx`), TypeArmor could be extended to perform a similar analysis on those as well.

3.5 Runtime Enforcement

In this section, we describe how TYPEARMOR uses the results from the static analysis, discussed in Section 3.4, to provide security guarantees at runtime. During application load time, TYPEARMOR's runtime component instruments the application's binary and loaded libraries to enforce our CFI and CFC policies. We achieve this by adding integrity and containment code at the forward edges and labels at function entry points. The runtime component can be split in three parts: (1) shadow code memory preparation, (2) CFI enforcement, and (3) CFC enforcement.

3.5.1 Shadow Code Memory Preparation

At every library load, this part of TYPEARMOR's runtime component allocates memory to store instrumented code, dubbed *shadow code* (as implemented by Dyninst [11]). The shadow code is essentially a copy of the original code that also contains the instrumentation of the callsites. Program execution is performed using the instrumented shadow code. Whenever we reach an indirect callsite during normal program execution, the instrumentation code at this location performs an integrity check between the type of the callsite and the type of the callee. If the types are compatible with each other, the callsite branches to the callee. Note that the branch target of the callsite still remains in the original code region. Therefore, we replace the beginning of each AT function in the original code with a jump instruction that jumps to the corresponding function in the shadow code region.

We perform the integrity check by retrieving and processing the function's label, located right before the function entry point in the original code region. Using this strategy, we do not have to ensure that our label does not overwrite code since the code that is executed is located in the shadow code region. Choosing the right label is not an easy task because we have to verify that this label does not occur at locations other than AT functions.

We tackle this problem with an approach similar to the pointer masking technique discussed by Wahbe et al. [139]. After moving all the code to the shadow code region, the unused locations in the original code region (i.e., all but AT function entry points) are filled with trap instructions[2]. Furthermore, during program execution, the integrity check that is performed at indirect callsites first masks the target address, so it can only point to the original code region, before continuing the execution. Using this strategy, indirect callsites can only point to

[2]Byte 0xCC is a trap instruction and disassembles to *int3*.

compatible AT functions.

Note that without the added instrumentation for *type* compatibility checks, the implementation with the shadow code region results in a coarse-grained CFI solution for forward edges, in which indirect callsites can target **all** AT functions without any restrictions.

3.5.2 CFI Enforcement

TYPEARMOR instruments binaries for enforcing that callsites can only target functions with a compatible *type*. This essentially means (1) a callsite with a higher number of prepared arguments can target all the functions that any callsite with a lower number of prepared arguments can also target, but not vice versa, and (2) a callsite that expects a return value can only target functions that return a value, whereas a callsite that does not expect a return value can target both functions that do and do not return a value. To implement these policies, TYPEARMOR has to instrument both, callsites and callees, based on the information collected through static analysis (see Section 3.4).

Callee instrumentation We label each AT function (i.e., prepend it with a magic number) similar to the original instrumentation scheme of Abadi et al. [2]. In the context of TYPEARMOR, there are seven possible labels (no arguments (0) to all arguments (6)), therefore, we use a 3-bit representation. In addition, we use one more bit to represent whether the function returns a value. We use 1 to encode *void* functions and 0 for functions that return a value. This is an important design decision for the callsite instrumentation, because callsites that expect a return value need to be handled in a special way (i.e., they can only target non-void functions) and this allows us to do it with just *one* extra instruction.

In practice, we use a 4-byte label and encode the function type using four bits of the label. For the return type, we use the least significant bit and, for the number of arguments, the adjacent three bits of the label. For example, we represent the bits of a *void* function that has four arguments according to the static analysis as `1001`.

To have a unique combination of four bytes that does not occur at any other code location, we choose `0xCCCCCC40` as a base label and use the four least significant bits to encode the function type. This form is suitable because all unused bytes are set to the trap instruction with which also the original code region is filled (see Section 3.5.1). The upper half of the least significant byte is set to four,

[3] In little endian, the label `0xCCCCCC40` would be represented as 0x40 0xCC 0xCC 0xCC in memory, which assembles to the code `REX INT3; INT3; INT3`.

because regardless of the value of the lower half of the byte, this byte assembles into the REX instruction prefix for the trap instruction[3]. Since REX has no effect when combined with the trap instruction, this label does not lead to valid targets for an attacker.

Callsite instrumentation At each callsite, TYPEARMOR's runtime component inserts a check to determine if the target is legal as per the CFI policy. It does so by retrieving the callee's label, decoding the type and checking if the result is compatible with the callsite. The instrumented check does the following:

1. Get the address of the target.
2. Mask the target address to force the callsite to point into the original code region.
3. Read the target's memory at target -4 to get the label.
4. Apply `xor` at the label with the value `0xCCCCCC40`. Note that we do not explicitly check if this part of the label was correct. If the label was incorrect, the check for the number of arguments (step 6) fails, since the result represents an unexpected value.
5. Only for callsites that expect a return value: make sure that the last bit is 0 (i.e., the target function does return a value) which is done by applying a right rotate by 1 bit on the label. Note that if the callsite targets a void function, the subsequent check fails, since the bit rotation results in a large value.
6. Using an unsigned comparison, check if the resulting value is below or equal to the (hardcoded) number of arguments the callsite is expecting. The range of possible values for callsites that expect a return value is $0 - 6$ and for callsites that do not expect a return value, the range is $0 - 13$. Note that the latter range also includes the return type bit.

As an example, consider the case where an indirect callsite which prepares four arguments and expects a return value tries to target a void-typed function that expects at least one argument. This function is assigned label `0xCCCCCC43`. At the callsite (after masking the target address, retrieving the label, and xoring the label with `0xCCCCCC40`) a right rotate of 1 bit is performed, because the callsite expects a return value. This results in the value `0x80000001`. Subsequently, the check for the number of arguments fails, since the resulting value is larger than 4, i.e. the prepared number of arguments at the callsite.

3.5.3 CFC Enforcement

We enforce CFC by scrambling unused registers at indirect callsites. Using this strategy, we essentially enforce a zero percent underestimation rate at the callee, at the cost of losing the ability to detect ongoing attacks, but instead silently crashing. Similarly, CFC scrambles the unused return register `rax` at return instructions of void functions so that we eliminate overestimation of non-void callsites.

As an example, consider an AT function f that accepts five arguments, but for which TYPEARMOR conservatively concludes that it accepts at least two arguments. Now, consider an indirect callsite cs for which TYPEARMOR assumes that it sets no more than three arguments. Without enforcing CFC, cs is allowed to target f. By enabling CFC, TYPEARMOR instruments cs in such a way that the last three argument registers (i.e., `rcx`, `r8`, and `r9`) are initialized with a random values at the callsite. The used random values are generated and inserted into the instrumentation code during load time. Observe that this does not change the fact that cs is allowed to target f. What it does enforce, however, is that if f enters a path that uses the 4th and 5th argument registers, the program is likely to crash as their values are no longer valid. Notice that we use random values (precomputed at load time) to initialize the argument registers and not fixed ones (such as zero). This is on purpose to avoid the risk of attacks based on malicious control flows that leverage a known state of the argument registers.

3.6 Mitigating Advanced Code-Reuse Attacks

In this section, we discuss how effective TYPEARMOR is in stopping advanced code-reuse attacks (CRAs). Table 3.1 presents a short summary of recently published CRAs that rely on control-flow diversion and how TYPEARMOR addresses them. Note that *all* publicly available exploits that are not pure data-only attacks (like Control Flow Bending [25]) are successfully mitigated.

In the following sections, we discuss advanced CRAs in more detail, COOP in particular. First, in Section 3.6.1, we analyze a set of server applications for COOP gadgets while TYPEARMOR is in place and explore if COOP is still possible. Next, in Section 3.6.2, we walk through practical COOP exploits for Internet Explorer, Firefox, and Chrome to show how TYPEARMOR stops these attacks. In Section 3.6.3, we discuss how TYPEARMOR stops Control Jujutsu exploits [49]. In Section 3.6.4, we discuss further possibilities of COOP exploitation. Finally, we conclude in Section 3.6.5 with a discussion on pure data-only attacks such as those presented by Control-Flow Bending [25].

Table 3.1. TYPEARMOR stops existing code-reuse exploits. Since TYPEARMOR specifically targets x86_64 binaries, the IE 32-bit COOP exploit is out of scope. Note that even without deploying CFC, TYPEARMOR stops all exploits.

Attack type	Exploit	Stopped?	Reason
COOP ML-G [120]	IE (32-bit)	✗	Out of scope
	IE 1 (64-bit)	✓(CFI)	Spurious arguments
	IE 2 (64-bit)	✓(CFI)	Spurious arguments
	Firefox	✓(CFI)	Spurious arguments
COOP ML-REC [39]	Chrome	✓(CFI)	Spurious arguments / illegal void target
Control Jujutsu [49]	httpd	✓(CFI)	Target function is not address-taken
	nginx	✓(CFI)	Illigal void target

3.6.1 Effectiveness Against COOP

Armed with the knowledge that COOP relies on unused argument registers to enable data flow between gadgets, we are interested in how many of those spurious arguments remain when TYPEARMOR is in place. We applied TYPEARMOR's static analysis on a large set of server application binaries and compared results against ground truth obtained by LLVM (more details in Section 3.7). Table 3.2 shows, for each server application, (1) the number of indirect call instructions (*cs*), (2) the number of callsites for which our analysis reports the *exact* number of prepared arguments as defined at the source level (*0*), and (3) the number of callsites for which we overestimate the number of prepared arguments by 1, 2, ...(*+N columns*).

From Table 3.2, we conclude that TYPEARMOR is able to determine the exact number of prepared arguments for the majority of callsites. While this is fairly promising already, the remaining callsites are potentially dangerous and could still be used as the initial COOP gadget by an attacker. To investigate this further, we operated a heuristic search for all possible main-loop (ML-G) and recursive (REC-G) gadgets for each of the server applications. We depict overestimation results for these possible gadgets in Table 3.3. Not completely unexpected, we only found reasonable gadgets in the C++ binaries—MySQL and Node.js.

For the main loop gadgets, TYPEARMOR accurately identified the argument count for 94% of the callsites in MySQL and for 95% in Node.js. Similarly, for the recursive gadgets, we identified the exact argument count in 86% (MySQL) and 96% (Node.js) of the cases. This means, however, that the remaining callsites may allow data to flow via the overestimated argument register(s) as identified by TYPEARMOR: these registers are not explicitly initialized with an argument value

Table 3.2. Accuracy of TYPEARMOR compared to the ground truth for different server applications. The numbers in the columns depict how far off the analysis is in terms of number of arguments (i.e., how many additional arguments are erroneously assumed by TYPEARMOR) for the callsites in the analyzed server applications.

		Overestimation					
Server	**#Callsites**	*0*	*+1*	*+2*	*+3*	*+4*	*+5*
exim	76	65	6	3	1	0	1
lighttpd	54	47	0	2	0	0	5
memcached	48	41	3	2	0	2	0
nginx	218	161	35	16	3	1	2
openssh	134	130	4	0	0	0	0
proftpd	85	68	10	3	2	2	0
pure-ftpd	10	8	1	0	1	0	0
vsftpd	4	2	2	0	0	0	0
postgresql	491	392	52	22	9	3	13
mysql	7,532	5,771	789	366	269	125	212
node.js	2,452	2,113	226	37	25	10	41

Table 3.3. Accuracy of TYPEARMOR compared to the ground truth for different server applications. The values are given in respect to callsites belonging to a specific type of gadget.

			Overestimation					
Gadget type	**Server**	**#Callsites**	*0*	*+1*	*+2*	*+3*	*+4*	*+5*
ML-G	mysql	173	163	3	1	1	0	5
	node.js	124	118	6	0	0	0	0
REC-G	mysql	278	240	14	11	2	4	7
	node.js	57	55	2	0	0	0	0

and may pass data set by one vfgadget to the next. As our automated gadget identification is not precise, we manually analyzed the remaining gadgets that might allow implicit data flow (data flow via spurious arguments). As noted in Section 3.2, the registers in question might still be unusable for data flow due to destructive updates in between the indirect calls.

For the main loop gadgets in MySQL, we found five callsites that were mistakenly reported to have an overestimated argument (caused by the fact that LLVM IR blocks may still get optimized or shuffled when bitcode is lowered to machine instructions), leaving only five callsites with true overestimation. For Node.js, overestimation affects six callsites. Manually analyzing the reported gadgets, however, revealed that no implicit data flow is possible for these call-

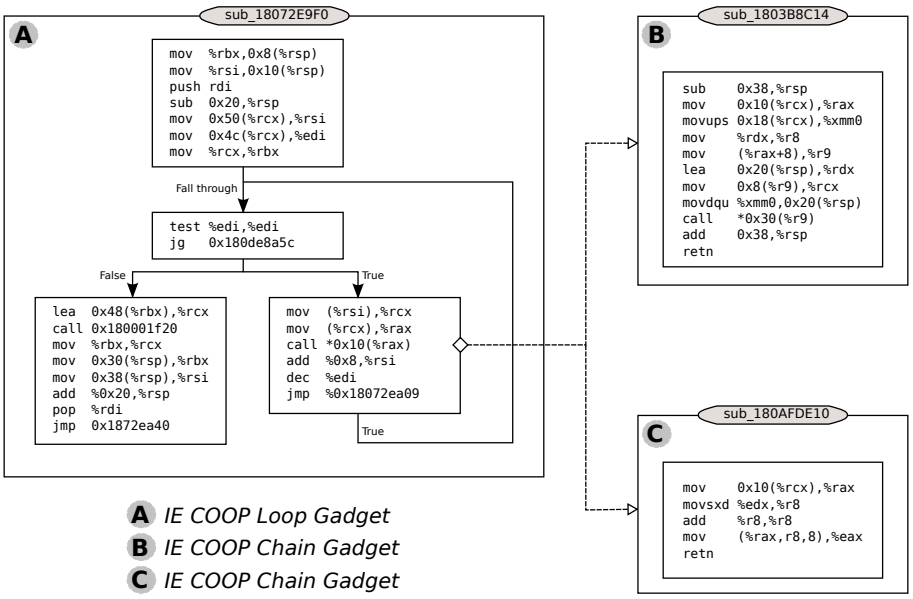


Figure 3.7. Gadgets used in COOP's 64-bit IE exploit.

sites. Results for recursive gadgets look equally promising: overestimation of prepared argument count occurred for 38 callsites in MySQL and two in Node.js. Manual analysis revealed that four gadgets were wrongly identified as REC-Gs, 13 could not set up an implicit data flow due to destructive updates, and for 23, CFC prevents data flow.

3.6.2 Stopping COOP Exploits in Practice

In the following sections, we analyze the published exploits for Internet Explorer (IE), Firefox [120], and Chrome [39] and show how TYPEARMOR stops these attacks.

Exploit on 64-bit IE The original COOP paper presents two exploits against 64-bit IE, both using the main loop gadget ML-G shown in Figure 3.7 (A). After initialization, the function `sub_18072E9F0` enters a loop and remains looping until `edi` reaches zero. The loop itself (1) loads a (counterfeit) object by setting the first argument (the `this` pointer): `mov (%rsi),rcx`, (2) prepares and calls a virtual function: `call *0x10(%rax)`, (3) increases `rsi` to point to the next virtual function: `add $0x8, %rsi`, and (4) decreases the loop counter: `dec edi`.

[4]Note that the Microsoft x64 calling convention is different from the System V AMD64 ABI: only the first four arguments are passed via registers, namely: `rcx`, `rdx`, `r8`, `r9`.

By controlling memory near `(%rsi)`, the exploit can chain virtual functions and launch the attack[4].

Let us now walk through TypeArmor's callsite analysis. It starts at the basic block that contains the indirect call instruction and concludes that only the first argument (`rcx`) is set in this block. Since there is no conclusive result for the remaining argument registers yet, it moves to the previous block which contains the loop condition (`test %edi,%edi`). As this block does not touch any argument register, it continues by searching for incoming edges to this block. The analysis finds two blocks: the entry block of the function, and the loop block that directly follows the indirect call instruction (ending with `jmp %0x18072ae09`). By following the second edge, we again see no write operations on argument registers, and the analysis must continue by searching for incoming edges to `add %0x8, %rsi`. It is at this moment that TypeArmor hits the `call *0x10(%rax)` instruction and can stop its analysis: the call instruction forces the compiler to reset any argument register if it is required by the program later on. So far, TypeArmor observed only one argument register to be set and concludes that this callsite sets *at most* one argument.

TypeArmor thus ensures that the indirect callsite in the loop gadget may target only those functions that accept zero or one argument. Looking at the chain of virtual functions, however, we find several *vfgadgets* that use a minimum of two arguments. One such function is `sub_1803B8C14`, illustrated in Figure 3.7 (B). The first two argument registers `rcx` (via `mov 0x10(%rcx),%rax`), and `rdx` (via `mov %rdx, %r8`) are read-before-write, and TypeArmor thus concludes that this is a function that expects *at least* two arguments. TypeArmor's many-to-many map now enforces that the indirect callsite in the loop gadget (that prepares *at most one argument*) is not allowed to call `sub_1803B8C14` (which expects *at least two arguments*). TypeArmor thus successfully stops the exploit.

The second COOP exploit against IE relies on the same ML-G, but deploys a different chain of virtual functions. Similar to the first exploit, it uses a virtual function that expects at least two arguments (shown in Figure 3.7 (C)—`mov 0x10(%rcx),%rax`) and `movsxd %edx,%r8`. Therefore, TypeArmor also stops this exploit.

Exploit on 64-bit Firefox We examined COOP's Firefox exploit and found that the ML-G used for the Firefox attack prepares only one argument (the `this` pointer, in `rdi`). Similar to what we observed for the IE exploits, this is correctly inferred by TypeArmor's callsite analysis. Moreover, the gadget chain relies on implicit data flows through argument registers and consists of functions

that expect *at least* two arguments, among others. This means that TYPEARMOR successfully stops the Firefox COOP exploit.

Exploit on Chrome In contrast to the ML-G gadgets used by the previous exploits, the improved COOP attack against Google Chrome alternates between two recursive gadgets (REC-G) to chain virtual functions. By analyzing the gadget chain, we find that three consecutive gadgets use `rsi` to pass data. Looking at the `SkComposeShader::contextSize()` REC-G, however, we find that TYPEARMOR identifies that its second indirect call (used to direct control flow to the second REC-G, `blink::XMLHttpRequest::AddEventListener()`), prepares only one argument. This means that TYPEARMOR's CFC enforcement scrambles data stored in `rsi` and thus breaks the exploit.

Additionally, the first indirect callsite in `SkComposeShader::contextSize()` is non-`void`, meaning that it can only call functions that set `rax`. One of the chained vfgadgets, `TtsControllerImpl::SetPlatformImpl()`, however, is of type `void` and never writes to `rax`. Thus, TYPEARMOR's CFI mechanism stops this attack as well.

3.6.3 Control Jujutsu

The two Control Jujutsu exploits [49] combine data and control-flow diversion attacks: the authors assume a (restricted) memory write to prepare a certain state, followed by overwriting a function pointer. The new function pointer stills targets a function entry, but one that can use the prepared state to give the attacker control over the program [49]. Inspecting the attacks with TYPEARMOR in mind, we can infer that we stop both attacks: (1) the attack against nginx diverts a non-void callsite in `ngx_output_chain` to target a void function `ngx_execute_proc`, which TYPEARMOR correctly detected as such; (2) the attack against Apache httpd diverts a callsite in `ap_run_dirwalk_stat` to invoke a target function that does not have its address taken (`piped_log_spawn`), which TYPEARMOR does not allow. Although the authors argue that in this scenario the attack can still succeed by calling `ap_open_piped_log_ex` instead (which wraps `piped_log_spawn`), this is not properly evaluated. By looking at the source code, it is likely that this extra level of indirection corrupts the attacker's prepared state.

3.6.4 COOP Extensions

While we demonstrated in Section 3.6.2 how TYPEARMOR stops all published COOP exploits, we now discuss the feasibility of advanced, previously unexplored techniques that could extend COOP.

Data Flow in COOP The original COOP paper presents multiple approaches to pass arguments between vfgadgets, distributed among three classes: (1) data flow using unused argument registers, (2) data flow using overlapping counterfeit object fields or global variables, and (3) data flow by relying on arguments actually passed to the callee. Note that the first class specifically targets x86_64, as it mostly uses registers to pass arguments to a function. We refer to this class as *implicit* data flow and for the remaining two as *explicit* data flow.

As the published COOP exploits demonstrate, *implicit* data flow is often key to successful exploitation: in many cases, ML-Gs and REC-Gs prepare only few arguments for the callsite, leaving the attacker with many registers she can use for data flow. Having more registers at her disposal, in turn, increases the probability of finding vfgadgets that implement useful functionality on these registers. One has to make sure, however, that the main-loop (or recursive) gadget does not overwrite said registers in between virtual function calls.

Explicit data flow, on the other hand, is characterized by enabling data flow using *actual* arguments to the vfgadget. Most notably, this also includes the first argument (which, for C++, depicts the object pointer). By overlapping multiple objects of different classes, two vfgadgets may operate on the same (overlapped) object field. This idea can be extended to other arguments as well, which is what COOP uses to enable data flow for their 32-bit IE exploit on Windows x86. In this approach, it uses an initial gadget that always passes the (same) field of the initial counterfeit object to the various vfgadgets. This field can then be used to pass arguments between gadgets and requires vfgadgets to dereference the corresponding argument and read from or write to it. Such field is defined as *argument field* [120].

Impact of TypeArmor on Data Flow TypeArmor effectively prevents implicit data flow. In Section 3.6.1, we show that static analysis is accurate enough to precisely determine the correct argument count for indirect callsites in many cases. Consequently, CFC scrambles all argument registers that are known to be unused. This prevents implicit data flow by design, both for ML-Gs and REC-Gs.

If TypeArmor fails to determine the exact argument count a callsite prepares, an attacker might be able to use this discrepancy to enable data flow. Note, however, that compared to the original COOP setting, she is still severely constrained. First, she does not have as many registers to choose from, which lowers the probability of finding vfgadgets with the desired semantics. Second, CFI is still in place, which significantly reduces the target set. In fact, our manual analysis shows that even for those cases, TypeArmor still makes implicit data flow impossible.

Looking at explicit data flows, we distinguish two cases. First, data flows using overlapping object fields, for which we refer to the original COOP paper: it already concludes that these scenarios are difficult to apply in practice. The second case enables a different class of COOP data-flow semantics, which relies on the presence of an argument field. As with the first scenario, however, this is hard to realize in practice since not passing the particular argument may heavily interfere with the program's semantics.

Advanced argument-passing techniques can be tackled by source-level CFI solutions: they have access to type information of the callsite, and can thus enforce a match to types of the callee. In particular, such information can reduce the number of gadgets applicable for data flow via argument fields (object fields that are passed as parameter to a vfgadget by the ML-G). If an analysis determines an argument field to be a pointer, the ML-G's callsite can only target vfgadgets that expect a pointer for the corresponding argument and vice versa. We anticipate that this argument type distinction is also possible at the binary level and consider it as something to explore in future work.

Although we confirm that advanced COOP exploitation is still possible in theory, we stress that a significant reduction of the attack surface at the binary level is possible. In fact, with TYPEARMOR in place, only the really elaborate, but inherently constrained, options for argument passing survive for building working COOP exploits.

3.6.5 Pure Data-Only Attacks

The Control-Flow Bending (CFB) paper evaluates the general effectiveness of ideal CFI solutions and evidences their limitations against sophisticated CFG-aware attacks [25]. The authors show that CFB attacks against CFI solutions that are complemented by a shadow stack are more difficult, but sometimes still possible.

As any other CFI solution, TYPEARMOR cannot stop pure data-only attacks. However, attacks that use an arbitrary memory write to overwrite a function pointer can still potentially be stopped: if the attacker overwrites a pointer to point to a function that expects more arguments than the original target, or if the new target assumes that certain callsite arguments that have been scrambled by CFC contain a specific value, the attack will be stopped.

Through personal communication, the CFB authors shared their exploit notes for the presented Apache and Wireshark attacks; two attacks that work even in the presence of a runtime shadow stack and ultimately overwrite a function pointer at some point during the exploit. After analyzing the exploits in depth,

we conclude that these truly are pure data-only attacks, and cannot be stopped by TYPEARMOR. It is worth mentioning that even source-level CFI solutions cannot stop these two attacks.

3.7 Performance

TYPEARMOR is implemented on Linux for x86_64. The callee and callsite analysis component, outlined in Section 3.4, is implemented in 5,532 lines of C++ code and depends on the Dyninst v8.2.1 binary analysis framework to disassemble machine code [11]. The runtime component, outlined in Section 3.5, also relies on Dyninst to perform binary instrumentation and consists of 743 lines of code. The prototype supports generic 64-bit ELF binaries as long as they do not emit self-modifying code.

The evaluation testbed is a system equipped with an Intel i5-2400 CPU at 3.10GHz and 8GB of RAM. We ran our tests on Ubuntu 14.04 x86_64 running kernel 3.13. We focus our performance evaluation on popular Linux server applications, given that (1) they are widely adopted in the research community for evaluation purposes, (2) they are popular exploitation targets, and (3) they naturally contain a relevant number of indirect callsites that can greatly benefit from the protection offered by TYPEARMOR. Specifically, we evaluated TYPEARMOR with three FTP servers (namely, vsftpd v1.1.0, ProFTPD v1.3.3, and Pure-FTPd v1.0.36), two web servers (nginx v0.8.54 and lighttpd v1.4.28), an SSH server (the OpenSSH Daemon v3.5), an email server (Exim v4.69), two SQL servers (MySQL v5.1.65 and PostgreSQL v9.0.10), a general-purpose distributed memory caching system (Memcached v1.4.20), and a cross-platform runtime environment for server-side web applications (Node.js 0.12.5, statically compiled with Google's v8 JavaScript engine). Finally, we considered all C and C++ SPEC CPU2006 benchmarks for completeness and direct comparison with prior work.

To benchmark the web servers and Node.js (which we configure to serve a JavaScript page that mimics default web-server behavior), we configured the Apache benchmark [239] to issue 250,000 requests with 10 concurrent connections and 10 requests per connection. To benchmark the FTP servers, we configured the pyftpbench benchmark [242] to open 100 connections and request 100 1 KB-sized files per connection. To benchmark Memcached, we used the memslap benchmark [238]. To benchmark the SQL servers, we configured the Sysbench OLTP benchmark [240] to issue 10,000 transactions using a read-write workload. Finally, to benchmark OpenSSH and Exim, we used the OpenSSH test suite [243] and a homegrown script which repeatedly launches the sendemail program [245],

Table 3.4. Runtime performance overhead for server applications. The *CFI* column depicts the overhead of TYPEARMOR's CFI implementation (checking callee labels before each indirect call instruction). *+CFC* depicts the slowdown of TYPEARMOR's complete configuration.

		Overhead	
Server	**#Indirect calls per second**	*CFI*	*+CFC*
exim	4,574	1.068	1.067
lighttpd	1,425,099	1.116	1.174
memcached	72,519	1.014	1.017
nginx	5,084,715	1.132	1.155
openssh	78	1.021	1.013
proftpd	542,443	1.007	1.002
pure-ftpd	17	1.020	1.013
vsftpd	24,024	1.025	1.051
postgresql	18,024,485	1.160	1.205
mysql	19,693,937	1.239	1.222
node.js	1,965,955	1.061	1.055

respectively. We configured all applications and benchmarks with their default settings. We ran all the experiments 11 times (checking that the CPUs were fully loaded throughout the tests) and report the median (with marginal standard deviation observed across runs).

To evaluate the impact of TYPEARMOR's instrumentation on runtime performance, we measured the time to complete the execution of the benchmarks and compared against the baseline. The baseline refers to the original version of the benchmark with no binary instrumentation applied. Table 3.4 details the normalized runtime for two configurations. The *CFI* configuration refers to TYPEARMOR solely enforcing forward-edge CFI as outlined in Section 3.5.2. As shown in the table, this configuration introduces a noticeable performance impact (7.6% on average, geometric mean), owing to about half of the applications executing millions of indirect callsites per second. The overhead is comparable to TYPEARMOR's complete (and default) configuration (*CFI+CFC*), which accounts for TYPEARMOR also clearing unused argument registers at each callsite (8.6% on average, geometric mean). On applications that execute less than a million indirect callsites per second, TYPEARMOR has a marginal performance impact.

To obtain standard and comparable performance results across TYPEARMOR's configurations, we measured the time to complete the SPEC CPU2006 benchmarks and compared it against the baseline. Again, the baseline refers to the original version of the SPEC2006 benchmarks with no binary instrumentation

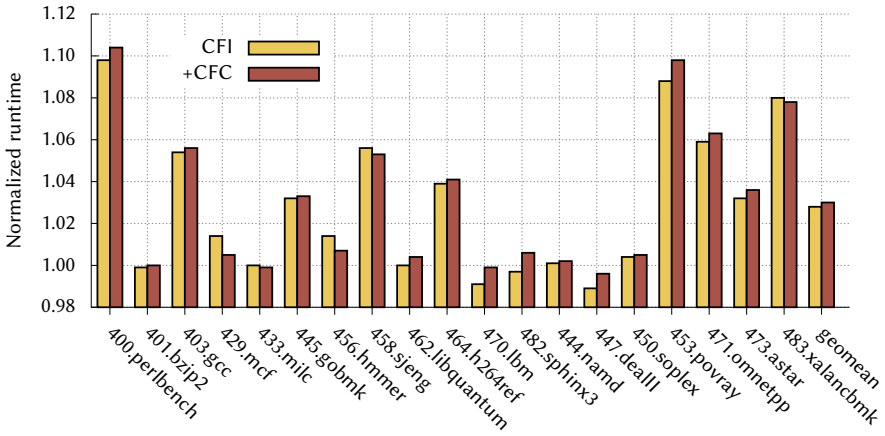


Figure 3.8. Benchmark runtime normalized against the baseline for SPEC CPU2006.

applied. We present results in Figure 3.8 and confirm the general behavior observed for the server applications, with an average performance overhead of only 2.4% for TYPEARMOR in a CFI-only configuration and of 2.5% for TYPEARMOR in the default (CFI+CFC) configuration (geometric mean).

Overall, TYPEARMOR imposes a relatively low runtime performance impact on all the test programs considered. This confirms that our lightweight instrumentation is successful in producing a runtime overhead that is comparable to, or even faster than existing binary rewriting-based CFI solutions [157].

3.8 Security Analysis

Common evaluation metrics used to assess the effectiveness of defense mechanisms have been questioned by the community [25]. In this paper, we acknowledge that additional research is required for converging on a more efficient security evaluation system. However, for completeness and comparability with similar works, in this section we evaluate TYPEARMOR using security metrics proposed by other systems.

Table 3.5 presents accuracy results for (1) callsite and (2) callee analysis. To validate TYPEARMOR's static analysis results—ensuring no underestimation occurs at the callsite and no overestimation is observed at the callee—and to compute the accuracy in detecting return usage and exact number of prepared/consumed arguments, we compared TYPEARMOR's results against the ground truth generated from source code. For this purpose, we (1) relied on the LLVM frame-

Table 3.5. Static analysis results for recovering callsite signatures and callee prototypes. The **Callsites** and **Callees** groups report statistics on (1) how many callsites/callees were found, (2) in how many cases our static analysis correctly identified the number of set/used arguments, and (3) the number of correctly detected non-void callsites and void callees. For the latter, the percentage displayed inside parentheses shows the correctness ratio. For example, for lighttpd 84% of the callsites that expect a return value and 20% of the void functions were correctly identified as such.

	Callsites			**Callees**		
Server	#	*args*	*non-void (%)*	#	*args*	*void (%)*
exim	76	65	44 (67.69)	615	495	32 (17.98)
lighttpd	54	47	21 (84.00)	353	311	19 (20.00)
memcached	48	41	15 (100.00)	236	210	11 (7.91)
nginx	218	161	155 (90.64)	1,111	869	57 (22.62)
openssh	134	130	67 (100.00)	715	625	48 (13.19)
proftpd	85	68	62 (93.94)	1,188	1,045	69 (26.74)
pure-ftpd	10	8	3 (50.00)	201	169	0 (0.00)
vsftpd	4	2	1 (100.00)	445	371	29 (12.03)
postgresql	491	392	328 (87.23)	9,312	8,054	494 (15.13)
mysql	7,532	5,771	4,783 (70.97)	9,961	6,977	1,277 (36.60)
node.js	2,452	2,113	1,199 (91.39)	34,703	28,698	3,444 (22.26)

work to compile source code into an intermediate representation (LLVM IR) at different optimization levels, (2) extracted ground truth numbers (number of arguments prepared for each indirect callsite, number of arguments consumed for each function, and the list of callsites/callees that expect or set a return value), and (3) lowered LLVM bitcode to machine code (using the same optimization levels) on which we ran TYPEARMOR's static analysis. Table 3.5 reports results for `-O2`, but we observed similar results at other optimization levels. For this experiment, we excluded libraries to ensure a fair comparison across server applications. In addition, we included callee analysis results (second group in Table 3.5) for *all* functions in the program.

Table 3.5 shows that the static analysis results are very accurate in identifying the exact number of used arguments (ranging from 50% to 97% for callsites and from 70% to 89% for callees). The forward static analysis results are slightly better than those obtained with the backward static analysis, given that the stop condition for the callee analysis is stronger than the one used for callsites. Nonetheless, results are encouraging, given that TYPEARMOR can, overall, compute the exact number of source-level arguments in more than 75% of the cases, while operating entirely at the binary level and in a conservative fashion. Similarly, with a success rate of 84% on average, results for detecting non-void callsites are also

Table 3.6. Median number of targets for an indirect callsite across different CFI policies, both for **Binary**-level policies (address-taken and TYPEARMOR) and for **Source**-based defenses (address-taken and IFCC [130]).

	Binary			**Source**	
Server	*AT*	*CFI*	*+CFC*	*AT*	*IFCC*
exim	615	41	40	67	3
lighttpd	353	50	47	59	6
memcached	236	14	14	14	1
nginx	1,111	352	254	518	25
openssh	715	32	6	90	4
proftpd	1,188	390	376	402	3
pure-ftpd	201	6	4	14	0
vsftpd	445	12	12	15	1
postgresql	9,312	2,357	2,304	2,509	12
mysql	9,961	4,158	3,698	6,097	150
node.js	34,703	4,804	4,714	7,527	341

accurate. On the other hand, detecting void functions is much harder: we detect less than 20% of the actual void callees. This is caused by `rax` being used as scratch register in many cases, resulting in an underestimation of the number of void functions.

The **Binary** group in Table 3.6 displays the median number of legal indirect callsite targets as enforced by existing (binary-level) address-taken-based solutions and TYPEARMOR's policies. It reflects the static analysis results on the number of legal targets, measuring the strength of CFI and CFC invariants. The *AT* column reports results for existing state-of-the-art binary-level CFI solutions that allow indirect callsites to target any address-taken function [105, 157]. This results in a CFI solution allowing indirect callsites to target all valid function entry points. The *CFI* and *+CFC* columns report results for TYPEARMOR deployed in a CFI-only configuration and in a full CFI+CFC configuration, respectively.

Table 3.6 shows that on average, TYPEARMOR is capable of reducing the number of legal targets by roughly two orders of magnitude (91% reduction on average for CFI+CFC) compared to the conservative binary-level address-taken strategy (*AT*) adopted in prior solutions. The results also demonstrate the effectiveness of CFC, which can further reduce the targets allowed by CFI alone (110 vs. 141 targets on average).

The **Source** group in Table 3.6 allows us to compare TYPEARMOR with source-level techniques to assess the strength of the of constraints imposed on indirect callsites. We compared TYPEARMOR with an LLVM-based tool for address-taken

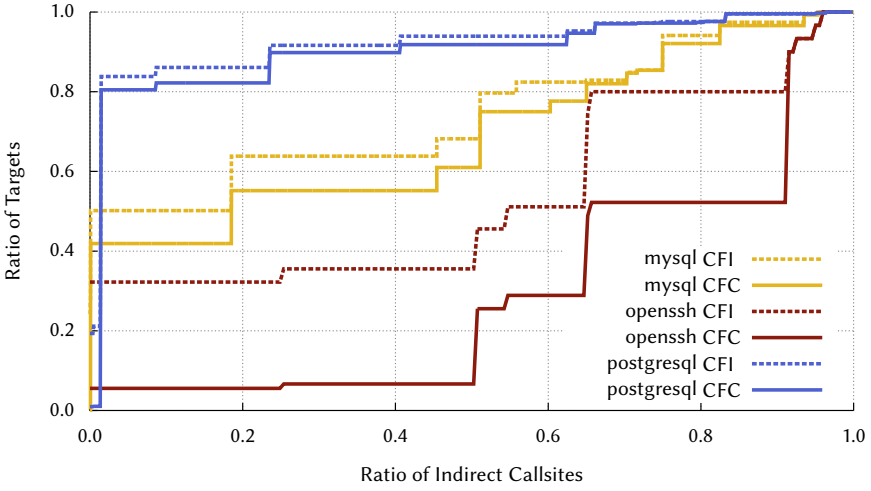


Figure 3.9. CDF of legal indirect callsite targets enforced by TYPEARMOR's CFI and CFC policies.

(AT) analysis, and state-of-the-art source-level CFI defenses, i.e., IFCC [130]. As expected, IFCC significantly reduces the available targets of indirect callsites compared to TYPEARMOR. However, note that for certain programs (e.g., OpenSSH) TYPEARMOR performs equally well, although applied at the binary level, and, in *all* cases, TYPEARMOR yields the same or better results than source-based address-taken analysis.

For a more accurate view of the invariants enforced by TYPEARMOR, we report a CDF in Figure 3.9 of legal callsite targets. For clarity, we limit the CDF to CFI and CFC with applications that (1) yield minimal target reduction compared to source-level AT results (PostgreSQL, blue), (2) contain many indirect callsites and AT functions (MySQL, yellow), and (3) yield high reduction (OpenSSH, red).

Based on the CDF of Figure 3.9, we observe that CFC results for each program follow the same trend as CFI. This is inherent to the deployment of the callsite-oriented invariants, with the number of indirect callsites being constant. We observe that results for PostgreSQL, due to the unusual internal structure of the program and the weaker quality of the resulting invariants, are more conservative than other cases, with over 90% of the indirect callsites allowing 80% or more targets. This difference is due to the distribution of the argument count for AT functions: for PostgreSQL, over 85% of the AT functions are detected as consuming at least 0 or 1 argument. This means that as soon as TYPEARMOR's backward analysis finds that a callsite prepares an additional argument, it must

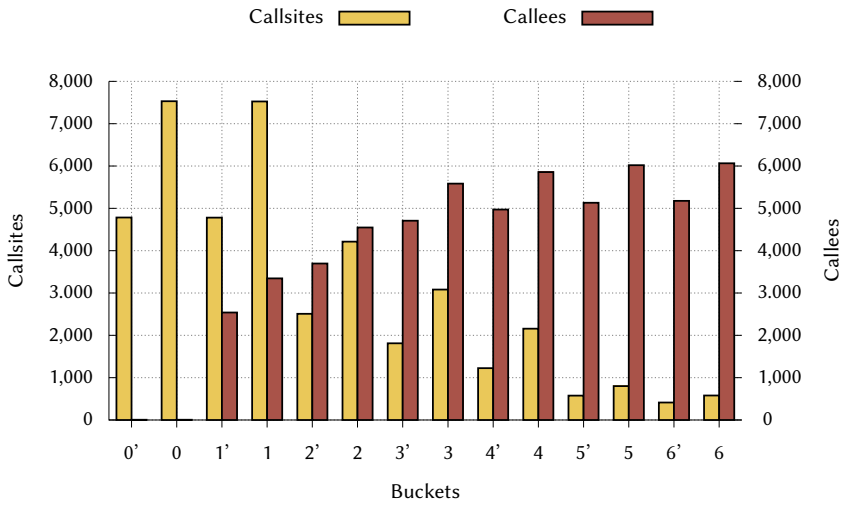


Figure 3.10. Distribution of CFC buckets for MySQL. A tick (’) denotes a bucket containing callsites/callees that expect and set return values.

allow all those 85% as a possible target. To make this more concrete, TypeArmor concludes that for OpenSSH, only 26 (out of 90) AT functions accept 0 or 1 argument. Encouragingly, other programs exhibit a regular internal structure, resulting in much stronger type-based invariants. For example, we find that results for OpenSSH are impressive: for 90% of all indirect callsites, CFC still yields an almost 50% reduction of the legal targets. Moreover, for 35% of the callsites, TypeArmor allows only 7% of all AT functions as valid target.

To further analyze the distribution of possible callees among callsites, Figure 3.10 depicts a histogram of the different buckets that are enforced by TypeArmor’s CFC policy. For each bucket, it shows the number of callees (red) and callsites (yellow) that fall into it. Without return-use information, the System V ABI enables six buckets: callees that take at least 0 arguments, 0 to 1 arguments, 0, 1, or 2 arguments, …, callees that take any number of arguments. By adding return use information (denoted with a tick ’ in Figure 3.10), the number of buckets is doubled. As an example, consider bucket 3. This bucket contains the callees that expect 0, 1, 2, or 3 arguments, but not those that expect at least 4 arguments or more. On the other hand, it contains callsites that prepare *at most* 6, 5, 4, or 3, arguments, but not 2 or less. Another example is bucket 3’, which consists of the same set of only callsites and callees that set and expect a return value.

Figure 3.10 illustrates the intuitive effectiveness of TypeArmor: there is a limited set of callsites (around 500 for MySQL) that are allowed to target any AT

function (over 6000), while there are many callsites (7500) that can target only a limited amount of callees (less than 4000). Note that since MySQL is a C++ program, and thus `rdi` is often used to hold the `this` pointer, we see almost zero callees in the first two buckets.

Overall, we conclude that TYPEARMOR's CFI and CFC invariants yield a significant reduction in the number of legal targets at indirect callsites.

3.9 Related Work

Ever since the original CFI proposal by Abadi et al. [2] and the rise of advanced code-reuse attacks [25, 26, 43, 57, 58, 120, 124], there have been several CFI techniques proposed in the literature, targeting both *source* and *binary* compatibility and with different strength of invariants. In this section, we briefly review state-of-the-art CFI solutions *vis-à-vis* TYPEARMOR.

Binary-level CFI. Realizing binary-level CFI in practice is hard, since computing the CFG of a program is an undecidable problem and instrumentation usually incurs overheads. Therefore, there has been research for CFI approximations, realized through coarse-grained CFI [154, 157]. However, these approximations have been demonstrated vulnerable [57]. Lockdown [105] and CFCI [158] attempt to deploy fine-grained CFI schemes, while VTint [153], vfGuard [110], and T-VIP [51] focus on protecting just VTables at the binary level. However, it was recently demonstrated that even fine-grained schemes can be bypassed [25, 49] and VTable protections without access to C++ semantics are infeasible [120]. PathArmor shows how recent hardware features can be used to deploy context-sensitive CFI with low overhead [131], but, in absence of forward-edge context-sensitive invariants, COOP attacks are still possible. In contrast to these solutions, TYPEARMOR enforces binary-level invariants based on the number of function arguments, targeting exclusive protection against all these advanced exploitation techniques that bypass fine-grained CFI schemes and VTable protections, at the binary level.

Source-level CFI. Source-level solutions, such as IFCC/VTV [130], SAFEDISPATCH [66], CPI [83], and ShrinkWrap [61] can realize CFI with increased accuracy. In this respect, TYPEARMOR is an attempt to approximate source-level accuracy at the binary level. Although TYPEARMOR is less accurate than such source-level solutions, we argue in this paper that, in the context of sophisticated attacks such as advanced COOP extensions, it is questionable whether additional accuracy that is provided by source-level solutions is required. For most advanced techniques, such as all publicly released COOP exploits, invariants as enforced by TYPEARMOR may be sufficient and effective in practice.

3.10 Conclusion

In this paper, we presented TYPEARMOR, a new detection and containment solution against advanced code-reuse attacks. TYPEARMOR relies on binary-level static analysis to derive both target-oriented and callsite-oriented control-flow invariants and efficiently apply security policies at runtime. In particular, TYPEARMOR relies on target-oriented invariants to enumerate legal callsite targets and *detect* attacks that transfer control to illegal targets (akin to traditional CFI, but with much stronger binary-level invariants). In addition, TYPEARMOR relies on callsite-oriented invariants to invalidate illegal function arguments at each callsite and *contain* attacks that rely on type-unsafe function argument reuse, using a protection technique dubbed Control-Flow Containment (CFC). CFC further improves the quality of our target-oriented invariants, resulting in the strictest binary-level CFI solution to date.

The COOP papers questions whether it is even *possible* to mitigate sophisticated forward-edge attacks using binary-level CFI solutions. TYPEARMOR contrasts these claims with concrete evidence that constructing a strict binary-level CFI solution to counter the most advanced code-reuse attacks in the literature is *possible* and realistic in practice. To substantiate our claims, we demonstrated that TYPEARMOR stops all published COOP exploits.

Shared Authorship

I share first authorship on TYPEARMOR with Enes Göktaş. Enes is the main author of the runtime component, while my focus was on the static analysis implementation and overall evaluation.

4 VPS: Excavate High-Level C++ Constructs from Low-Level Binaries to Protect Dynamic Dispatching

Low-level languages such as C++ are prone to both temporal and spatial memory corruption vulnerabilities. Features like polymorphism that make C++ suitable for writing complex software increase the binary-level attack surface because they rely on function pointers in a so called *virtual function table (vtable)* that attackers can potentially hijack. In practice, *vtable hijacking* is one of the most important attack techniques.

In this chapter, we present *VTable Pointer Separation* (VPS), a binary-level defense against vtable hijacking in C++ applications. VPS achieves accurate protection by restricting virtual callsites to validly created objects. More specifically, VPS ensures that virtual callsites can only use objects created at valid object construction sites. VPS prevents wrongly identified virtual callsites from breaking the binary, an issue most previous work do not consider. We evaluate the prototype implementation of VPS on a diverse set of large applications (MySQL server, Node.js, SPEC CPU2017 and CPU2006), showing that our approach protects on average 97.7% of all virtual callsites in SPEC CPU2006 and 97.4% in SPEC CPU2017 (all C++ benchmarks), with a moderate performance overhead of 9% and 11% geometric mean, respectively. Furthermore, our evaluation reveals 86 false negatives in *VTV*, a source-based defense part of GCC.

4.1 Introduction

Despite over three decades of research on software security, memory corruption vulnerabilities are still a major threat to the integrity of today's software [225]. Software implemented in low-level languages such as C and C++ is particularly vulnerable to such attacks, and the sophistication of attacks is rising (e.g., [25, 34, 57, 120]). In practice, especially C++ is often the programming language of choice for complex software because it allows developers to structure software by encapsulating data and functionality in *classes*, simplifying the development process. Unfortunately, the binary-level implementations of C++ features like polymorphism and (multiple) inheritance are often vulnerable to control flow hijacking attacks, most notably *vtable hijacking*.

Vtable hijacking abuses common binary-level implementations of C++ virtual methods where every object with virtual methods contains a pointer to a *virtual function table* (*vtable*) that stores the addresses of all the class's virtual functions. To call a virtual function, the compiler inserts an indirect call through the corresponding vtable entry (a *virtual callsite*). Using temporal or spatial memory corruption vulnerabilities such as arbitrary write primitives or use-after-free bugs, attackers can overwrite the vtable pointer so that subsequent virtual calls use addresses in an attacker-controlled alternative vtable, thereby hijacking the control flow. In practice, vtable hijacking is a common exploitation technique used in exploits that target complex applications written in C++ such as web browser and server applications [237].

Control-Flow Integrity (CFI) solutions [2, 16, 97, 107, 130] protect indirect calls by verifying that control flow is consistent with a Control Flow Graph (CFG) derived through static analysis. However, most generic CFI solutions do not take C++ semantics into account and leave the attacker with enough wiggle room to build an exploit [57, 120]. Consequently, approaches that specifically protect virtual callsites in C++ applications were developed. If source code is available, compiler-level defenses are preferable because they benefit from the rich class hierarchy information available at the source-level [20, 23, 130, 152]. In contrast, binary-level defenses that can protect already compiled, proprietary binaries [46, 51, 103, 110, 153] cannot take advantage of source-level information. As a result, the accuracy of their (automated) analysis is of uttermost importance since not only their security guarantees hinges on it, but also if their instrumentation is prone to breaking the application.

In this chapter, we present *VTable Pointer Separation* (VPS), a binary-level defense against vtable hijacking attacks. Unlike previous binary-only approaches

that restrict the set of vtables permitted for each virtual callsite, we ensure the vtable pointer was not modified since the object was created. Moreover, we observe that the vtable pointer in a legitimate live object never changes, allowing VPS to avoid virtual pointer hijacking by enforcing this rule. While CFIXX [23] performs enforcement in a similar way, it only works on source code and lacks the binary analysis that VPS introduces. Our approach is particularly suitable for binaries because, unlike other binary-level solutions, we avoid the inherent inaccuracy in binary-level CFG and class hierarchy reconstruction. Because VPS allows only the initial virtual pointer(s) of the object, we reduce the attack surface even compared to hypothetical implementations of prior approaches that statically find the set of possible vcall targets with perfect accuracy. In addition, our method prevents use-after-free attacks.

Given that binary-level static analysis is challenging and unsound in practice, we also propose a way to deal with potential false positive results in the virtual callsite identification. We carefully over-approximate the set of callsites and implement an efficient slow path to handle possible false positives at runtime, while allowing fast checks for previously verified callsites. This approach allows us to prevent false positives from breaking the application due to the security checks at virtual callsites like they do in existing work [46, 51, 110, 153]. Additionally, we note that, while existing work [68, 70, 71, 103] only considers *directly* referenced vtables, the compiler can also generate code that references vtables *indirectly*, i.e., through the Global Offset Table (GOT). As a result, VPS is able to find all code locations that instantiate objects by writing the vtable into it. This is an important observation since misses open our approach to breaking the application as well.

We built a prototype of VPS and show that our analysis is sufficiently precise to handle complex, real-world C++ applications such as MySQL server, Node.js, and all C++ applications contained in the SPEC CPU2006 and CPU2017 benchmark suites. Our results indicate that we can on average correctly identify 97.7% and 97.4% of virtual callsites in SPEC CPU2006 and SPEC CPU2017, with a high precision of 95.6% and 91.1%, respectively. Compared to previous work on binary-level defenses against vtable hijacking such as *Marx* [103], our analysis identifies 5.9% more virtual callsites in SPEC CPU2006 and 26.5% in SPEC CPU2017 and thus offers improved protection. Interestingly, our evaluation revealed 86 virtual callsites that are not protected by the source code based approach *VTV* which is part of GCC [130]. A further investigation with the help of the *VTV* maintainer shows that these misses are due to a conceptual problem in *VTV* which requires non-trivial engineering to fix. VPS induces a moderate performance overhead:

we measured a geometric mean overhead of 9% for all C++ applications in SPEC CPU2006 and 11% for SPEC CPU2017.

Contributions. In summary, we provide the following contributions in this chapter:

- We present a binary-only defense against vtable hijacking attacks called VPS that sidesteps the imprecision problems of other work on this topic. The key insight is that vtable pointers should never change once an object is created. We provide a detailed comparison against existing binary-only approaches in Section 4.3.1.
- We develop an instrumentation approach that is capable of handling false positive identification of C++ indirect callsites which can break the application and are ignored by most existing work. In addition, we investigate and handle indirect vtable references, which were also not considered in prior work.
- Our evaluation shows that we can on average correctly identify 97.7% and 97.4% of the existing virtual callsites in SPEC CPU2006 and CPU2017, with a high precision of 95.6% and 91.1%, respectively. In addition, our evaluation uncovered a conceptual problem in *VTV*, a well-known source-level defense that is part of GCC, leading to false negatives.

4.2 C++ at the Binary Level

This section provides background on C++ internals needed to understand how VPS handles C++ binaries. We focus on how high-level C++ constructs translate to the binary level. For a more detailed overview of high-level C++ concepts, we refer to the corresponding literature [169].

4.2.1 Virtual Function Tables

C++ supports object-oriented programming with *polymorphism*. A class can inherit functions and fields from another class. The class that inherits is called the *derived* class and the class from which it inherits is the *base* class. In addition to *single inheritance* (one class inherits from one other class), C++ also allows *multiple inheritance*, where a derived class has multiple base classes.

A base class can declare a function as *virtual*, which allows derived classes to override it with their own implementations. Programmers may choose not to implement some functions in a base class: *pure virtual* functions. Classes con-

taining such functions are *abstract* classes and cannot be instantiated. Classes deriving from an abstract base can only be instantiated if they override all pure virtual functions.

Polymorphism is implemented at the binary level using *virtual function tables* (*vtables*) that consist of the addresses of all virtual functions of a particular class. Each class containing at least one virtual function has a vtable. Instantiated classes (called *objects*) hold a pointer to their corresponding vtable, which is typically stored in read-only memory. Since each class has its own corresponding vtable, it can also be considered as the type of the object. Throughout this chapter, we refer to the pointer to a vtable as a *vtblptr*, while the pointer to the object is called *thisptr*.

The Itanium C++ ABI [175] defines the vtable layout for Linux systems. The *vtblptr* points to the first function entry in the vtable, and the vtable contains an entry for each virtual function (either inherited or newly declared) in the class. For example, in Figure 4.1, class *B*'s vtable contains two function entries because the class implements virtual functions *funcB1* and *funcB2*. Preceding the function entries, a vtable has two metadata fields: *Runtime Type Identification* (RTTI) and *Offset-to-Top*.

RTTI holds a pointer to type information about the class. Among other things, this type information contains the name of the class and its base classes. However, RTTI is optional and often omitted by the compiler. It is only needed when the programmer uses, e.g., *dynamic_cast* or *type_info*. Hence, a reliable static analysis cannot rely on this information. Classes that do not contain RTTI have the RTTI field set to zero.

Offset-to-Top is needed when a class uses multiple inheritance. As Figure 4.1 shows, a class that has multiple base classes like class *C* also has multiple vtables (a base vtable and one or more sub-vtables). Offset-to-Top specifies the distance between a sub-vtable's own *vtblptr* and the base *vtblptr* at the beginning of the object. In our running example, the *vtblptr* to class *C*'s sub-vtable resides at offset `0x10` in the object, while the *vtblptr* to the base vtable resides at offset `0x0`. Hence, the distance between the two, as stored in the Offset-to-Top field in sub-vtable *C*, is `-0x10`. Offset-to-Top is zero if the vtable is the base vtable of the class or no multiple inheritance is used.

Vtables can contain one additional field, called *Virtual-Base-Offset*, but it is only used in case of virtual inheritance, an advanced C++ feature for classes that inherit from the same base multiple times (diamond-shaped inheritance). An in-depth explanation is out of scope here because VPS needs no adaptations to support virtual inheritance, so we defer to [175].

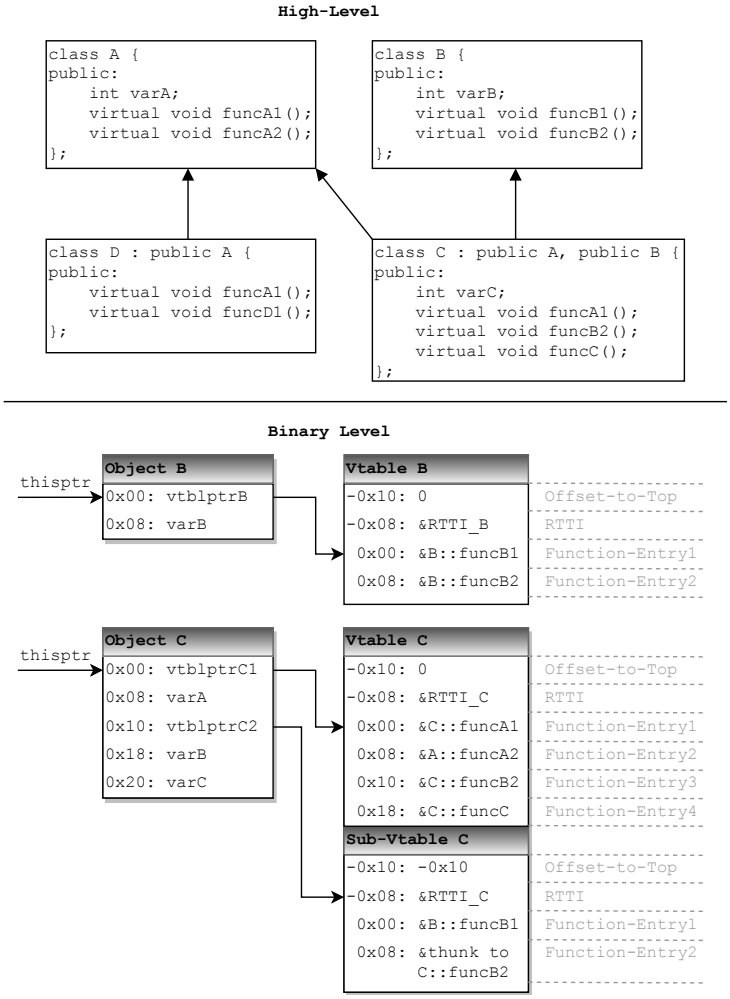


Figure 4.1. An example C++ class structure. The high-level view at the top shows base classes *A* and *B*; derived class *C* which overrides virtual functions *funcA1* and *funcB2*; and derived class *D* which overrides virtual function *funcA1*. The bottom details the binary-level structure of objects of classes *B* and *C*.

4.2.2 C++ Object Initialization

Because VPS secures virtual callsites by protecting the *vtblptr* set at initialization time, we discuss object initialization of classes with vtables here. For the remainder of this chapter, we only consider classes and objects that have at least one virtual function and therefore a vtable.

During object instantiation, the *vtblptr* is written into the object by the *constructor*. Figure 4.1 depicts an object's memory layout at the binary level. The

vtblptr is at offset `0x0`, the start of the object. For classes with multiple inheritance, the constructor also initializes *vtblptr*s to the sub-vtable(s). In addition, the programmer may initialize class-specific fields in the constructor. These fields are located after the *vtblptr* and, in case of multiple inheritance, after any sub-*vtblptr*s.

For classes that have one or more base classes, the constructors of the base classes are called before the derived class's own initialization code. As a result, the base class places its *vtblptr* into the object, which is subsequently overwritten by the derived class's *vtblptr*. Depending on the optimization level, constructors are often inlined, which may complicate analysis that aims to detect constructors.

Abstract classes form a special case: although programmers cannot instantiate abstract classes, and despite the fact that their vtables contain *pure_virtual* function entries, the compiler can still emit code that writes the *vtblptr* to an abstract class into an object. However, this happens only when creating or releasing an object of a derived class, and the abstract *vtblptr* is immediately overwritten.

4.2.3 C++ Virtual Function Dispatch

Because classes can override virtual functions, the compiler cannot determine the target of a call to such a function at compile time. Therefore, the emitted binary code uses an indirect function call through the vtable of the object. This is called a *virtual function call*, or *vcall* for short. In the Itanium C++ ABI [175], the compiler emits the following structure:

```
mov RDI, thisptr
call [vtblptr + offset]
```

The *thisptr* is an implicit call argument, so it is moved into the first argument register, which is `RDI` on Linux x86-64 systems. Next, the `call` instruction uses the *vtblptr* to fetch the target function address from the object's vtable. The `offset` added to the *vtblptr* selects the correct vtable entry. Note that the offset is a constant, so that corresponding virtual function entries must be at the same offset in all vtables of classes that inherit from the same base class.

The same code structure holds for cases that use multiple inheritance. Depending on which (sub-)vtable the virtual function entry resides in, the *vtblptr* either points to the base vtable or one of the sub-vtables. However, if the *vtblptr* points to a sub-vtable, the *thisptr* does not point to the beginning of the object, but rather to the offset in the object where the used *vtblptr* lies. Consider the example from Figure 4.1: when a function in the sub-vtable of class *C* is called, the call uses the *vtblptr* to this sub-vtable, and the *thisptr* points to offset `0x10`

of the object. Because the code structure is the same, the program treats calls through sub-vtables and base vtables as analogous.

4.2.4 VTable Hijacking Attacks

As Section 4.2.3 explained, virtual callsites use the *vtblptr* to extract the pointer to the called virtual function. Since the object that stores the *vtblptr* is dynamically created during runtime and resides in writable memory, an attacker can overwrite it and hijack the control flow at a virtual callsite.

The attacker has two options to hijack an object, depending on the available vulnerabilities: leveraging a vulnerability to overwrite the object directly in memory, or using a dangling pointer to an already deleted object by allocating attacker-controlled memory at the same position (e.g., via a use-after-free vulnerability). In the first case, the attacker can directly overwrite the object's *vtblptr* and use it to hijack the control flow at a vcall. In the second case, the attacker needs not to overwrite any memory; instead, the vulnerability causes a virtual callsite to use a still existing pointer to a deleted memory object. The attacker can control the *vtblptr* by allocating new memory at the same address previously occupied by the deleted object.

4.3 Related Work

In the following, we compare our design against related work to show our advances. We first consider approaches that work on the binary level, which are closest to our system as this involves a complex binary analysis to find which code needs to be instrumented for protection. Afterwards, we consider systems that require source code. These systems lack the binary analysis, but provide similar protections in case source code is available.

4.3.1 Binary-Only Defenses

Table 4.1 provides an overview of the compared approaches and what mechanisms they implement. *Marx* [103] reconstructs class hierarchies from binaries and uses the results for *VTable Protection* and *Type-safe Object Reuse*. VTable Protection verifies at each vcall whether the *vtblptr* resides in the reconstructed class hierarchy. However, the analysis is incomplete and the instrumentation falls back to *PathArmor* [131] for missing results. Hence, the security policy is reduced to PathArmor. VPS does not rely on reconstruction of the class hierarchy, always requiring an exact *vtblptr* match. Marx' Type-safe Object Reuse uses reconstructed

Table 4.1. Overview of binary-only mitigation techniques for C++ binaries

(a) Characteristics of each proposed mitigation. *Marx* (vtable) refers to the VTable protection application of Marx, while *Marx* (type-safe) refers to its type-safe memory reuse approach.

Defense	Binary only	Protects vcalls	Protects type	Protects dangl. ptrs	Tolerates FP vcalls
Marx [103] (vtable)	✓	✓	✗	✓	✓
Marx [103] (type-safe)	✓	✗	✗	✓	n.a.
vfGuard [110]	✓	✓	✗	✓	✗
T-VIP [51]	✓	✓	✗	✓	✗
VTint [153]	✓	✓	✗	✓	✗
VCI [46]	✓	✓	✗	✓	✗
VTPin [118]	RTTI	✗	✗	✓	n.a.
VPS	✓	✓	✓	✓	✓

(b) Security strategy for each mitigation technique

Defense	Security Strategy
Marx [103] (vtable)	*vtblptr* in reconstructed class hierarchy (fallback to PATHARMOR).
Marx [103] (type-safe)	Memory allocator uses class hierarchy as type.
vfGuard [110]	Call target resides in at least one vtable at correct offset.
T-VIP [51]	*vtblptr* and random vtable entry must point to read-only memory.
VTint [153]	Verifies vtable ID, vtable must be in read-only memory.
VCI [46]	*vtblptr* must be statically found, in class hierarchy, or *vfGuard*-allowed.
VTPin [118]	Overwrites *vtblptr* when object freed.
VPS	Check at vcall if object was created at a legitimated object creation site.

class hierarchies to prevent memory reuse between different class hierarchies, reducing the damage that can be done in a use-after-free attack using a dangling pointer. This approach leaves considerable wiggle room for attackers for large class hierarchies, while VPS only allows the correct type. Moreover, Marx only protects the heap whereas VPS protects objects everywhere in memory.

VTint [153] instruments vtables with IDs in order to check their validity at each identified vcall instruction. However, VTint does not prevent an attacker from exchanging the original *vtblptr* with a new pointer to an existing vtable. VPS prevents attackers not only from injecting a fake vtable, but also protects against reuse of existing vtables. Moreover, VTint breaks the binary in case of false positives in their virtual callsite analysis (see Section 4.6.3 for details).

VTPin [118] overwrites the *vtblptr* whenever an object is freed, to protect against use-after-free attacks. However, VTPin fails to protect binaries without RTTI, and does not prevent an attacker from overwriting the *vtblptr*.

vfGuard [110] identifies vtables and builds a mapping which target functions

can appear at which vtable offsets. At virtual callsites, it ensures the target is valid for the offset and the calling convention is consistent. In case of the virtual callsite residing in a virtual function, it can also narrow down the set of allowed targets to be in the same object. However, vfGuard does not verify the validity of the vtable itself, so an attacker could construct a fake vtable as long as the entries do appear in some vtable at the same offset. VPS, on the other hand, protects the original type of the object, completely preventing vtable forgery. Additionally, in cases of falsely identified virtual callsites, vfGuard breaks the instrumented binary while VPS does not.

T-VIP [51] protects vcalls against fake vtables. Each vcall checks if both the *vtblptr* and a random entry point to read-only memory. However, this approach breaks the binary when vtables reside in writable memory (see Section 4.6.1 for details). Moreover, an attacker can still exchange the *vtblptr* to one that satisfies the heuristics. Since VPS does not rely heuristics, its security policy is stronger and also protects the type of the object.

VCI [46] only allows a specific set of vtables at each vcall, mimicking the source code approach *VTV* [130]. Since the analysis is not able to rebuild the sets precisely, they resort to a different kind of set depending on the results of the analysis. Depending on whether the analysis succeeds, the set includes targets that could be resolved statically, targets that reside in the recovered class hierarchy, or targets allowed by vfGuard. However, false positive virtual callsites in VCI break the application (see Section 4.6.3 for details). Furthermore, incomplete class hierarchies may also break the application, i.e., the hierarchies may not be completely recovered due to abstract classes [103]. VPS allows calls through any legitimately created object. Moreover, even in a hypothetical case of a perfect analysis, VCI allows an attacker to change the *vtblptr* to another one residing in the set. Hence, it does not protect the type of the object which VPS does.

4.3.2 Defenses Requiring Source Code

CFIXX [23] is the state-of-the-art in source-based C++ defenses. Like VPS, it stores the *vtblptr* in a safe memory region when an object is created. At each callsite, the *vtblptr* is not extracted from the object itself, but fetched from the safe memory region in order to prevent attacks. Since no check is done against the *vtblptr* stored in the object, vtable hijacking attacks are prevented but not detected. As it is implemented as LLVM compiler extension, CFIXX cannot protect proprietary legacy applications for which no source code is available. Moreover, not all software compiles on LLVM out-of-the-box (a notable example being the

Linux kernel [196]). While enforcement is similar between CFIXX and VPS, our analysis targeting binaries is completely different. Performing accurate analysis on a binary is a challenging problem, especially with regards to object creation sites, where false negatives would break the protected application. Unlike in the source code, the analysis has to take both direct and indirect access to the vtable into account. Second, identifying the virtual callsites for the subsequent instrumentation with security checks is a challenging task since no type information is available. Any false positive in this result breaks the application, which makes an instrumentation capable of handling these necessary.

VTV [130] is a GCC compiler pass, which only allows a statically determined set of vtables at each vcall. This approach is mimicked by most binary-only approaches [46, 51, 103, 110] and used as a ground truth for the accuracy evaluation. Hence, in order to be able to compare VPS with other binary-only approaches, we do the same.

4.4 Threat Model

We assume the attacker is provided with an arbitrary memory read and write primitive. However, the attacker is only able to write to memory locations that are marked as writable by the application (i.e., data memory such as the stack or heap) and cannot modify memory that is only readable/executable (such as code and read-only data). This is the same threat model that is assumed by the original work on CFI and others in this field (e.g., [2, 46, 130, 153]). Furthermore, the attacker's goal is to hijack the control flow at a virtual callsite (since VPS focuses on protecting them). Other attacks such as data-only manipulations or hijacking the control flow by overwriting return addresses are out-of-scope.

4.5 System Overview

VPS is based on the observation that the *vtblptr* is only written during object initialization and cannot change afterwards. Therefore, only the *vtblptr* that is written into the object by the constructor is a valid value. If a *vtblptr* changes after the object was created, a vtable hijacking attack is in progress. Since these attacks target virtual callsites, it is sufficient to check at each virtual callsite if the *vtblptr* written originally into the object still resides there.

Figure 4.2 depicts the differences between a traditional application and a VPS-protected application. The traditional application initializes an object in the code at 2 and uses a vcall and the created object at 4 to call a virtual function. As

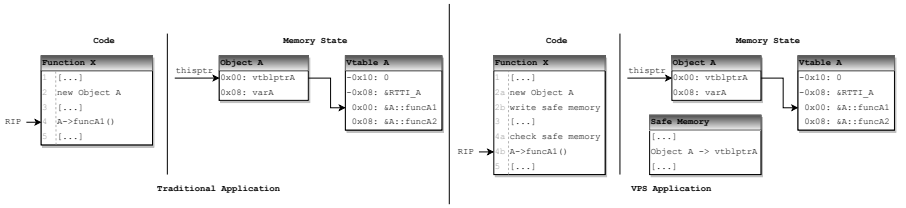


Figure 4.2. High-level overview of the object instantiation and virtual callsite of a traditional application (left side) and a VPS protected application (right side). For both applications the memory state is given while the instruction pointer executes the function call.

explained in Section 4.2.3, the application uses the vtable to decide which virtual function to execute. If an attacker is able to corrupt the object at `3` during the execution, she can place her own vtable in memory and hijack the control flow. In contrast, the VPS-protected application adds two additional functionalities to the executed code. While the object is initialized, the *vtblptr* is also stored in a safe memory region at `2a/b`. Before a vcall is executed at `4a/b`, the application checks if the *vtblptr* in the object is still the same as the one stored for object *A* in the safe memory region. The vcall is only executed when the check succeeds. As a result, the same attacker that is able to corrupt the object at `3` is no longer able to hijack the control flow. In contrast to other binary-only defenses for virtual callsites [46, 51, 103, 110, 153] that allow a specific overestimated set of classes at a virtual function dispatch, VPS has a direct mapping between an object initialization site and the reachable vcalls.

4.6 Analysis Approach

VPS protects binary C++ applications against control flow hijacking attacks at virtual callsites. To this end, we first analyze the binary to identify C++-specific properties and then apply instrumentation to harden it.

We divide the analysis into three phases: *Vtable Identification*, *Object Initialization Operations*, and *Virtual Callsite Identification*. While the Vtable Identification static analysis is an improved and more exact version of Pawlowski et al. [103] (finding vtables in `.bss` and GOT, considering indirect referencing of vtables), the other analyses are novel to VPS. In the remainder of this section, we explain the details of our analysis approach. Note that we focus on Linux x86-64 binaries that use the Itanium C++ ABI [175]. However, our analysis approach is conceptually mostly generic and with additional engineering effort can be applied to other architectures and ABIs. For architecture-specific steps in our analysis, we

describe what to modify to port the step to other architectures.

4.6.1 Vtable Identification

To protect *vtblptr*s in objects, we need to know the location of all vtables in the binary. To find these, our static analysis searches through the binary and uses a set of rules to identify vtables. Whenever all rules are satisfied, the algorithm identifies a vtable. As explained earlier, Figure 4.1 shows a typical vtable structure. The smallest possible vtable in the Itanium C++ ABI [175] consists of three consecutive words (*Offset-to-Top*, *RTTI*, and *Function-Entry*). We use the following rules to determine the beginning of a vtable:

R-1 In principle, our algorithm searches for vtables in read-only sections such as `.rodata` and `.data.rel.ro`. However, there are exceptions to this. If a class has a base class that resides in another module and the compiler uses copy relocation, the loader will copy the vtable into the `.bss` section [53]. Additionally, vtables from other modules can be referenced through the Global Offset Table (GOT), e.g., in position-independent code [0]. To handle these cases where the vtable data lies outside the main binary, we parse the binary's dynamic symbol table and search for vtables that are either copied to the `.bss` section or referenced through the GOT. Note that we do not rely on debugging symbols, only on symbols that the loader uses, which cannot be stripped.

R-2 Recall that the *vtblptr* points to the first function entry in a class's vtable, and is written into the object at initialization time. Therefore, our algorithm looks for code patterns that reference this first function entry. Again, there are special cases to handle. The compiler sometimes emits code that does not reference the first function entry of the vtable, but rather the first metadata field at offset `-0x10` (or `-0x18` if virtual inheritance is used). This happens for example in position-independent code. To handle these cases, we additionally look for code patterns that add `0x10` (or `0x18`) to the reference before writing the *vtblptr* into the object, which is necessary to comply with the Itanium C++ ABI [175]. Our algorithm also checks for the special case where vtables are referenced through the GOT instead of directly.

R-3 As depicted in Figure 4.1, the *Offset-to-Top* is stored in the first metadata field of the vtable at offset `-0x10`. In most cases this field is `0`, but when multiple inheritance is used, this field gives the distance between the base *vtblptr* and the sub-*vtblptr* in the object (see Section 4.2.1). Our algorithm checks the sanity of

this value by allowing a range between `-0xFFFFFF` and `0xFFFFFF`, as proposed by Prakash et al. [110].

R-4 Recall that the RTTI field at offset `-0x8` in the vtable, which can hold a pointer to RTTI metadata, is optional and usually omitted by the compiler. If omitted, this field holds `0`; otherwise, it holds a pointer into the `data` section or a relocation entry if the class inherits from another class in a shared object.

R-5 Most of the vtable consists of function entries that hold pointers to virtual functions. Our algorithm deems them valid if they point into any of the `.text`, `.plt`, or `.extern` sections of the binary, or are a relocation entry.

Abstract classes are an edge case. For each virtual function that has no implementation, the vtable has a function entry to a special function called *pure_virtual*. Because abstract classes are not meant to be instantiated, calling *pure_virtual* throws an exception. Additionally, the first function entries in a vtable can be `0` if the compiler did not emit the code of the corresponding functions (e.g., for destructor functions). To cope with this, Pawlowski et al. [103] allow `0` entries in the beginning of a vtable. We omit this rule because our approach can safely ignore the instantiation of abstract classes, given that *vtblptr*s for abstract classes are overwritten shortly after object initialization.

In case of multiple inheritance, we do not distinguish between vtables and sub-vtables. That is, in the example in Figure 4.1, our approach identifies *Vtable C* and *Sub-Vtable C* as separate vtables. As discussed later, this does not pose any limitations for our approach given our focus on *vtblptr* write operations (as opposed to methods that couple class hierarchies to virtual call sites).

The combination of multiple inheritance and copy relocation poses another edge case. In copy relocation, the loader copies data residing at the position given by a relocation symbol into the `.bss` section without regards to the type of the data. For classes that use multiple inheritance, the copied data contains a base vtable and sub-vtable(s), but the corresponding relocation symbol holds only information on the beginning and length of the data, not the vtable locations. To ensure that we do not miss any, we identify every 8-byte aligned address of the copied data as a vtable. For example, if the loader copies a data chunk of `0x40` bytes to the address `0x100`, we identify the addresses `0x100`, `0x108`, `0x110`, … up to `0x138` as vtables. While this overestimates the set of vtables, only the correct vtables and sub-vtables are referenced during object initialization.

Note that on other architectures, the assumed size of 8-byte per vtable entry as used by our rules may have to be adjusted. For example, Linux on x86 (32-bit)

and ARM would use 4-byte entries. From a conceptual point of view, nothing changes.

4.6.2 Object Initialization Operations

The next phase of our static analysis is based on the observation that to create a new object, its *vtblptr* has to be written into the corresponding memory object during the initialization. The goal of this analysis step is to identify the exact instruction that does this. This step is Linux-specific but architecture-agnostic.

First, we search for all references from code to the vtables identified in the previous step. Because vtables are not always referenced directly, the analysis searches for the following different reference methods:

1. A direct reference to the start of the function entries in the vtable. This is the most common way vtables are referenced.

2. A reference to the beginning of the metadata fields in the vtable. This is mostly used by applications compiled with position-independent code (i.e., MySQL server which additionally uses virtual inheritance).

3. An indirect reference through the GOT. Here, the address to the vtable is loaded from the GOT.

Next, starting from the identified references, we track the data flow through the code (using Static Single Assignment (SSA) form [165]) to the instructions that write the *vtblptr*s into memory during object initialization. We later instrument these instructions, adding code that stores the *vtblptr* in a safe memory region. Our approach handles writes into memory objects agnostic to the location the C++ object resides in (i.e., heap, stack, or global memory).

During our research, we encountered functions with inlined constructors where the compiler emits code that stores the *vtblptr* temporarily in a stack variable to use it at multiple places in the same function. Therefore, to ensure that we do not miss any *vtblptr* write instructions, our algorithm continues to track the data flow even after a *vtblptr* is written into a stack variable. Because we cannot easily distinguish between a temporary stack variable and an object residing on the stack, our algorithm also assumes that the temporary stack variable is a C++ object. While this overestimates the set of C++ objects, it ensures that we do not fail to instrument any *vtblptr* write instructions and hence this overapproximation is safe.

4.6.3 Virtual Callsite Candidates

Because VPS is geared specifically towards protecting vcalls against control flow hijacking, we have to differentiate between vcalls and normal indirect call instructions that are not C++ vcalls. We follow a two-stage approach to make this distinction: we first locate all possible vcall candidates and subsequently verify them. The verification step consists of a static analysis component and a dynamic one. In the following, we first explain how we identify *candidate* virtual callsites.

We use a similar technique as previous work [46, 51, 110, 153]. The analysis searches for the vcall pattern described in Section 4.2.3, where the *thisptr* is the first argument (stored in the `RDI` register on Linux x86-64) to the called function and the vcall uses the *vtblptr* to retrieve the call target from the vtable. Note that the *thisptr* is also used to extract the *vtblptr* for the call instruction. A typical vcall looks as follows:

```
mov RDI, thisptr
mov vtblptr, [thisptr]
call [vtblptr + offset]
```

Note that these instructions do not have to be consecutive in the application, but can be interspersed with other instructions. Two patterns can be derived from this sequence: the first argument register always holds the *thisptr*, and the call instruction target can be depicted as `[[`*thisptr*` ] + offset]`, where `offset` can be `0` and therefore omitted. This specific dependency between call target and first argument register is rare for non-C++ indirect calls.

With the help of the SSA form, our algorithm traces the data flow backwards starting from the first argument and call target of indirect calls. Either when the data flow intersects or when the beginning of the function is reached (i.e., if an argument register of the function is used directly for the call instruction), the analysis checks if the previously described dependency is satisfied. If so, we consider the indirect call instruction a vcall candidate.

Note that the same pattern holds for classes with multiple inheritance. As described in Section 4.2.3, when a virtual function of a sub-vtable is called, the *thisptr* is moved to the position in the object where the sub-vtable resides. Therefore, the first argument holds *thisptr* `+ distance`, and the call target `[[`*thisptr* `+ distance] + offset]`. This still satisfies the aforementioned dependency between first argument and call target. Furthermore, the pattern is also applicable for Linux ARM, Linux x86, and Windows x86-64 binaries, requiring only a minor modification to account for the specific register or memory location used for the

first argument in the platform's calling convention (`R0` for ARM, the first stack argument for Linux x86, and `RCX` for Windows x86-64).

To effectively protect vcalls, it is crucial to prevent false positive vcall identifications, as these may break the application during instrumentation. This is also required for related work [46, 51, 110, 153]. While the authors of prior approaches report no false positives with the above vcall identification approach, our research shows that most larger binary programs do indeed contain patterns that result in indirect calls being wrongly classified as virtual callsites.

A possible explanation for the lack of false positives in previous work is that most prior work focuses on Windows x86 [51, 110, 153], where the calling conventions for vcalls and other call instructions differ. That is, on Windows x86, the *thisptr* is passed to the virtual function via the `ECX` register (*thiscall* calling convention), while other call instructions pass the first argument via the stack (*stdcall* calling convention) [176]. This is not the case for Windows x86-64 and Linux (x86 and x86-64). On these architectures, the *thisptr* is passed as the first argument in the platform's standard calling convention (*Microsoft x64*, *cdecl* and *System V AMD64 ABI*, respectively). While Elsabagh et al. [46], who focus their work on Linux x86, did not report false positives, our evaluation does show false positives in the same application set. We contacted Elsabagh et al., but were unable to find an explanation for these differing results. Unfortunately, given that source is unavailable for the approach of Elsabagh et al., we were unable to reproduce their results.

4.6.4 Virtual Callsite Verification

Because a single false positive can break our approach, the next phase in our static analysis verifies the virtual callsite candidates. This phase is divided into the following steps:

Data flow graphs Our analysis starts by using SSA form to track the data flow backwards from all identified vtable references in the code. We perform this data flow tracking interprocedurally. This means that if the data flow ends in an argument register (as specified by the calling convention), the tracking continues at the call instruction that calls this function. If the data flow ends in the return value register `RAX`, we continue the data flow tracking at each return instruction of the function that created the return value (i.e., the function called just before the return site). This analysis creates a data flow graph with the instruction referencing the vtable as the only sink node and the instructions where the data flow starts as source nodes.

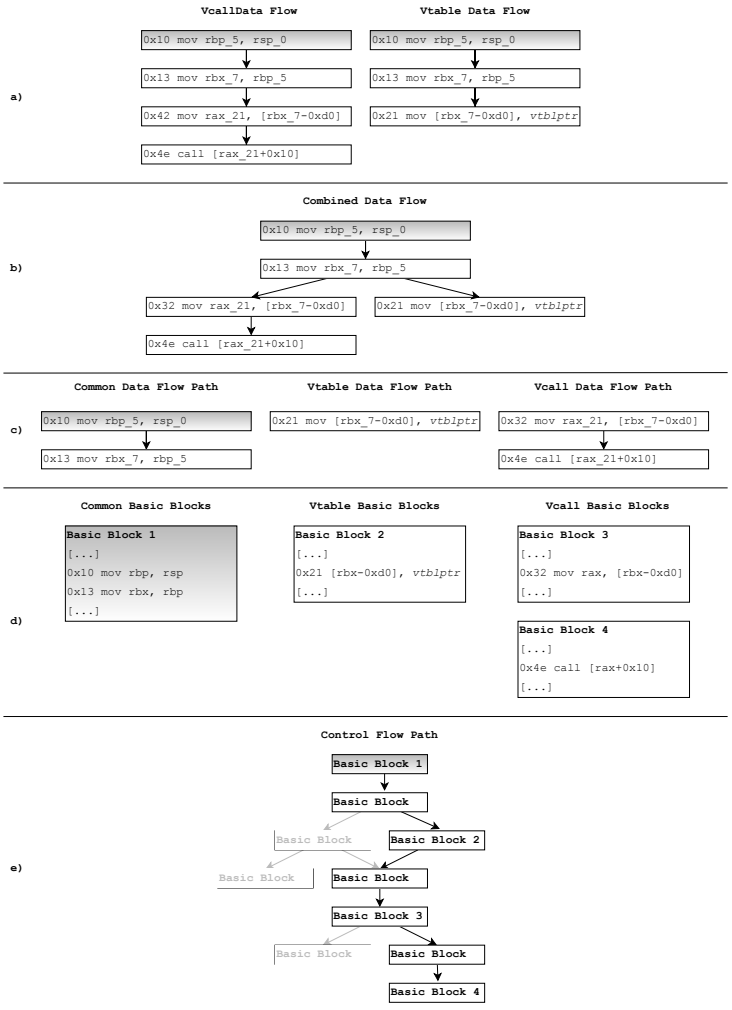


Figure 4.3. The five steps of the vcall verification phase. Step *a)* shows the data flow graph in SSA form, with the starting node in gray. Step *b)* combines the data flow graphs of *a)*. Step *c)* separates the paths through the data flow graph into three components. Step *d)* shows the basic blocks corresponding to the data flow paths. Step *e)* shows a path through the CFG containing all previously identified basic blocks.

We perform the same data flow analysis for the call target of each identified virtual callsite candidate. Again, this creates a data flow graph with the candidate call instruction as the only sink node and the instructions starting the data flow as source nodes. Figure 4.3a shows an example.

Combined data flow graphs Next, the algorithm merges the data flow graphs for indirect calls with those for vtable references, joining graphs that share at least one source, as shown in Figure 4.3b. This creates a new combined graph depicting the data flow to vtable-referencing instructions that could potentially reach the vcall candidate. In Figure 4.3b, both graphs are merged because they share the instruction at address `0x10` as their source node.

Data flow paths The next step searches all paths from the source node to the vtable-referencing node and from the source node to the vcall node. When at least one path is found, we split the path into a common prefix (nodes with addresses `0x10` and `0x13` in Figure 4.3c), a part belonging only to the vtable-referencing instruction (node with address `0x21`), and a part belonging only to the vcall instruction (nodes with addresses `0x32` and `0x4e`).

Basic blocks Note that the data flow graphs do not give any information on whether a control flow path exists that starts at the source node instruction, visits the vtable-referencing instruction, and ends at the vcall instruction. For this reason, we search the corresponding basic block for each data-flow node, as shown in Figure 4.3d, considering basic blocks that correspond to multiple data flow nodes only once.

Control flow path Next, the analysis looks for a path through the Control Flow Graph (CFG) of the function that traverses the data source, the vtable-referencing instruction, and the vcall instruction. We do this by successively searching for sub-paths between basic blocks of interest, starting with a path between `Basic Block 1` and `Basic Block 2`, then a path between `Basic Block 2` and `Basic Block 3`, and so on. We traverse the CFG in a breadth-first search (BFS) manner, so that the search yields the shortest possible paths. After we find paths for all relevant basic block pairs, we combine the paths into one control flow path, as shown in Figure 4.3e.

A special case occurs if a basic block relevant to the vcall already resides in the sub-path found for the vtable-referencing instruction. For example, suppose that in the given example `Basic Block 3` lies between `Basic Block 1` and `Basic Block 2` in the CFG. Then, adding `Basic Block 4` to the path yields a full path with the desired semantics. However, our algorithm does not find this path because there is no sub-path from `Basic Block 2` to `Basic Block 3`. To handle this, we check if the destination basic block resides in a previously found path. If it does, the analysis uses the next basic block as the target instead.

Symbolic execution As a last step, we symbolically execute the obtained control flow paths to track the flow of the *vtblptr* through the binary. When an instruction writes a *vtblptr* into the memory state, we replace that *vtblptr* with a symbolic value. To keep the analysis scalable to large real-world applications, our symbolic execution simply executes basic blocks without checking whether branches can actually be taken in a concrete execution. If a basic block contains a call instruction that is not part of our original data flow path, we simply execute a return instruction immediately after the call instead of symbolically executing the called function. When the symbolic execution reaches the vcall instruction, we check the obtained memory state to verify that the *vtblptr* is used for the call target. If so, we conclude that the vcall candidate is in fact a vcall.

In addition to explicit vtable-referencing instructions, this analysis checks implicit vtable references as well. In case the earlier backward data flow analysis shows that a vcall target stems from the first argument register, we check whether the calling function is a known virtual function (by checking whether the function resides in any previously identified vtable). If it is, we add a special virtual function node to the data flow graph. We then search for a path from this virtual function node to the vcall instruction. If a path is found, we apply the steps described previously for transforming the data flow path to a control flow path. For such paths, before starting the symbolic execution, we add an artificial memory object containing the *vtblptr* and place the *thisptr* in the first argument register. This way, we simulate an implicit use of the vtable through the initialized object.

We perform the whole vcall verification analysis in an iterative manner. When the data flow tracking step stops at an indirect call instruction, we repeat it as soon as our analysis has verified the indirect call as a vcall instruction and has therefore found corresponding vtables for resolving the target. The same applies to data flow tracking that stops at the beginning of a virtual function (because no caller is known). As soon as we are able to determine a corresponding vcall instruction, we repeat the analysis. The analysis continues until we reach a fixed point where the analysis fails to find any new results.

4.6.5 Dynamic Virtual Call Profiling

Our approach includes a dynamic profiling phase that further refines the vcall verification. During this phase, we execute the application with instrumentation code added to all virtual callsite candidates. Whenever the execution reaches a vcall, the instrumentation code verifies that the first argument contains a valid *thisptr*. To verify this, we check if the first element of the object the *thisptr* points

to contains a valid pointer to a known vtable (*vtblptr*). If it does, we consider the vcall verified. Otherwise, we regard the vcall as a false positive of the static analysis and discard it.

Because this phase only instruments vcall candidates identified by the static analysis described in Section 4.6.3, it is safe to assume the dependency between first argument and call instruction target. Hence, the above dynamic profiling check is sufficient to remove false positives seen during the profiling run, given that the odds of finding a C-style indirect callsite with such a distinctive pattern that uses C++ objects is extremely unlikely. We did not encounter any such case during our comprehensive evaluation.

4.7 Instrumentation Approach

VPS protects virtual callsites against control-flow hijacking attacks by instrumenting the application using the results from the analysis phase. We instrument two parts of the program: *Object Initialization* and *Virtual Callsites*. The following describes how both kinds of instrumentation work.

4.7.1 Object Initialization

We use the data collected in Section 4.6.2 to instrument object initialization, specifically the instruction that writes the *vtblptr* into the object. When an object is created, the instrumentation code stores a key-value pair that uses the memory address of the object as the *key* and maps it to the *vtblptr*, which is the associated *value*. To prevent tampering with this mapping, we store it in a safe memory region.

Recall that when a C++ object is created that inherits from another class, the initialization code first writes the *vtblptr* of the base class into the object, which is then overwritten by the *vtblptr* of the derived class. Our approach is agnostic to inheritance and simply overwrites the *vtblptr* in the same order (because each *vtblptr* write instruction is instrumented).

Similarly, our approach is agnostic to multiple inheritance, because object initialization sites use the address where the *vtblptr* is written as the object address. As explained in Section 4.2.3, at a virtual callsite the *thisptr* points to the address of the object the used *vtblptr* resides in. For a sub-vtable, this is not the beginning of the object, but an offset somewhere in the object (in our running example in Figure 4.1 offset `0x10`). Because this is exactly the address that our approach uses as the key for the safe memory region, our approach works for multiple inheritance without any special handling.

Moreover, despite the fact that we instrument only object initialization and ignore object deletion, our approach does not suffer from consistency problems: when an object is deleted and its released memory is reused for a new C++ object, the instrumentation code for the initialization of this new object automatically overwrites the old value in the safe memory region with the current *vtblptr*.

4.7.2 Virtual Callsites

Because a single false positive virtual callsite can break the application, we designed the vcall instrumentation code such that it can detect false positives and filter them out. In doing so, the vcall instrumentation continuously refines the previous analysis results. The vcall instrumentation consists of two components: *Analysis Instrumentation* and *Security Instrumentation*. In the following, we describe both components.

Analysis instrumentation We add analysis instrumentation code to all vcall candidates that we were unable to conclusively verify in our prior analysis. For verified vcall sites, we only add security instrumentation and omit the analysis code.

Before executing a vcall candidate, the analysis instrumentation performs the same check as the dynamic profiling phase described in Section 4.6.5. If the check fails, meaning that this is not a vcall but a regular C-style indirect call, we remove all instrumentation from the call site. If the check succeeds, we replace the analysis instrumentation with the more lightweight security instrumentation for verified virtual callsites described in Section 4.7.2, and immediately run the security instrumentation code.

Through our use of adaptive instrumentation, our approach is able to cope with false positives and further refine the analysis results during runtime. By caching the refined results on disk, we can reuse these results in later launches of the same application, so that VPS's performance improves over time.

Because the analysis instrumentation verifies any potential false positives at runtime, the static vcall verification from Section 4.6.4 and the dynamic profiling from Section 4.6.5 are optional. Omitting these steps does not affect the correctness of our approach, although we recommend using them for optimal performance.

Security instrumentation We protect verified vcall sites against control-flow hijacking by adding security instrumentation code that runs before allowing the vcall. The instrumentation uses the *thisptr* in the first argument register to re-

trieve the *vtblptr* stored for this object in the safe memory region. To decide whether to allow the vcall, the instrumentation code compares the *vtblptr* from the safe memory region with the one stored in the actual object used in the vcall. If they are the same, the instrumentation allows the vcall. If the two pointers differ, we terminate the execution with a warning that a control-flow hijacking attack was detected.

4.8 Implementation

Based on the approach from Section 4.6, we integrated our static analysis into the *Marx* framework [103]. This framework provides a basic symbolic execution based on the VEX-IR from the Valgrind project and data structures needed for C++ binary analysis. It is written in C++ and targets Linux x86-64 binaries. To support integration of our approach into the *Marx* framework, we added support for SSA and a generic data-flow tracking algorithm.

Because the framework uses VEX-IR as the back-end, it is easily extendable to other architectures. The same is true for our approach, which is mostly independent from the underlying architecture, as described in Section 4.6. To balance precision and scalability, the symbolic execution emulates only a subset of the 64-bit VEX instructions that suits our focus on vtable-centered data-flow tracking in real-world applications.

We use IDAPython for vtable identification and CFG extraction. Additionally, we use instruction data provided by IDA Pro to support the SSA transformation, and use Protocol Buffers to export the results in a programming language–agnostic format. We implement dynamic profiling with Pin [91]. We build the runtime component of VPS on top of Dyninst v9.3.2 [11]. Dyninst is responsible for installing object initialization and (candidate) virtual callsite hooks. We inject these wrappers into the target program's address space by preloading a shared library.

To configure the safe memory region, our preloaded library maps the lower half of the address space as a safe region at load time; this is straightforward for position-independent executables as their segments are mapped exclusively in the upper half of the address space by default. To compute safe addresses, we subtract 64 TB[1] from the addresses used by object initialization or virtual calls. To thwart value probing attacks in the safe region, we (1) mark all safe region pages as inaccessible by default and make them accessible on demand, and (2) use a fixed offset chosen randomly at load time for writes to the safe region. To

[1]Linux x86-64 provides 47 bits for user space mappings, and $2^{47} = 128$ TB.

achieve the latter, we write a random value to the `gs` register and use it as the offset for all accesses to the safe region. To mark pages readable/writable, we use a segfault handler that uses `mprotect` to allow accesses from our library.

We omit an evaluation of potential optimizations already explored in prior work [23, 83], such as avoiding Dyninst's penalties for (re)storing unclobbered live registers or removing trampoline code left over after nopping out analysis instrumentation code. Similarly, we do not implement hash-based safe region compression that would reduce virtual and physical memory usage and allow increased entropy in the safe region, nor do we use Intel MPK [192] to further secure the safe region. We consider these optimizations orthogonal to our work.

4.9 Evaluation

In this section, we evaluate VPS in terms of performance and accuracy. We focus our evaluation on MySQL, Node.js, and the fifteen C++ benchmarks found in SPEC CPU2006 and CPU2017.

4.9.1 Virtual Callsite Identification Accuracy

In order to measure the accuracy of the protection of VPS, we evaluate the accuracy of the vcall identification analysis. As applications for our evaluation, we use the C++ programs of SPEC CPU2006 and SPEC CPU2017 that contain virtual callsites, as well as the MySQL server binary (5.7.21), and the Node.js binary (8.10.0). We used the default optimization levels (`O2` for CPU 2006, `O3` for all others). All programs are compiled with GCC 8.1.0. In order to gain a ground truth of virtual callsites, we use VTV [130] and compare against our analysis results. Since VTV leverages source code information, its results are usually used as ground truth for binary-only approaches focusing on C++ virtual callsites. Unfortunately, compiling *450.soplex* results in a crash and it is therefore omitted. Table 4.2 shows the results of our vcall accuracy evaluation.

We observe that the analysis of VPS is capable of identifying the vast majority of virtual callsites in the binary, ranging from 91.4% (*510.parest_r*) to all vcalls detected (several benchmarks). Our average recall is 97.7% on SPEC CPU2006 and 97.4% on SPEC CPU2017. With the exception of one outlier (*526.blender_r* with a precision of 68.3%) we have a low number of false positives, with precisions ranging from 87.0% (*447.dealII*) to no false positives at all (several benchmarks). To cope with the problem of false positive identifications, we verify vcalls before we actually instrument them with our security check. The static analysis verification is able to verify 37.9% in the best case (*526.blender_r*) and in the worst

Table 4.2. Accuracy evaluation of our vcall identification.

(a) This table shows (1) the ground truth generated by VTV; (2) static vcall identification details, depicting the true positives, false positives, recall, and precision for indirect call instructions identified as vcall; and (3) static vcall verification results, displaying the number of verified vcalls, its percentage, and the number of false positives.

		Static identification				**Static verification**		
Program	*# GT*	*# TP*	*# FP*	*Recall (%)*	*Prec. (%)*	*#*	*%*	*# FP*
447.dealII	1,558	1,443	215	92.6	87.0	379	24.3	7
450.soplex	–	–	–	–	–	–	–	–
453.povray	102	102	10	100.0	91.1	32	31.4	0
471.omnetpp	802	800	0	99.8	100.0	245	30.6	0
473.astar	1	1	0	100.0	100.0	0	0.0	0
483.xalancbmk	13,440	12,915	17	96.1	99.9	2,122	15.8	0
Average (SPEC CPU2006)				**97.7**	**95.6**		**20.4**	
510.parest_r	4,678	4,275	528	91.4	89.0	660	14.1	13
511.povray_r	122	122	14	100.0	89.7	33	27.1	0
520.omnetpp_r	6,430	6,190	23	96.3	99.6	1,585	24.7	0
523.xalancbmk_r	33,880	33,069	12	97.6	100.0	1,948	5.8	0
526.blender_r	174	172	80	98.9	68.3	66	37.9	0
541.leela_r	1	1	0	100.0	100.0	0	0.0	0
Average (SPEC CPU2017)				**97.4**	**91.1**		**18.3**	
MySQL	11,876	11,589	179	97.6	98.5	1,330	11.2	3
Node.js	12,643	12,320	353	97.5	97.2	1,538	12.2	10

(b) Static and dynamic verification results. For each program, this table depicts the number of verified vcall instructions and its percentage, verified false positives and removed false positive identified vcalls. Cases where dynamic verification failed due to VTV false positives are in parentheses.

	Static and dynamic verification			
Program	*#*	*%*	*# FP*	*# removed*
447.dealII	423	27.2	18	0
450.soplex	–	–	–	–
453.povray	55	53.9	0	6
471.omnetpp	530	66.1	0	0
473.astar	0	0.0	0	0
483.xalancbmk	3,792	28.2	1	0
Average (SPEC CPU2006)		**35.1**		
510.parest_r	(660)	(14.1)	(13)	–
511.povray_r	62	50.8	0	6
520.omnetpp_r	2,286	35.6	6	0
523.xalancbmk_r	4,961	14.6	0	0
526.blender_r	70	40.2	0	49
541.leela_r	0	0.0	0	0
Average (SPEC CPU2017)		**25.9**		
MySQL	(1,330)	(11.2)	(3)	–
Node.js	2,559	20.2	45	118

```
92  /**
93  * Destroy the object pointed to by a pointer type.
94  */
95  template<typename _Tp>
96      inline void
97      _Destroy(_Tp* __pointer)
98      { __pointer->~_Tp(); }
```

```
2545  Vector<double> us[dim];
2546  for (unsigned int i=0; i<dim; ++i)
2547      us[i].reinit (dof_handler.n_dofs());
```

Figure 4.4. Two source code snippets where VTV fails to identify a virtual callsite. The top snippet was taken from `stl_construct.h`; the bottom is from `grid_generator.cc`.

case none. On average we verified 20.4% on SPEC CPU2006 and 18.3% on SPEC CPU2017. Dynamic verification (see Section 4.6.5) considerably improves verification performance, verifying 35.1% and 25.9% respectively. Unfortunately, we were not able to execute *510.parest_r* and *MySQL* with VTV. Both applications crashed with an error message stating that VTV was unable to verify a vtable pointer (i.e., a false positive).

A manual analysis of the missed virtual callsites reveals two possibilities for a miss: the data flow was too complex to be handled correctly by our implementation, or the described pattern in Section 4.6.3 was not used. The former can be fixed by improving the implemented algorithm that is used for finding the described pattern. In the latter, the *vtblptr* is extracted from the object, however, a newly created stack object is used as *thisptr* for the virtual callsite which does not follow a typical C++ callsite pattern. This could be addressed by considering additional vcall patterns, at the risk of adding false positives. Given our already high recall rates, we believe this would not be a favorable trade-off.

As evident from the table, our verification process verified 86 cases which VTV did not recognize as virtual callsites. A manual verification of all cases show that these are indeed vcall instructions and hence missed virtual callsites by VTV. For example, the top snippet in Figure 4.4 depicts the relevant code for 34 of these cases that are linked to the compiler provided file `stl_construct.h`. Line 98 provides the missed vcall instruction that calls the destructor of the provided object. Since the destructor of a class is also a virtual function, it is invoked with the help of a virtual callsite. Another example is in the bottom snippet of Figure 4.4 for *510.parest_r*. Here a vector is created and the function `reinit()` is invoked on line 2547. However, since the class `dealii::Vector<double>` is

Table 4.3. Results of *Marx*'s vcall accuracy evaluation. For each application this table shows (i) the ground truth generated by VTV; (ii) static vcall identification, depicting the number of indirect call instructions identified as vcall that are true positives, the false positives, recall and precision.

		Static identification			
Program	*# GT*	*# TP*	*# FP*	*Recall (%)*	*Prec. (%)*
447.dealII	1,558	1,307	122	83.9	91.5
450.soplex	–	–	–	–	–
453.povray	102	98	10	96.1	90.7
471.omnetpp	802	701	3	87.4	99.6
473.astar	1	1	0	100.0	100.0
483.xalancbmk	–	–	–	–	–
Average (SPEC CPU2006)				**91.8**	**95.4**
510.parest_r	4,678	3,673	295	78.5	92.6
511.povray_r	122	115	11	94.3	91.3
520.omnetpp_r	6,430	5,465	22	85.0	99.6
523.xalancbmk_r	33,880	23,541	33	69.4	99.9
526.blender_r	174	171	1,347	98.3	11.3
541.leela_r	1	0	0	0.0	0.0
Average (SPEC CPU2017)				**70.9**	**65.8**
MySQL	11,876	10,867	1,214	81.3	88.8
Node.js	–	–	–	–	–

provided by the application and `reinit()` is a virtual function of this class, this function call is translated into a virtual callsite. We contacted the VTV authors about this issue and they confirmed that this happens because the compiler accesses the memory of the objects directly when calling the virtual function in the internal intermediate representation. Usually, the compiler accesses them while going through an internal *vtblptr* field. Unfortunately, to fix this issue in VTV would require a lot of non-trivial work since the analysis has to be enhanced.

A direct comparison of the accuracy with other binary-only approaches is difficult since different test sets are used to evaluate it. For example, vfGuard evaluates the accuracy of their approach against only two applications, while T-VIP is only evaluated against one. VTint states absolute numbers without any comparison with a ground truth. VCI evaluates their approach against SPEC CPU2006, but the numbers given for the ground truth created with VTV differ completely from ours (i.e., 9,201 vs. 13,440 vcalls for *483.xalancbmk*) which makes a comparison difficult. Additionally, the paper reports no false positives during their analysis which we encounter in the same application set with a similar identification technique. Unfortunately, as discussed in Section 4.6.3, we were

Table 4.4. Object creation accuracy results. For each application, this table shows the number of vtable references in the code as found in the ground truth, and as identified or missed by our analysis.

Program	*# GT*	*# identified*	*# missed*
447.dealII	–	–	–
450.soplex	102	228	0
453.povray	103	226	0
471.omnetpp	372	871	0
473.astar	0	8	0
483.xalancbmk	2,918	6,530	0
510.parest_r	12,482	25,804	0
511.povray_r	103	224	0
520.omnetpp_r	1,381	3,280	0
523.xalancbmk_r	2,790	6,323	0
526.blender_r	–	–	–
541.leela_r	87	180	0
MySQL	8,532	11,524	0
Node.js	7,816	19,204	0

not able to determine the reason for this. Since Marx is open source, we analyzed our evaluation set with it. In order to create as few false positives as possible we used its conservative mode. Unfortunately, Marx crashed during the analysis of *483.xalancbmk* and *Node.js*. The results of the analysis can be seen in Table 4.3. Compared to Marx, we have considerably higher recall with similar precision. Averaged over the CPU2006 benchmarks supported by Marx, VPS achieves 98.2% recall (91.8% for Marx) and on CPU2017 97.4% versus 70.9% respectively. This does not come at the cost of more false positives, as our precision is similar on CPU2006 (94.5% vs. 95.4%) and much better on CPU2017 (91.1% vs. 65.8%).

Overall, our analysis shows that VPS is precise enough to provide an application with protection against control flow hijacking attacks at virtual callsites. In addition, it shows that even source code approaches such as VTV do not find all virtual callsite instructions and can benefit from binary-only approaches such as VPS. Furthermore, the occurring number of false positive identification underlines the design approach to handle them during the instrumentation rather than confide in not having them.

4.9.2 Object Initialization Accuracy

To avoid breaking applications, VPS must find and instrument all valid object initialization sites. To ensure that this is the case, we compare the number of

vtable-referencing instructions found by VPS to a ground truth. We generate the ground truth with an LLVM 4.0.0 pass that instruments Clang's internal function `CodeGenFunction::InitializeVTablePointer()`, which Clang uses for all vtable pointer initialization.

Table 4.4 shows the results for the same set of applications we used in Section 4.9.1. We omit results for *447.dealII* from SPEC CPU 2006 and *526.blender_r* from SPEC CPU 2017 because these benchmarks fail to compile with LLVM 4.0.0. The results for the remaining applications show that our analysis conservatively overestimates the set of vtable-referencing instructions, ensuring the security and correctness of VPS at the cost of a slight performance degradation due to the overestimated instruction set.

4.9.3 Performance

This section evaluates the runtime performance of VPS by measuring the time it takes to run each C++ benchmark in SPEC CPU2006 and CPU2017. We compare VPS-protected runtimes against the baseline of original benchmarks without any instrumentation. We compile all test cases as position-independent executables with GCC 6.3.0. For each benchmark, we report the median runtime over 11 runs on a Xeon E5-2630 with 64 GB RAM, running CentOS Linux 7.4 64-bit. We use a single additional run with more logging enabled to obtain statistics such as the number of executed virtual calls. Table 4.5 details our results.

For each benchmark, the first group in Table 4.5 shows the number of instrumented object initializations, positive virtual calls, and candidate virtual calls found by our analysis. These numbers match show the variety in properties of C++ applications; some programs make little to no use of virtual dispatching, e.g., *444.namd*, *508.namd_r*, *531.deepsjeng_r*, and *473.astar*. Others contain thousands of object initializations and virtual callsites, e.g., *510.parest_r* with over 12,000 object initializations, or *483.xalancbmk* in CPU2006 with more than 1,300 verified virtual callsites.

The second group in Table 4.5 first shows the number of verified virtual calls (true positive) and regular indirect calls (false positive). These numbers show that most vcall candidates turn out to be real vcalls, indicating that our analysis is relatively conservative and that certain, possibly heuristic-based improvements, may be possible. This second group also depicts the absolute numbers of executed virtual calls and object initializations. With numbers in the billions for some applications, it is clear that our instrumentation must be lightweight.

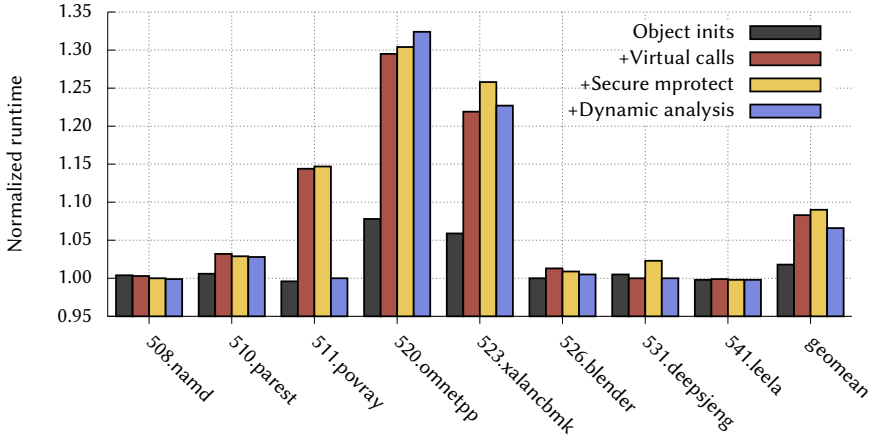
Finally, the third group in Table 4.5 shows the absolute runtime in seconds for each benchmark, comparing uninstrumented baseline runs to VPS-protected

Table 4.5. VPS performance results and runtime statistics. For each binary, this table shows (1) **binary instrumentation** details, depicting the number of instrumented object initializations (*#inits*), positive virtual calls (*#positive*), and candidate vcalls (*#candidates*); (2) **runtime statistics**, listing the number of true positive (*#TP*) and false positive (*#FP*) virtual calls, and the total number of virtual calls (*#vcalls*) and object initializations (*#inits*); and (3) **runtime overhead**, listing runtime overhead (VPS) compared to the baseline (*base*).

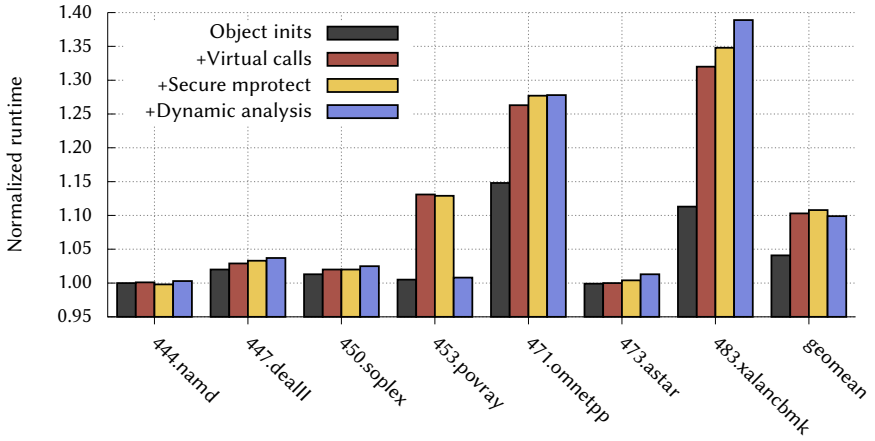
	Binary instrumentation			**Runtime statistics**				**Runtime overhead**	
	# inits	*# pos.*	*# cand.*	*# TP*	*# FP*	*# vcalls*	*# inits*	*base*	VPS
444.namd	6	0	2	0	0	0	2,018	343,5	342,9 (+ 0%)
447.dealII	4,283	161	1,459	47	0	97m	21m	289.7	299.2 (+ 3%)
450.soplex	120	195	364	48	0	1,665,968	40	215.8	220.2 (+ 2%)
453.povray	98	21	91	21	6	101,743	162	135.8	153.3 (+13%)
471.omnetpp	507	117	677	327	0	1,585m	2,156m	290.0	370.2 (+28%)
473.astar	0	0	1	0	0	0	0	350.3	351.6 (+ 0%)
483.xalancbmk	4,554	1,348	11,623	1,639	0	3,822m	2,316m	185.0	249.4 (+35%)
Geometric mean (SPEC CPU2006)									**+ 9%**
508.namd_r	48	0	0	0	0	0	21	271.8	271.8 (+ 0%)
510.parest_r	12,206	243	4,539	350	4	2,625m	119m	586.3	603.1 (+ 3%)
511.povray_r	113	19	121	21	6	4,577	183	498.7	572.0 (+15%)
520.omnetpp_r	2,591	447	5,310	751	0	7,958m	2,070m	507.4	661.7 (+30%)
523.xalancbmk_r	4,512	801	30,771	2,844	0	4,873m	2,314m	366.8	461.5 (+26%)
526.blender_r	43	37	174	4	46	11	3	325.8	328.6 (+ 1%)
531.deepsjeng_r	0	0	0	0	0	0	0	345.1	353.1 (+ 2%)
541.leela_r	177	0	2	0	0	0	404,208	535.5	534.6 (+ 0%)
Geometric mean (SPEC CPU2017)									**+11%**

runs. Runtime overhead varies from 0% for programs with little to no virtual dispatch code to 35% for the worst-case scenario (*483.xalancbmk*). In almost all cases, we see a correlation between increased overhead and number of instrumentation points (object initializations and virtual calls). An exception is *511.povray_r*, which shows a 15% performance decrease despite a relatively low number of vcalls and object initializations. Further inspection shows that this is caused by the 6 false positives candidate vcalls; if we disable hot-patching, our vcall instrumentation code is called over 18 billion times. While we remove instrumentation hooks for the majority of these cases, which are not real vcalls, our current implementation does not remove the Dyninst trampolines. These trampolines are the source of the unexpected overhead.

To better understand the overhead of VPS, we gathered detailed statistics for both SPEC CPU2006 and SPEC CPU2017 in varying configurations. Figure 4.6 shows the results. Comparing to the baseline, we first run SPEC with only instrumentation for object initializations enabled. In this run, the entire safe region is read/writable and the instrumentation only (1) computes the address in the safe region to store the vtable pointer at, and (2) copies the vtable pointer there. In



(a) Microbenchmarks for SPEC CPU2017



(b) Microbenchmarks for SPEC CPU2006

Figure 4.6. Normalized runtime for C++ programs in SPEC CPU2006 and CPU2017, with cumulative configurations: (1) only instrument *object initializations*; (2) also instrument *virtual call instructions*; (3) *secure the safe region* by marking all pages unwritable, and only selectively `mprotect`-ing them if they are accessed from our own instrumentation code; and (4) include offline *dynamic analysis* results, reducing the need for hot-patching.

the second configuration, we additionally instrument virtual calls. We check whether candidates are actual vcalls by testing the call's first argument and, if it can be dereferenced, looking this value up in the list of known vtables. We then either patch verified vcalls to enable the fast path, or remove instrumentation for false positives. The fast path fetches the vtable pointer by dereferencing

the first argument, and then compares it against the value stored in the safe region. The third configuration additionally makes the safe region read-only and uses a segfault handler to mark pages writable on demand. Finally, the fourth configuration includes dynamic analysis results, removing the need to hot-patch previously verified vcalls at runtime. Figure 4.6 shows that the majority of VPS's overhead stems from object initializations and virtual callsite instrumentation.

Overall, with a performance overhead of 9% for SPEC CPU2006 and 11% for SPEC CPU2017 (geometric means), VPS shows that binary-level CFIXX is possible with moderate performance impact.

4.10 Discussion

This section first discusses the susceptibility of VPS to Counterfeit Object-oriented Programming [120]. Following this, we discuss the limitations of VPS.

4.10.1 Counterfeit Object-Oriented Programming

CFI approaches targeting C++ must cope with advanced attackers using Counterfeit Object-oriented Programming (*COOP*) attacks [39, 120]. This attack class thwarts defenses that do not accurately model C++ semantics. As we argue below, VPS reduces the attack surface sufficiently that practical COOP attacks are infeasible.

For a successful *COOP* attack, an attacker must control a container filled with objects, with a loop invoking a virtual function on each object. The loop may be an actual loop, called a *main loop gadget*, or can be achieved through recursion, called a *recursion gadget*. We refer to both types as *loop gadget*. The attacker places counterfeit objects in the container, allowing them to hijack control flow when the loop executes each object's virtual function. To pass data between the objects, the attacker can overlap the objects' fields.

The first restriction VPS imposes on an attacker is that it prevents filling the container with counterfeit objects; because the objects were not created at legitimate object creation sites, the safe memory does not contain stored *vtblptr*s for them. Only two conceivable ways would allow an attacker to craft a container of counterfeit objects under VPS: either the application allows attackers to arbitrarily invoke constructors and create objects, or the attacker can coax the application into creating all objects needed for an attack through legitimate behavior. The former occurs (in restricted form) only in applications with scripting capabilities. The latter scenario, besides requiring an cooperative victim application, hinges on the attacker's ability to scan data memory to find all needed

objects without crashing the application (hence losing the created objects) and filling the container with pointers to these.

The second restriction VPS imposes is that it prohibits overlapping objects (used for data transfer in COOP) because objects can only be created through legitimate constructors. This means that a would-be COOP attack would have to pass data via argument registers or via a scratch memory area instead. Data passing via argument registers works only if the loop gadget does not modify the argument registers between gadget invocations. Additionally, the virtual functions used as gadgets must leave their results in precisely the correct argument registers when they return. Passing data via scratch memory limits the attack to the use of virtual functions that work on memory areas. The pointer to the scratch memory area must then be passed to the virtual function gadgets either via an argument register (subject to the limitations of passing data via argument registers), or via a field in the object. To use a field in the object as a pointer to scratch memory, the attacker must overwrite that field prior to the attack, which could lead to a crash if the application tries to use the modified object.

As a third restriction, VPS's checks of the *vtblptr* at each vcall instruction mean that the attacker is limited in the virtual functions they can use at a *loop gadget*. Only the virtual function at the specific vtable offset used by the vcall is allowed; attackers cannot "shift" vtables to invoke alternative entries. This security policy is comparable to *vfGuard* [110].

To summarize, VPS restricts three crucial COOP components: object creation, data transfer, and *loop gadget* selection. Because all proof-of-concept exploits by Schuster et al. [120] rely on object overlapping as a means of transferring data, VPS successfully prevents them. Moreover, Schuster et al. recognize *vfGuard* as a significant constraint for an attacker performing a *COOP* attack. Given that VPS raises the bar even more than *vfGuard*, we argue that VPS makes practical COOP attacks infeasible.

We found that multiple of the virtual callsites missed by *VTV* (as shown in Section 4.9.1) reside in a loop in a destructor function (similar to the *main loop gadget* example used by Schuster et al. [120]). Because the loop iterates over a container of objects and uses a virtual call on each object, COOP attacks can leverage these missed callsites as a *main loop gadget* even with *VTV* enabled. This demonstrates the need for defense-in-depth, with multiple hurdles for an attacker to cross in case of inaccuracies in the analysis.

4.10.2 Limitations

At the moment, our proof-of-concept implementation of the instrumentation ignores object deletion because it does not affect the consistency of the safe memory. As a result, when an object is deleted, its old *vtblptr* is still stored in safe memory. If an attacker manages to control the memory of the deleted object, they can craft a new object that uses the same vtable as the original object. Because the *vtblptr* remains unchanged, this attack is analogous to corrupting an object's fields and does not allow the attacker to hijack control flow. Thus, while our approach does not completely prevent use-after-free attacks, it restricts them by forcing an attacker to re-use the type of the object previously stored in the attacked memory region.

Another limitation of our approach lies in the runtime verification of candidate vcall sites. If an attacker uses an unverified vcall instruction, they can force the analysis instrumentation to detect a "false-positive" vcall and remove the security instrumentation for this instruction, leaving the vcall unprotected. Because we cache analysis results, this attack only works for vcall sites that are unverified in the static analysis and have never been executed before in any run of the program, leading to a race condition between the analysis instrumentation and the attacker. The only way to mitigate this issue is by improving coverage during the dynamic profiling analysis and therefore reducing the number of unverified vcalls. This is possible by running test cases for the protected program or through techniques such as fuzzing [201, 113]. Note also that this attack requires specific knowledge of an unverified vcall instruction; if the attacker guesses wrong and attacks a known vcall, we detect the attack and log it for investigation.

VPS inherits some limitations from Dyninst, such as Dyninst's inability to instrument functions that catch or throw C++ exceptions, and Dyninst's inability to instrument functions for which it fails to reconstruct a CFG. These limitations are not fundamental to VPS and can be resolved with additional engineering effort.

Finally, we note that it is possible to enhance our safe memory region implementation, for example by using upcoming hardware features such as Memory Protection Keys (MPK) [192]. Because the safe region is merely a building block for VPS, we consider improvements to safe memory an orthogonal topic and do not explore it further in this work.

4.11 Conclusion

In this chapter, we presented VPS, a practical binary-level defense mechanism against C++ vtable hijacking attacks. Unlike prior work that restricts the target set of virtual callsites, our approach protects objects *at creation time* and restricts their usage to virtual calls that are reachable by the object. This sidesteps accuracy problems faced by prior work while simultaneously extending the threat model to include use-after-free attacks. Moreover, VPS provides improved correctness guarantees by handling false positives at vcall verification time. Our evaluation shows that VPS precisely protects applications from modern C++ code-reuse attacks, including whole-function reuse. Our analysis for detecting virtual callsites in binaries uncovered inaccuracies in *VTV*, a source-based approach that is commonly used as ground truth in this research area and broadly considered the state-of-the-art for C++-based defenses. We reported these issues to the VTV maintainers. To support future work on binary analysis (e.g., techniques for extracting higher-level programming language features from legacy binaries) and advanced mitigation techniques, we will release our analysis framework and instrumentation code.

Shared Authorship

I share authorship on VPS with Andre Pawlowski from Ruhr-University Bochum, Germany. Andre is first author of the paper that is under review and the sole author of the static analysis framework. I relied on Andre's static analysis results to (1) implement the runtime component, and (2) execute our performance evaluation.

5 The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later

In 2007, Shacham published a seminal paper on Return-Oriented Programming (ROP), the first systematic formulation of code reuse. The paper has been highly influential, profoundly shaping the way we still think about code reuse today: an attacker analyzes the "*geometry*" of victim binary code to locate gadgets and chains these to craft an exploit. This model has spurred much research, with a rapid progression of increasingly sophisticated code reuse attacks and defenses over time. After ten years, the common perception is that state-of-the-art code reuse defenses are effective in significantly raising the bar and making attacks exceedingly hard.

In this chapter, we show that an attacker going beyond "*geometry*" (static analysis) and considering the "*dynamics*" (dynamic analysis) of a victim program can easily find function call gadgets even in the presence of state-of-the-art code-reuse defenses. To support our claims, we present Newton, a runtime gadget-discovery framework based on constraint-driven dynamic taint analysis. Newton can model a broad range of defenses by mapping their properties into simple, stackable, reusable constraints, and automatically generate gadgets that comply with these constraints. Using Newton, we systematically map and compare state-of-the-art defenses, demonstrating that even simple interactions with popular server programs are adequate for finding gadgets for all state-of-the-art code-reuse defenses. We conclude with an nginx case study, which shows that a Newton-enabled attacker can craft attacks which comply with the restrictions of advanced defenses, such as CPI and context-sensitive CFI.

5.1 Introduction

Ever since the advent of Return-Oriented Programming (ROP) [124], a substantial amount of research has explored code reuse attacks in depth. Starting from a relatively simple scheme where return instructions served to link together snippets of existing code (*gadgets*), the code reuse concept was quickly generalized to include forward edges such as indirect calls and jumps [17, 120], and even signal handling [18]. Not surprisingly, defenses kept pace with the attack techniques, and a myriad of increasingly advanced attacks [25, 49, 57, 116] was met by equally advanced defenses. Some of these defenses work by constraining control transfers to a specific set of legal flows [2, 130, 131, 135], while others complicate attacks by making it difficult to find reusable code snippets [9, 10, 14, 21, 38, 39, 89, 129]. Yet other defenses protect a program by ensuring the integrity of code pointers [83, 90, 92].

In principle, exploitation may still be possible even in the presence of these defenses; for instance, through implementation issues [26, 48]. However, in practice, code-reuse attacks on a system with state-of-the-art defenses are extremely challenging. Such attacks require an attacker to analyze the protected program to find available defense-specific gadgets that can be used to implement the desired malicious payload. Crucially, the literature on code reuse attacks has thus far focused on the threat model introduced in Shacham's original work on ROP [124], which is based on (manual or automatic) *static analysis*. This is an important observation, because modern defenses reduce the set of available gadgets to the point that finding a sufficient set of gadgets for an exploit stretches the abilities of even the most advanced static analysis techniques. In this chapter, we introduce a novel approach for constructing code reuse attacks even in the presence of modern defenses. The key insight is that the required analysis effort to construct an attack can be greatly reduced and scale across a broad range of defenses by using *dynamic analysis techniques* instead of only static analysis.

Static flesh on the bone The original paper introducing Return-Oriented Programming appeared at CCS in 2007 [124], and demonstrated the first general formulation of a code-reuse attack. With ROP, an attacker would use static analysis to scan the binary for useful snippets of code that ended with a return instruction. Out of these code snippets, known as *gadgets*, the attacker would construct a malicious payload and link them by means of the return instructions at the end of the gadgets. By injecting the appropriate return addresses on the stack of a vulnerable program, an attacker could craft arbitrary functionality.

All code-reuse techniques since have followed this same basic approach—using static analysis to first identify which gadgets are available, and then constructing a malicious payload out of them. This is true even for advanced exploitation that performs such analysis "just in time" [126].

Modern code-reuse defenses push such attacks to their limits and the analysis required to bypass them is now highly sophisticated [116, 120]. In the absence of implementation errors or side channels, an attacker would be hard-pressed to locate the gadgets, let alone stitch them together. In other words, state-of-the-art defenses have been successful in raising the bar: they may not stop all possible attacks, but they make them exceedingly difficult.

Beyond static analysis The key assumption for the effectiveness of current defenses is that future attacks follow essentially the same—static analysis-based—approach as proposed by Shacham in 2007. In this chapter, we challenge this assumption, demonstrating that a switch of attack tactics to include dynamic analysis renders current defenses far less effective and attacks far less laborious. In reality, attackers do not care about gadgets or ROP chains—all they want is to execute a sensitive call such as `execve` or `mprotect` with arguments they control. There is no reason to assume that they would limit themselves to static analysis.

The goal of modern defenses is to prevent attackers from subverting a program's control flow to reach a desired target, even if an attacker is able to read or write arbitrary data. The question that the attackers must answer is which memory values they should modify to gain control over the program. Ideally, they would answer this question without resorting to complex static analysis.

The key insight in this chapter is that we can model such an attacker's capabilities by means of dynamic taint analysis. In particular, we taint all bytes that an attacker can modify with a unique color and then track the flow of taint until we reach code that, given the right values for the tainted bytes, allows the attacker to launch a code-reuse attack. For instance, if a tainted code pointer and a tainted integer later flow into an indirect call target and its argument (respectively), we have concrete evidence that the attacker can fully control a particular call instruction "gadget". Shacham's original static analysis tool is named *Galileo*, a play on its use of "*geometry*". Since our approach is largely based on dynamic ("*dynamics*") rather than static ("*geometry*") analysis, we refer to our gadget-discovery framework as NEWTON.

As we shall see, our approach requires an attacker to simply run the victim process with NEWTON's dynamic analysis enabled. Moreover, our approach can easily emulate common constraints imposed by modern defenses against code

reuse. Depending on the defense, we may be able to corrupt some locations (but not others) and target some functions (but not others). As detailed later, we can map these per-defense restrictions to simple and stackable constraints (e.g., tainting policies) for our analysis. Moreover, an attacker may model such constraints once and reuse them across a wide variety of defenses and victim applications.

Contributions We can summarize the contributions of this chapter as follows:

- We show that a hybrid static/dynamic attacker model significantly lowers the bar for mounting code-reuse attacks against state-of-the-art defenses.
- We implement NEWTON, a novel framework for generating low-effort code-reuse attacks using constraint-driven dynamic taint analysis.
- We evaluate and compare existing defenses against code reuse, highlighting their respective strengths and weaknesses using constraints in NEWTON.
- We present an nginx case study to demonstrate how to use NEWTON to craft code reuse attacks against advanced defenses, such as secure implementations of CPI [83] [1] and context-sensitive CFI [131] [2].

5.2 Threat Model

We consider a code-reuse attacker armed with arbitrary memory read and write primitives based on memory corruption vulnerabilities (e.g., CVE-2013-2028 for nginx and CVE-2014-0226 for Apache), similarly to recent work [38, 90, 100, 116, 123]. We focus on a low-effort attacker, relying on such primitives and automatic gadget-discovery tools to craft attacks with limited application knowledge. Our attacker seeks to locate gadgets and mount code-reuse attacks, even in the face of state-of-the-art defenses such as Control-Flow Integrity (CFI) [130, 131, 135], leakage-resistant code randomization [21, 38], and Code-Pointer Integrity (CPI) [83]. We focus specifically on lightweight code-reuse defenses and leave

[1] We focus on the published implementation of CPI, which, as we verified, features no temporal checks and no read-side bounds checks. The authors of CPI informed us that the latter, which we had assumed to be an optimization, is really an implementation bug. Unfortunately, adding such expensive bounds and temporal checks to approximate full memory safety will non-trivially increase the CPI overhead. For this reason, we consider the efficient published implementation a more interesting and concrete design point to analyze as of today.

[2] We consider context-sensitive CFI (CsCFI) and context-insensitive CFI separately and, with CsCFI, we exclusively refer to stateful CFI policies based on execution history.

more general heavyweight defenses such as memory safety [93, 94] or Multi-Variant Execution (MVX) [78, 168] out of scope.

Given the overwhelming number of code-reuse defenses in the literature, we limit our analysis to only (1) defenses applicable to general programs (e.g., no vtable protection for C++ programs [130]), (2) the strongest designs in each class (i.e., effectiveness against weaker defenses is implied), and (3) the secure implementation of such designs (e.g., no side-channel [48, 100] or weak-context [26] bypasses). We also assume a strong baseline with ASLR [209], DEP [185], a perfect shadow stack [41] (making it impossible to divert control-flow by modifying return addresses), and coarse-grained forward-edge CFI [157] (callsites can only target function entry points) enabled.

We assume that the attacker has access to a binary equivalent to the one deployed by his prospective victim. Finally, for simplicity, we focus specifically on popular server programs, similar to much prior work in the area [15, 90, 99, 100, 116, 123, 131, 135].

5.3 Overview of Code-Reuse Defenses

In this section, we provide an overview of state-of-the-art code-reuse defenses considered in our threat model. We distinguish four classes of code-reuse defenses: (1) Control-Flow Integrity, (2) Information Hiding, (3) Re-randomization, and (4) Pointer Integrity. We now introduce each of these classes in turn, and later show how to map them to NEWTON constraints in Section 5.5.

Control-flow integrity (forward-edge) Control-Flow Integrity (CFI) mitigates code-reuse attacks by instrumenting indirect callsites to ensure that only legal targets allowed by the (inter-procedural) Control Flow Graph (CFG) of the program are permitted [2]. To determine the targets for each callsite, modern CFI solutions use either static or dynamic information.

CFI solutions that rely only on static information either allow callsites to target all function entry points [154, 157] or, more recently, construct the set of legal targets by mapping callsite types to target function types. In other words, a callsite of the form `foo(struct bar *p)` should only call functions of type `func(struct bar *p)`. In particular, IFCC [130] and MCFI [97] construct such mappings using source type information, while TypeArmor [135] approximates types based on argument count at the binary level.

CFI solutions that rely on dynamic information track execution state to improve the accuracy of static analysis. In particular, PICFI [99] implements a

"*history-based CFI*" (*HCFI*) policy, restricting the target set to function targets whose address has been computed at runtime. Context-sensitive CFI (*CsCFI*) solutions (or similar, with different definitions of "context") such as PathArmor [131], GRIFFIN [52], FlowGuard [88], kBouncer [102], and ROPecker [30] restrict the target set based on analysis of the last n branches recorded by hardware, e.g., the Last Branch Record (LBR) registers or Intel PT. The effectiveness depends on the amount of useful "context" in the branch history, which is necessarily limited in practical implementations: 16 or 32 LBR entries [30, 102, 131], 30 Intel PT packets [88], or a limited policy matrix [52].

Information hiding Information hiding (IH) aims to prevent code reuse by making the locations of gadgets unknown to an attacker. This is done by (1) diversifying the code layout using traditional Address-Space Layout Randomization (ASLR) [209] or more fine-grained variants [10, 12, 13, 21, 28, 164, 37–39, 54, 56, 62, 63, 75, 81, 101, 128, 142] and (2) "hiding" code pointers to an arbitrary memory read-enabled attacker. The latter property is enforced in different ways by different leakage-resistant randomization solutions.

Oxymoron [10] removes all the code references from the code, preventing an attacker reading any given code page from gathering new code pointers that reveal the location of other code pages. Other solutions such as Readactor [38], software-based XnR [9], HideM [55], LR^2 [21], KHide [54], kR^X [109], Heisenbyte [129], and NEAR [143] implement eXecute-Only Memory (*XoM*) or similar semantics for code pages, preventing an attacker from reading useful gadgets from the code and thus fully "hiding" the code layout (in the ideal case). Finally, recent solutions such as Readactor++ [39] and CodeArmor [28] extend XoM semantics (*XoM++*) to also hide code pointer tables such as the Global Offset Table (GOT).

Re-randomization Re-randomization (RR) is another popular defense strategy against code reuse attacks. Unlike information hiding, re-randomization solutions seek to re-randomize and invalidate leaked information (ideally) before the attacker has a chance to use it and craft just-in-time code reuse attacks [126]. Existing solutions can be classified based on the particular information they periodically re-randomize during the execution.

Some RR solutions such as Shuffler [144], CodeArmor [28], and ReRanz [140] periodically re-randomize the code layout (*CodeRR*) but leave the function pointer values stored in data pages (heap, stack, etc.) immutable using indirection tables. In contrast, TASR [14] re-randomizes each code pointer value in memory

every time the corresponding code target is re-randomized. Finally, other solutions such as ASR_3 [56] and RuntimeASLR [89] re-randomize the full memory address space layout, including the values of code and data pointers at each re-randomization period.

Pointer integrity Pointer integrity (PI) solutions seek to counter code reuse by preventing attackers from tampering with code or data pointers. Existing solutions can be classified as encryption-based or isolation-based.

ASLRguard [90] is an encryption-based solution that encrypts each computed code pointer with a per-pointer key in a safe vault, (ideally) preventing attackers from crafting new code pointers in memory. In contrast, CCFI [92] encrypts each code pointer stored into a given memory address with an address-dependent key, also preventing attackers from reusing leaked code pointers in memory.

CPS [83] is an isolation-based solution that isolates all the code pointers in a protected safe region, (ideally) preventing an attacker from reaching and corrupting any of these pointers. CPI [83] extends CPS to also isolate data pointers that may indirectly be used by the program to access code pointers, (ideally) preventing an attacker from corrupting code and related data pointers in memory.

5.4 Overview of Newton

We now present Newton, our gadget-discovery framework to assist in crafting code-reuse attacks against arbitrary (modeled) defenses. For this purpose, Newton applies a uniform and blackbox strategy to dynamically retrieve gadgets as a set of *attacker-controllable forward CFG edges*. Each edge is expressed as a call-site with a number of possible target functions, and tagged with a number of *dependencies* (e.g., the target function is controlled by the code pointer stored at address X and the first argument is controlled by address Y). These edges can then be inspected by an attacker and used to call arbitrary functions via arbitrary memory read/write primitives. To call a sequence of arbitrary functions, an attacker can chain a number of such edges together over multiple interactions with the victim application.

To support a range of code-reuse defenses, Newton accepts user-defined constraints that limit the analysis to only gadgets allowed by the given modeled defense. The idea is to run the victim program mimicking the stages of the real attack and constrain Newton's dynamic gadget analysis using simple, reusable, and extensible policies that map the security invariants of a broad range of defenses. We discuss the mapping of defenses to constraints later, in Section 5.5.

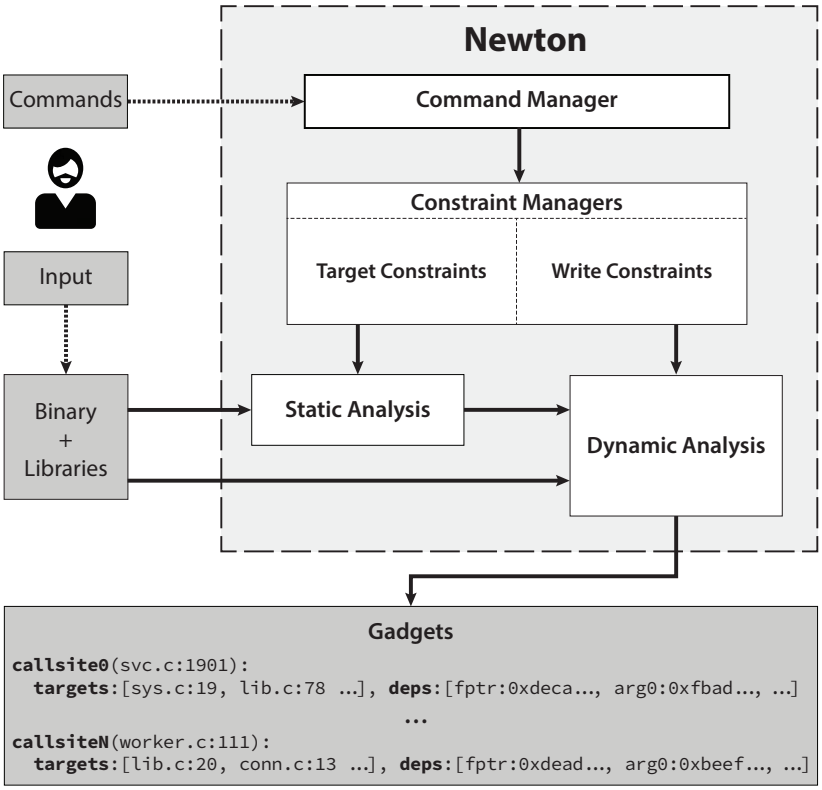


Figure 5.1. Design of NEWTON.

Figure 5.1 presents an overview of NEWTON and its high-level components. The NEWTON framework pushes the victim binary and its shared libraries through a pipeline of (1) static analysis, and (2) dynamic analysis—on top of a *dynamic taint analysis (DTA) engine*. During both phases, the *target* and *write* constraint managers apply user-defined constraints to the analysis, eventually yielding a list of callsites an attacker can control and, for each callsite, a list of callees an attacker can target under a given defense (or combination of defenses) regime.

In more detail, the workflow of NEWTON when analyzing a binary to craft a code reuse attack is as follows.

1. At the start of the analysis process, the user starts the target application binary normally. At this point, NEWTON is in a waiting state, and does not yet perform any analysis.

2. The user now brings the application into a stable state where they can effect arbitrary memory read/write primitives. In our evaluation, we assume

that the user brings the victim program into a simple *quiescent state*. For instance, in the case of a server application, the user would perform a minimal set of interactions to bring the server into an idle state with an open connection, where only long-lived data persists in memory, as in [100]. In general, the chosen quiescent state is program-dependent.

3. Next, the user signals NEWTON that the victim application is now in a quiescent state. At this point, NEWTON begins tracking user-controlled memory dependencies using its DTA engine.

4. At the same time, the user supplies NEWTON with a number of commands (in a script) to specify the target and write constraints that NEWTON should assume are used to defend the victim application. As a result, NEWTON will take these constraints into account during its analysis of controllable edges.

5. The user now interacts with the victim application, using the inputs they want to use during the final exploit. This allows NEWTON to track the dependencies during these interactions. Focusing on a low-effort attacker targeting a server application, we assume that the interactions amount to simple standard requests to the victim server.

6. Finally, NEWTON reports the results of its analysis. This yields a set of gadgets (callsites+targets) that are under the user's control given the user's chosen defense model, initial quiescent state and set of server interactions.

5.4.1 Constraints

As defined by our threat model, our goal as an attacker is to use an arbitrary memory read/write primitive to divert control flow. The baseline defenses described in Section 5.2 force us to achieve this by corrupting memory in such a way that later in the execution, the target of an indirect callsite no longer points to its intended callee. With this in mind we observe that, conceptually, all existing defenses attempt to avert successful attacks by enforcing constraints along one (or both) of the following two dimensions:

1. **Write constraints.** Write constraints limit an attacker's capability to corrupt writable memory. Without any defense deployed, an attacker can corrupt anything: (1) pointers to code (function pointers), (2) pointers to data, and (3) non-pointer values such as integers or strings.

2. **Target constraints.** Constraints on targets limit the attacker in his selection for possible callees of a controlled callsite. Without any target con-

straints beyond the baseline, the target set always consists of all functions in the program and library code. We show later how different defenses and their constraints reduce the wiggle room for an attacker.

5.4.2 Write Constraint Manager

The write constraint manager accepts user-defined constraints, describing the memory regions the attacker is allowed to overwrite under the modeled defense. Then, using *constraint-driven dynamic taint analysis*, it pinpoints callsites and arguments which can still be controlled by the attacker, despite the assumed defenses. NEWTON's DTA engine is a heavily modified version of LIBDFT [74] which supports arbitrary tags per memory location, as well as additional functionality to support the command manager API (see Section 5.4.4). The steps of the analysis are as following:

1. **Initial tainting.** We model attacker-controlled memory by initially marking regions under the attacker's control as tainted. To easily model different defenses, NEWTON exposes taint limiting commands that allow control over how the initial taint is applied (see Section 5.4.4). NEWTON's DTA engine propagates the taint information to callsites and arguments.

2. **Tracking dependencies.** We configure our taint engine with a unique tag for each byte in memory, allowing us to track attacker-controlled memory dependencies at byte granularity. Our dynamic taint analysis engine is capable of tracking the taint source address for each tainted value or pointer in memory. For each tainted byte, this tells us exactly by which memory addresses it was tainted. This allows us to track, when a tainted callsite is discovered, where the taint originated for the associated function pointer and each of the arguments. The source of the taint is then a candidate value for the attacker to corrupt, to control the callsite and mount a code-reuse attack.

   LIBDFT's original implementation implements a basic taint strategy [74], able to track only direct attacker-controlled memory dependencies (i.e., callsite X uses code pointer at tainted address Y) and not indirect ones (i.e., callsite X' uses code pointer read via data pointer at tainted address Y'). To support the latter, we implemented pointer tainting for memory reads in LIBDFT [74] (i.e., taint every value read via a tainted pointer), allowing us to model an attacker corrupting data pointers and non-pointers to indirectly control code pointers (and arguments) used by tainted callsites.

3. **Logging.** When an indirect call is executed, NEWTON logs the relevant taint information for this callsite. Specifically, for each tainted callsite, we emit information detailing the taint dependencies for the callsite's target, and the first six arguments.

5.4.3 Target Constraint Manager

Like the write constraint manager, the target constraint manager models constraints imposed by code reuse defenses. It uses static and dynamic analysis to extract callsite and callee information, which it then uses to impose the user-defined constraint policy.

Static analysis We use a static analysis based on DWARF debugging symbols to extract all callsites and potential callees from the target binary and shared libraries, along with associated type information. NEWTON uses the extracted information (if instructed) to simulate a number of policies for existing defenses, such as type-based CFI [97, 130, 135].

Dynamic analysis In addition to the aforementioned static analysis, we also use dynamic analysis to scan user-defined ranges of writable memory (such as `.data`, or the heap) for code pointers. We define a *live code page* as a memory page pointed to by a *live code pointer*, i.e., a code pointer stored in live data memory that can be leaked and overwritten. Our dynamic analysis allows us to track live code pointers and code pages. We use this information to model target constraints imposed by defenses such as Readactor [38], which limit an attacker to using "live" gadgets in memory.

The target constraint manager logs the valid targets for each callsite based on the constraints derived by the static and dynamic analysis, as guided by the user-defined script modeling the defense.

5.4.4 Command Manager

As mentioned, NEWTON includes write constraint and target constraint managers which model the constraints imposed by a particular defense, based on a user-defined script. To handle the scripting commands, NEWTON includes a *command manager*. The command manager is a preloaded library that loads along with the analyzed binary, and listens for commands on a Unix domain socket. When a command is received, the command manager dispatches it to the right constraint manager, which handles it as needed.

NEWTON exposes the following command functions, sufficient to map all of the defenses we evaluate in Section 5.6. In Section 5.5, we show examples of these commands used in practice to model defenses.

- `taint-mem`: This command instructs the taint analysis engine to mark all writable memory as tainted, simulating the arbitrary read/write primitive we assume in our threat model (see Section 5.2). It initializes the source taint for each value to its own address. In Section 5.5, we show how among other things, we use `taint-mem` to taint all memory after bringing a victim server program into a quiescent state.
- `taint-flip`: This command untaints all tainted data, and taints all untainted data. We use the ability to flip taint when crafting history-flushing attacks against context-sensitive CFI defenses, as explained further in Section 5.5.
- `taint-prop-toggle`: This command pauses or resumes the propagation of taint (also implies `taint-log-toggle`) by NEWTON's DTA engine. *Default*: on.
- `taint-log-toggle`: Similar to `taint-prop-toggle`, this command pauses or resumes the logging of tainted callsites. This is used to avoid logging uninteresting callsites. Taint propagation continues normally. *Default*: on.
- `taint-ptr-toggle`: This command enables or disables pointer tainting on memory reads. *Default*: on.
- `taint-wash (CPtr|Ptr|AddressRange)`: This command clears the taint for particular memory locations: locations with code pointers, data pointers, or in a given address range.
- `constrain-targets`: This command specifies target constraints to enforce on tainted callsites.
- `get-gadgets`: This command retrieves all gadgets collected during the execution.

5.5 Mapping Defenses

As mentioned in Section 5.4, for the purpose of finding gadgets for code reuse with NEWTON, we model the security provided by code-reuse defenses along two axes: (1) write constraints imposed by the defense, and (2) the imposed target

Table 5.1. Mapping of code-reuse defenses to NEWTON constraints. Empty entries for write/target constraints indicate that the defense imposes no write/target constraints, respectively.

Defense			Write constr.	Target constr.	
Class	*Subclass*	*Solutions*	*Details*	*Details*	*Dynamic*
CFI	TypeArmor	[135]			Bin types
	Safe IFCC/MCFI	[97, 130]			Safe src types
	IFCC/MCFI	[97, 130]			Src types
	HCFI	[99]		✓	Computed
	CsCFI	[30, 52, 60, 88, 102, 131]	Segr		
IH	Oxymoron	[10]		✓	Live +page
	XoM	[9, 21, 38, 54, 55, 109, 129, 143]		✓	Live
	XoM++	[28, 39]		✓	Live ¬GOT
RR	CodeRR	[28, 140, 144]		✓	Live
	TASR	[14]	¬CPtr	✓	Live
	PtrRR	[56, 89]	¬Ptr	✓	Live
PI	ASLR-Guard	[90]		✓	Live
	CCFI/CPS	[83, 92]	¬CPtr	✓	Live
	CPI	[83]	¬Ptr	✓	Live

constraints. In this section, we map the defenses from Section 5.3 according to these constraints. This mapping allows us to easily create scripts that teach NEWTON about the constraints (security restrictions) imposed when searching for attacker-controllable gadgets (callsites and possible targets).

5.5.1 Deriving Constraints

Table 5.1 summarizes the constraints imposed by each defense class. We now discuss each class in detail.

Control-flow integrity We distinguish five subclasses within the CFI class of defenses: (1) TypeArmor, (2) IFCC/MCFI, (3) Safe IFCC/MCFI, (4) HCFI, and (5) CsCFI.

TypeArmor imposes target constraints which enforce that call sites can only target functions with a type matching the call site's type; such types are approximated by statically analyzing the program binary (*Bin types*). Since TypeArmor is the only defense which offers function type-based CFI at the binary level, it has its own dedicated subclass in Table 5.1.

The IFCC/MCFI subclass provides similar constraints as the TypeArmor subclass, except that function type information is computed at the source rather than at the binary level. This leads to a stronger target constraint (*Src types*) and hence security. This is because source information allows IFCC/MCFI to compute more accurate type information and derive a smaller legal target set.

Safe IFCC/MCFI comprises the same defenses as the IFCC/MCFI subclass, except that in this case the defenses run in a "safe" mode, where type information is less strict for compatibility reasons with real-world programs—discussed in the original IFCC paper [130]. For instance, in this mode, pointer parameters such as `int*` or `void*` are each considered to be interchangeable with other pointer types. This leads to a weaker target constraint (*Safe src types*) compared to the non-safe variant of this subclass.

In the HCFI (history-based CFI) subclass, the set of valid targets for each call site is determined by the set of code pointers that have been computed during the execution. This is a dynamic target constraint (*Computed*), which can be used in isolation or combined with other static target constraints.

All the CFI subclasses thus far have been modeled using target constraints. Somewhat counter-intuitively, we model the CsCFI subclass using *only* write constraints. The reason is that this makes it much easier to write a NEWTON CsCFI-aware script for a low-effort attacker. Formulating CsCFI in terms of target constraints would require us to provide NEWTON with knowledge about the context-sensitive analysis, the branch history size, and the time of validation (e.g., syscall time). Furthermore, when assuming a "perfect" (but practical) implementation of CsCFI, the branch history can be arbitrarily large (but not unlimited), allowing a "perfect" context-sensitive analysis to always detect invalid targets in the large context provided. In other words, the only way for an attacker to bypass the defense is to force the application to flush the (arbitrarily large) branch history [26] before triggering the exploit. This leaves CsCFI with no context to constrain the controlled target set.

For this purpose, the attacker needs to (1) corrupt some *segregated* (independent and stable) application state, (2) send an arbitrarily large number of idempotent *history-flushing inputs* to the application that do not interfere with the segregated state, (3) send the final input to trigger the exploit based on the segregated state. This translates to a write constraint (*Segr*) that limits writes to the segregated state specified by the attacker. At first glance, identifying such state and the history-flushing input seems complicated. In practice, this is possible even for a low-effort attacker. For example, for common server applications that handle multiple connections in a single worker process (e.g., nginx), we can sim-

ply instruct NEWTON to use the connection-specific data of a first connection as segregated state and a simple request over a second connection as the history-flushing input (as done in Section 5.5.2).

Information hiding We distinguish three subclasses within the IH class of defenses: (1) Oxymoron, (2) XoM, and (3) XoM++.

The Oxymoron subclass allows only targets contained in *live code pages*. This translates to a target constraint (*Live +page*) that limits the set of valid (i.e., leaked by an attacker) targets to pages pointed to by live code pointers.

The XoM subclass contains defenses that hide the code layout from an attacker. This translates to a target constraint (*Live*) that limits the set of valid targets to live code pointers (again stored and then leaked from memory), given that the attacker can make no assumptions on the other code pointers.

Finally, defenses in the XoM++ subclass implement XoM semantics and additionally hide the GOT from an attacker. This translates to a stronger target constraint (*Live* $\neg$*GOT*) than XoM's, where live code pointers in the GOT are no longer valid. Since the GOT itself is no longer reachable and thus not corruptible, this also translates to a write constraint ($\neg$*GOT*), which, for simplicity, we leave implicit in our analysis and presentation of the results (its impact typically aligns with its target constraint counterpart).

Re-randomization We distinguish three subclasses within the RR class of defenses: (1) CodeRR, (2) TASR, and (3) PtrRR. Since all these subclasses hide the code layout under ideal conditions, they all impose a target constraint that allows only live code pointers to be used as valid targets (*Live*). However, the subclasses differ in terms of their write constraints.

First, the CodeRR subclass only hides (i.e., re-randomizes) the code layout and imposes no additional write constraints. The second RR subclass, TASR, does impose an additional write constraint. Not only does TASR periodically re-randomize the code layout, but it also re-randomizes the code pointer representation (even for code pointers stored in data memory). In doing so, it prevents attackers from successfully overwriting code pointers. This translates to a write constraint ($\neg$*CPtr*) that forbids writes to memory locations containing code pointers. In other words, this constraint teaches NEWTON that the only way to find gadgets that bypass CodeRR is to corrupt data pointers (or non-pointers) to force the program to access an attacker-controlled live code pointer rather than the original intended target (e.g., corrupting `c` to hijack `c->handler()`).

Finally, the PtrRR subclass is similar to TASR, except that the imposed write

constraint is stronger. Not only code pointers but *all* pointers are re-randomized and thus "protected" against overwrites. This translates to a write constraint ($\neg$*Ptr*) that forbids writes to memory locations containing either code or data pointers. In other words, this constraint teaches NEWTON that the only way to find gadgets that bypass PtrRR is to corrupt non-pointers such as integers (e.g., corrupting `idx` to hijack `func[idx]->handler()`).

Pointer integrity We distinguish three subclasses within the PI class of defenses: (1) ASLR-Guard, (2) CCFI/CPS, and (3) CPI. All three of these prevent an attacker from crafting new code pointers from scratch, thus enforcing a target constraint that limits targets to live code pointers (*Live*).

ASLR-Guard does not impose any additional constraints. It implements the aforementioned target constraint by using per-pointer secret keys to encrypt all code pointers. Thus, while an attacker cannot introduce new code pointers, it is still possible to replace a code pointer with another arbitrary live code pointer, given that the secret key is not location-aware.

The second PI subclass, CCFI/CPS, does impose an additional write constraint that forbids writes to memory locations containing code pointers ($\neg$*CPtr*). In the case of CCFI (Cryptographically-enhanced CFI), this is implemented by encrypting pointers with a memory location-dependent key. In the case of CPS, the same effect is achieved by isolating code pointers in a memory region not accessible to an attacker.

Finally, CPI is equivalent to CPS, except that it isolates not only code pointers, but also data pointers that point to structures containing code pointers. Thus, CPI imposes a stronger write constraint than CPS, forbidding writes to memory locations containing either code or data pointers ($\neg$*Ptr*).

5.5.2 Implementation

Figure 5.2 graphically depicts the constraints imposed by the defenses, as detailed in Table 5.1. The x-axis shows the write constraints imposed by each defense subclass, while the y-axis shows the target constraints. Defenses that share both the same write and target constraints impose equivalent security restrictions, so that each (x, y) point in Figure 5.2 forms an *equivalence class*.

It is interesting to note that even defenses that seem quite different on the surface actually turn out to offer comparable guarantees. For instance, the figure reveals the following equivalence classes that contain multiple defense techniques each: $\{XoM, CodeRR, ASLR-Guard\}$, $\{TASR, CCFI/CPS\}$, and $\{PtrRR, CPI\}$. Note that these equivalences hold only when assuming

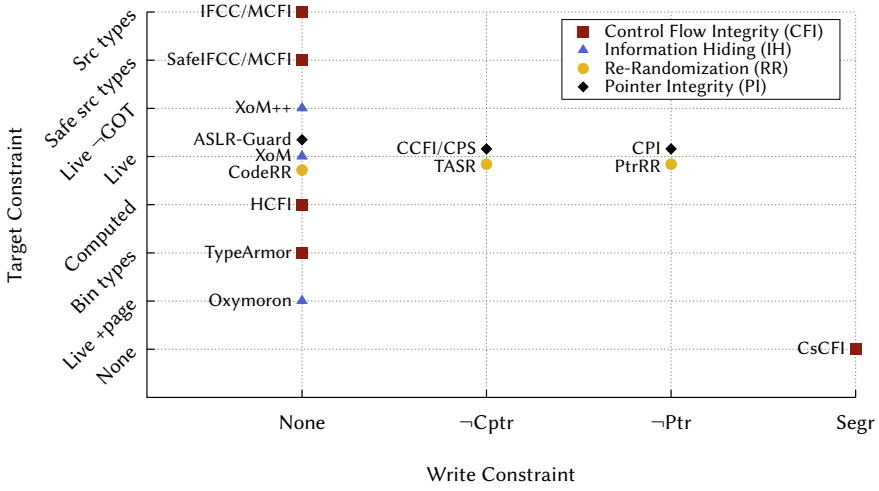


Figure 5.2. Mapping of defense classes to write (x-axis) and target (y-axis) constraints in NEWTON. Constraints on the two axes are sorted based on their effectiveness in reducing the number of gadgets available to a low-effort attacker on nginx, when sending a plain `HTTP GET` request.

"perfect" implementations of each defense, without any implementation-specific vulnerabilities. In addition, our constraint-based classification abstracts away implementation details and hence ignores implementation-specific differences across defenses. For instance, the ¬Ptr constraint in RuntimeASLR protects all data pointers, and could thus be considered stronger than the same constraint in CPI, which protects only data pointers that can be used to read code pointers. The key advantage of our approach is that it allows us to focus on the general constraints for gadget generation across many different defenses.

We now demonstrate how to concretely implement constraints for the mapped defenses in NEWTON, using the commands detailed in Section 5.4. We organize the following discussion around the write constraints imposed by each defense.

Corrupting code pointers All defense subclasses that do not implement write constraints allow any memory to be corrupted, including code pointers. These defenses are on the left of the x-axis in Figure 5.2 (*None*). To model these, we use the NEWTON script from Figure 5.4a. All our example scripts assume a low-effort attacker attacking a server application. After starting the server, the script first informs NEWTON about any target constraints; this guides NEWTON's static and dynamic analysis of callees and live code pointers. NEWTON has internal support for each of the possible target constraints shown in Table 5.1 and Figure 5.2.

```
$ start server
constrain-targets $Cons
$ C1 = open connection
taint-mem
$ send request over C1
get-gadgets
```

(a) No write constraints.

```
$ start server
constrain-targets $Cons
$ C1 = open connection
taint-mem
taint-wash CPtr
$ send request over C1
get-gadgets
```

(b) ¬CPtr.

```
$ start server
constrain-targets $Cons
$ C1 = open connection
taint-mem
taint-wash Ptr
$ send request over C1
get-gadgets
```

(c) ¬Ptr.

```
$ start server
constrain-targets $Cons
taint-prop-toggle off
taint-mem
$ C1 = open connection
taint-flip
$ C2 = open connection
$ send N requests over C2
taint-prop-toggle on
$ send request over C1
get-gadgets
```

(d) Segr.

Figure 5.3. NEWTON command scripts for finding gadgets under different modeled write constraints.

Next, the script taints all memory using the `taint-mem` command. We then send a normal request to the server, causing NEWTON to track any taint propagated during this request. As the request is processed, NEWTON logs tainted callsites, their arguments, dependencies, and potential targets. These gadgets can then later be retrieved by the user (`get-gadgets` command).

Corrupting data pointers Defense subclasses with the ¬CPtr write constraint prevent code pointers from being overwritten, but do not protect other memory locations. This includes the CCFI/CPS and TASR subclasses. As a result, under these defenses, it is still possible to corrupt data pointers and non-pointers.

We model these defenses in NEWTON using the script shown in Figure 5.4b. The script is identical to the script we used to model defenses without any write constraints, except that after tainting all memory, we use the `taint-wash` command to untaint code pointers. This has the result of simulating that code pointers are not overwritable by an attacker, thus modeling defenses in the ¬CPtr write constraint class.

Corrupting non-pointers Under defenses that implement the ¬Ptr write constraint, neither code nor data pointers can be written, limiting the attacker to overwriting only non-pointers. We simulate this using the script shown in Figure 5.4c, in which we clear the taint for both code and data pointers after tainting memory.

Corrupting segregated state As mentioned in Section 5.5.1, we model the CsCFI subclass using write constraints instead of target constraints, as this makes CsCFI easier to emulate in NEWTON. As described earlier, the write constraints impose a "segregated memory" defense model, in which an attacker corrupts program state in such a way that this state is not modified by subsequent history-flushing requests. The attacker then uses an arbitrary number of these requests to flush the context of the CsCFI defense, after which it becomes possible to use the previously corrupted state to trigger an exploit.

We model this in NEWTON using the script shown in Figure 5.4d. The script begins by starting the victim server and setting the target constraints, as usual. Next, we disable taint propagation, after which we taint all memory and open an attack connection (c_1), and finally flip the taint state of all memory. Opening the connection has the effect of clearing taint on the memory touched by the connection state. Thus, when we flip the taint state, the untainted memory (containing the connection state) becomes tainted, while all other memory becomes untainted. This way, we model the initial segregated (connection) state, which will serve as the attack surface in the final exploit. Note that the segregated state is not an idle state as our attack connection is still open, and that there are possibly many more active open connections in parallel.

We now send an arbitrary number of idempotent requests to the server over an independent history-flushing connection c_2. This is to model flushing the CsCFI context and also ensure there is no interference with the state of connection c_2. Finally, we re-enable taint propagation, resume the attack connection c_1 (left open previously), and send the final request. The final result of the analysis is a list of callsites (with possible targets and dependencies) which are tainted *only* by attacker-controlled connection-specific state, and are thus controllable by the attacker after the history-flushing attack is complete. This voids the concern that some of the long-lived structures in the quiescent state may be modified by parallel connections.

5.6 Evaluation

We evaluate Newton against three web servers (nginx, Apache's httpd, and lighttpd), a general-purpose distributed memory cache system (Memcached), an in-memory database (Redis), and a domain name system (BIND). As is common these days, we compile the servers as position independent code, using gcc as our compiler.

Using Newton scripts as presented in Section 5.5.2, we instruct our target constraint manager to apply each of the target-based policies from Section 5.5 (in addition to the baseline as described in Section 5.2). As described there, we divide the deployed defenses into those with static target constraints, and dynamic ones.

Also recall from Section 5.5.2 that our scripts instruct the write constraint manager to apply the following types of write constraints: (1) None, this is our baseline where an attacker can corrupt anything, including code pointers; (2) ¬CPtr, policies that restrict the corruption of code pointers; (3) ¬Ptr, policies that enforce pointer integrity; and (4) Segr, for context-sensitive CFI.

We first perform a detailed evaluation for nginx, in which we provide statistics on the controllability of each executed indirect callsite. Later, in Section 5.7, we show how to use this information to mount defense-aware attacks against nginx. In the second part of this evaluation, we provide summarized results for all tested servers, to illustrate the wide applicability of our attack methodology.

Note that we do not evaluate the expressiveness of code-reuse attacks based on Newton, i.e., we do not study whether Newton can produce Turing-complete attacks. The motivation behind this is that Turing-completeness neither guarantees nor is a prerequisite for successful exploitation and as such does not affect the applicability of Newton: an attacker is unlikely to care about finding all Turing-complete gadgets if only one or two already provide him with enough means to gain arbitrary code execution. We consider a study in which existing defenses are evaluated with respect to whether they prevent Turing-complete ROP attacks as an interesting starting point for future work.

Although our evaluation focuses on popular system services, the principles of Newton also apply to user applications like browsers, document readers, and word processors. The large memory footprint of such applications, however, means that our libdft-based DTA engine (which is 32-bit only) quickly runs out of memory. This limitation is not fundamental to Newton, and can be addressed in future work with additional engineering effort (i.e., porting libdft to x86_64).

5.6.1 In-Depth Analysis of nginx

We now evaluate the controllability of each executed indirect callsite in nginx, under all types of write and target constraints. We first examine the residual attack surface per target constraint, and then do the same for each write constraint.

Target constraints Table 5.2 depicts the residual attack surface in nginx under different target constraints. Note that the numbers shown for dynamic target constraints are susceptible to the coverage of our dynamic analysis. As mentioned, we assume a low-effort attacker; thus, the numbers shown in Table 5.2a and 5.2b cover the case where the attacker sends only a simple `GET` request to nginx. It is conceivable that a more determined attacker could uncover even more attack surface than shown here.

Also note that we show absolute numbers for dynamic constraints, but median results for static constraints. This is because static target constraints limit the number of targets *per callsite*, while dynamic constraints limit the total number of legal pointers *in memory*.

To interpret the tables, we look at one example row from each table. We begin with an example from Table 5.2a. Consider the *Computed* target constraint, which is used by the *HCFI* defense subclass, implemented by Per-Input CFI [99]. Under this constraint, only code pointers which have been computed during program execution can be used by an attacker. Table 5.2a shows that after server initialization and handling of the `GET` request, 786 such pointers reside in memory. Thus, each indirect callsite may target each of these. Continuing the *Computed* target constraint example, Table 5.2b shows that of al computed pointers, 1 was stored on the stack, and 64 on the heap. The remaining originate from the loaded modules: 270 from nginx' data sections (`.data`, `.data.rel.ro`, or `.rodata`), 32 from its global offset tables (`.got`, `.got.plt`), and 25 pointers were found in the remaining sections and other modules.

To explain Table 5.2c, we consider the *Safe src types* constraint, imposed by the *SafeIFCC/MCFI* defense subclass, which provides type-based caller/callee matching. In this, the median indirect callsite is allowed to target 176 libc functions, and 376 functions in total. Each callsite may target at least 2 functions in nginx, while 90% of the callsites may target 69 functions in modules other than nginx or libc.

Overall, the main takeaway from Table 5.2 is the ease with which our methodology allows us to compare the strength of even extremely different defense subclasses. For instance, it becomes clear that the strongest dynamic target constraint is *Live* $\neg$*GOT*, imposed by the *XoM++* defense subclass. Comparing Ta-

Table 5.2. Number of permissible targets in nginx under each target constraint policy.

(a) Absolute number of legal function targets found in the main nginx module, libc, other modules, and in total, respectively, when applying dynamic target constraints.

Dynamic target constraint	Targets			
	nginx	libc	other	total
None	1,035	2,763	794	4,592
Live +page	811	1,264	411	2,486
Computed	363	323	100	786
Live	362	316	89	767
Live ¬GOT	360	279	69	708

(b) Location of the code pointers to legal targets when applying dynamic target constraints, binned to the stack, heap, or `.data`/`.got`/other segments in a particular module.

Dynamic target constraint	Target location										
	stack	heap	nginx			libc			other		
			data	*got*	*other*	*data*	*got*	*other*	*data*	*got*	*other*
Live +page	15	475	261	399	81	666	26	67	207	257	32
Computed	1	64	270	32	25	240	2	42	65	38	7
Live	1	64	269	31	25	237	2	41	60	31	6
Live ¬GOT	1	64	269	0	25	237	0	41	60	0	6

(c) Number of legal function targets per callsite and their distribution when applying static target constraints. The **Target (median)** group shows the median number of legal function targets found in the main nginx module, libc, other modules, and in total, respectively. The **Target distribution** group shows the minimum and 90th percentile number of targets pointing to each module, per callsite.

Static target constraint	Targets (median)				Target distribution					
	nginx	libc	other	total	nginx		libc		other	
					min	*90p*	*min*	*90p*	*min*	*90p*
Bin types	328	960	370	1,665	201	758	549	1,625	203	437
Safe src types	117	176	65	376	2	135	0	230	0	69
Src types	12	0	0	19	1	58	0	0	0	0
Source	12	0	0	19	1	57	0	0	0	0

ble 5.2a and 5.2c, it is also clear that static type-based constraints are in general stronger than dynamic ones, with the strongest target constraints being imposed by source-level type-based defenses. It is also worth noting that even for the strongest target constraints, there is still a significant residual attack surface.

Write constraints We now consider the potential controllability of callsites in nginx given varying write constraints. Moreover, we also show that for each executed callsite, a nontrivial attack surface remains even under the strongest combinations of write and target constraints. To obtain information on which callsites are potentially controllable, we examine the taint information which NEWTON yields during the aforementioned attacker-initiated `GET` request to nginx. We present these results in Table 5.3 on page 140.

To illustrate the semantics of Table 5.3, consider callsite number 27, at location `http_request.c:1126`. The target (*function pointer*) of this callsite is tainted by a code pointer, meaning that it can be controlled under write constraints which allow corrupting code pointers. Moreover, it is controllable from segregated state, i.e., the callsite is usable in a history flushing-based attack against *CsCFI*. All three arguments are tainted by non-pointer values, making them controllable even under the strictest write constraints. Controlling three arguments is often sufficient; for instance, both `execve` and `mprotect` take only three arguments (and `system` takes one).

Table 5.2a shows that, without any additional target constraints, the callsite at location `http_request.c:1126` has 4,592 legal targets. Imposing the strongest dynamic target constraint (*Live* $\neg$*GOT*) reduces this to 708 targets, while the strongest static target constraint (*Source types*) allows only 3 targets; the same number of targets as is allowed under the combination of these write constraints.

Note in Table 5.3 that 13 of the 35 callsites have a target that is tainted by a non-code pointer value, making them controllable even when code pointers are protected. Moreover, 8 callsites have a target tainted by a non-pointer value, making these callsites controllable under *all* write constraints imposed by current defenses. Many of these callsites have a significant number of legal targets, ranging up to 49 targets even when combining the strongest static and dynamic target constraints.

5.6.2 Generalized Results

Table 5.4 on page 141 shows that nginx is representative for all evaluated servers. The fraction of tainted callsites is comparable, with the exception that callsites in httpd are not controllable using segregated state; httpd creates a new process for each connection, preventing our history flushing attack. In all evaluated servers, attacker-controlled callsites remain even under $\neg$Ptr write constraints.

Moreover, in all servers, a significant number of legal targets remain even under the strongest dynamic target constraints (*Live* $\neg$*GOT*), with the exception of a small number (the aforementioned cases with httpd, and one case in Mem-

Table 5.3. Taint information and residual attack surface for nginx. **Callsite**: controllable indirect call when sending a plain `HTTP GET` request. **Taint source**: taint information for the function pointer (target) and first three arguments (arguments actually used are underlined). None indicates an untainted value, while CPtr, DPtr, and ¬Ptr indicate taint through a code pointer, data pointer (and possibly CPtr), or non-pointer value (and possibly Ptr), respectively. **Targets**: available targets for the given callsite under *Src* (source types, the strongest static) constraints and *Best*: combining *Live* ¬*GOT* (strongest dynamic constraints) and *Src*. Call targets marked with † are tainted by segregated state, indicating that the call may be used in a history flushing attack against CsCFI.

		Taint source				Targets	
	Callsite	*FPtr.*	*Arg0*	*Arg1*	*Arg2*	*Src.*	*Best*
1	connection.c:808	CPtr	¬Ptr	¬Ptr	¬Ptr	2	1
2	epoll_module.c:642	DPtr	DPtr	¬Ptr	None	19	6
3	event.c:245	CPtr	None	¬Ptr	None	1	1
4	event.c:286	CPtr	DPtr	¬Ptr	None	2	2
5	event_accept.c:258	DPtr	DPtr	¬Ptr	¬Ptr	6	1
6	http_chunked_filter_module.c:79	CPtr	¬Ptr	None	None	58	18
7	http_chunked_filter_module.c:92	CPtr	¬Ptr	None	¬Ptr	11	8
8	http_charset_filter_module.c:235	CPtr	¬Ptr	¬Ptr	¬Ptr	58	18
9	http_charset_filter_module.c:552	CPtr	¬Ptr	None	None	11	8
10	http_core_module.c:852	¬Ptr	¬Ptr	¬Ptr	¬Ptr	8	7
11	http_core_module.c:874	CPtr	¬Ptr	¬Ptr	¬Ptr	58	18
12	http_core_module.c:906	¬Ptr	¬Ptr	¬Ptr	¬Ptr	58	18
13	http_core_module.c:1075	CPtr	¬Ptr	¬Ptr	¬Ptr	58	18
14	http_core_module.c:1357	¬Ptr	¬Ptr	¬Ptr	DPtr	58	18
15	http_core_module.c:1825	CPtr	¬Ptr	None	None	58	18
16	http_core_module.c:1840	CPtr	¬Ptr	None	None	11	8
17	http_gzip_filter_module.c:256	CPtr	¬Ptr	None	None	58	18
18	http_gzip_filter_module.c:323	CPtr	¬Ptr	None	None	11	8
19	http_headers_filter_module.c:152	CPtr	¬Ptr	None	None	58	18
20	http_log_module.c:252	DPtr	¬Ptr	¬Ptr	None	6	1
21	http_log_module.c:297	DPtr	¬Ptr	¬Ptr	DPtr	12	11
22	http_not_modified_filter_module.c:61	CPtr	¬Ptr	None	¬Ptr	58	18
23	http_postpone_filter_module.c:82	CPtr	¬Ptr	None	None	11	8
24	http_range_filter_module.c:230	CPtr	¬Ptr	None	None	58	18
25	http_range_filter_module.c:551	CPtr	¬Ptr	None	¬Ptr	11	8
26	http_request.c:514	DPtr	DPtr	¬Ptr	¬Ptr	19	6
27	http_request.c:1126	CPtr †	¬Ptr	¬Ptr	¬Ptr	3	3
28	http_request.c:3002	¬Ptr	¬Ptr	¬Ptr	¬Ptr	58	18
29	http_ssi_filter_module.c:329	CPtr	¬Ptr	None	None	58	18
30	http_ssi_filter_module.c:392	CPtr	¬Ptr	None	None	11	8
31	http_userid_filter_module.c:205	CPtr	¬Ptr	¬Ptr	None	58	18
32	http_variables.c:404	¬Ptr	¬Ptr	¬Ptr	¬Ptr	61	49
33	http_write_filter_module.c:238	¬Ptr †	¬Ptr	¬Ptr	None	2	1
34	output_chain.c:65	¬Ptr	¬Ptr	None	None	11	8
35	palloc.c:80	¬Ptr	¬Ptr	None	¬Ptr	56	7

Table 5.4. Summarized number of controllable callsites and targets for each server.

(a) Callsites: number of tainted (controllable) callsites under varying write constraints. **Targets (static)**: total permissible targets (median) under each static target constraint.

	Callsites		Targets (static)		
Server	*Write constraint*	*Tainted*	*Bin types*	*Safe src types*	*Src types*
nginx	None	35	1,952	988	201
	¬CPtr	13	1,952	953	193
	¬Ptr	8	1,952	787	160
	Segr	2	1,571	108	5
	Segr & ¬Ptr	1	1,571	2	2
lighttpd	None	12	1,686	249	50
	¬CPtr	7	1,512	228	37
	¬Ptr	2	1,187	56	6
	Segr	8	1,686	230	39
	Segr & ¬Ptr	2	1,187	56	6
httpd	None	33	3,464	1,471	310
	¬CPtr	27	3,464	1,469	302
	¬Ptr	13	3,408	1,079	139
	Segr	0	0	0	0
	Segr & ¬Ptr	0	0	0	0
redis	None	14	2,253	470	219
	¬CPtr	11	2,253	470	219
	¬Ptr	11	2,253	470	219
	Segr	2	1,227	13	11
	Segr & ¬Ptr	2	1,227	13	11
memcached	None	8	2,314	275	35
	¬CPtr	3	1,624	243	7
	¬Ptr	3	1,624	243	7
	Segr	1	2,105	18	18
	Segr & ¬Ptr	0	0	0	0
bind	None	43	2,762	1,323	393
	¬CPtr	40	2,762	1,253	383
	¬Ptr	39	2,762	1,241	371
	Segr	1	1,936	199	20
	Segr & ¬Ptr	1	1,936	199	20

(b) Total permissible targets (absolute) under each dynamic target constraint. Note that these apply for *all* write constraints that have at least one tainted callsite.

	Targets (dynamic)				
Server	*Baseline*	*Live +page*	*Computed*	*Live*	*Live ¬GOT*
nginx	4,592	2,336	786	767	708
lighttpd	4,450	1,867	497	474	409
httpd†	6,113	3,835	2,002	1,985	1,928
redis	5,381	2,311	771	612	546
memcached‡	4,326	2,420	752	738	391
bind	7,693	2,829	1,028	1,010	918

†No callsites for Segr and Segr & ¬Ptr
‡No callsites for Segr & ¬Ptr

cached). The same is true for static target constraints; even under source-level type-based target constraints, an attacker has multiple targets to choose from (ranging from 2 to 393 targets) in each case where callsite corruption is possible. For several servers, including nginx, lighttpd, Redis, and BIND, these results apply even to a segregated state attack model with type-based target constraints.

These results show that NEWTON is capable of locating controllable callsites and a choice of potential targets under even the strongest defenses. Recall that these results assume a low-effort attacker, sending only a single request to each server; thus, these results are a lower-bound for the number of controllable callsites.

5.7 Constructing Attacks

This section documents our experience using NEWTON to bypass two advanced state-of-the-art defenses: CsCFI [30, 52, 88, 102, 131] and CPI [83]. Our case studies are constructed in an architecture-independent fashion: unlike traditional ROP, we operate on program semantics. Thus, our results are generally applicable on different architectures, such as x86 and ARM. We specifically focus our analysis on secure implementations of CPI and CsCFI, given that existing work has already discussed the general limitations of CFI [25, 26, 43, 49, 57, 58] and leakage-resistant randomization [116].

5.7.1 CsCFI

In this case study, we target CsCFI on nginx. As described in Section 5.5, to bypass CsCFI's write constraint (*Segr*), we look for callsites controllable from a segregated (connection-specific) state. We (1) open a connection c_1 to prepare its memory state, (2) flush the branch history by sending n parallel requests over another connection c_2 (disabling CsCFI's protection), and finally (3) send a request over connection c_1 to divert control flow from a $C1$-controlled callsite.

As shown in Table 5.3, NEWTON provides us with two candidate callsites to bypass CsCFI (those with the *Segr* column checked). We select callsite 33 in the function `ngx_http_write_filter`:

```
chain = c->send_chain(c, r->out, limit);
```

Here, `c` is a pointer to our connection state (`ngx_connection_t`), which contains a code pointer called `send_chain`. Clearly, the connection state and code pointers stored therein are isolated from other connections. Other than `send_`-

chain and c itself (first argument), Newton also reports that the second r->out argument is tainted and controllable from corrupted connection-specific state.

With manual inspection, we verified that (1) controlling the target and arguments with an arbitrary memory write to segregated state allows request handling to complete without crashes, (2) we also control the third argument by controlling the limit_rate and limit_rate_after configuration variables and flipping a single (uncovered) branch in the execution, and (3) execution continues correctly if the send_chain call is diverted to a different target returning a 0 value, allowing us to chain successful calls via repeated interactions with the server.

Newton also provides us with a list of all the possible 4592 targets (no target constraints) for our selected callsite. We target mprotect to escalate code reuse to a code corruption attack. This function expects three arguments: (1) the start address of the affected memory region, (2) the size of the region, and (3) the protection flags.

To select the start address, we overwrite the c pointer and repoint it to a counterfeit object prepared with identical connection state in a memory location of our choosing. To select the protection flags, we overwrite the limit_rate_after variable to ensure the final limit computation has the PROT_READ|WRITE|EXEC bits set in the lowest byte. To select the size, we need to redirect the r->out pointer to a value of our choosing. However, it is challenging to enforce a small r->out pointer value, since the lower part of the address space is not normally mapped. To address this challenge, we aim for a large mprotect surface, spanning from the heap (i.e., the controlled c pointer) all the way to libc code. The latter is the next region in line in the address space, only separated from the heap by a single unmapped gap. To fill the gap, we use a preliminary request to redirect control to libc's malloc without worrying about its argument—since this is a pointer, calling malloc will result in a large allocation, adjacent to libc in our setting.

At this point, we safely hijack our victim callsite to call mprotect and make the (now larger) heap and the entire libc code readable, writeable, and executable. Once mprotect succeeds, we issue another request to corrupt the gettimeofday function of libc with our own shellcode. The shellcode runs when nginx processes the next request, giving us arbitrary code execution. Figure 5.5 shows an overview of the attack.

Evidently, even a state-of-the-art defense like CsCFI alone is not sufficient to stop an attacker armed with dynamic analysis. Instead, to limit the power of these attacks, we must carefully combine context-sensitive CFI with traditional

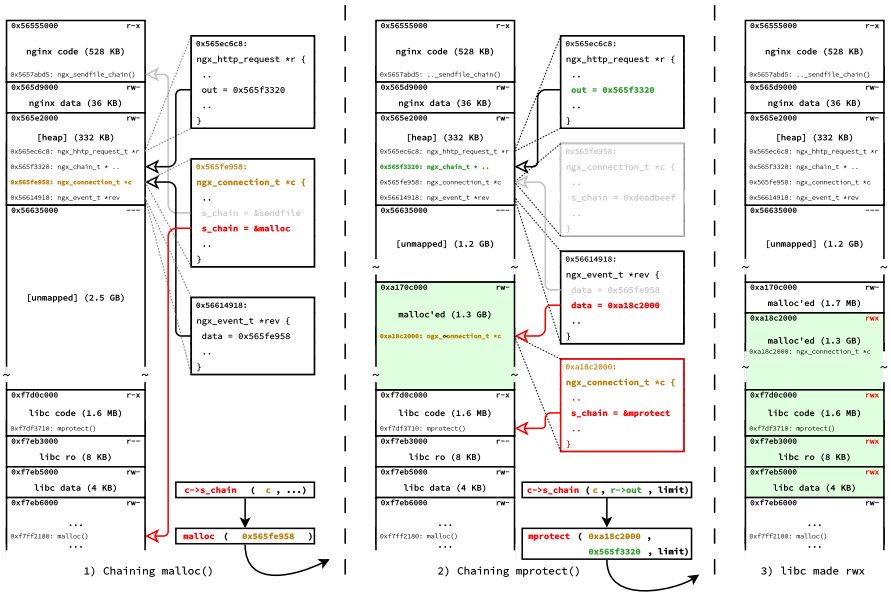


Figure 5.5. Chaining `malloc` and `mprotect` in nginx to make libc code pages writable, using the callsite `c->send_chain(c, r->out, limit)`. This figure illustrates memory layout and key variables of the nginx process before, during, and after our attack against CsCFI.

We first overwrite the `send_chain` code pointer in `c` with the address of `malloc`. Since the callsite uses the address of `c` as first argument, this results in a `0x565fe958` B = 1.3 GB allocation, adjacent to libc code. We then overwrite the same code pointer with the address of `mprotect` and construct a counterfeit `c` structure at a convenient location: knowing that the value of `r->out` will be the `len` argument for `mprotect(void *addr, size_t len, int prot)`, we place `c` at libc − `r->out`, i.e., `0xf7eb6000` − `0x565f3320` = `0xa18c2000` (rounded to the page boundary). To make nginx use our counterfeit object, we must also update the `data` pointer in the relevant `ngx_event_t *rev` structure. By using partial HTTP requests, we divide a single control-flow diversion into multiple steps: (1) open a connection c_1 and send a partial request; (2) use the arbitrary memory read/write primitive to corrupt the connection state of c_1, e.g., overwrite the `send_chain` code pointer; (3) open connections $c_2 \ldots c_n$ to perform n HTTP requests in parallel to flush CsCFI history, i.e., recorded branches that set `send_chain` to `ngx_sendfile_chain` are pruned from memory, (4) finish the partial request of c_1, triggering the control-flow diversion while CsCFI is unable to find in which context the overwritten code pointer was originally set.

CFI or other defenses. Note that state-of-the-art binary-level CFI policies based on argument/return count matching (TypeArmor) cannot prevent our `mprotect` hijacking attack, given that the callsite is diverted with a compatible function signature. Thus, stronger static (e.g., *Src types*) or dynamic (e.g., *Live*) target or write constraints that protect pointer corruption (e.g., CPI's ¬*Ptr*) are necessary.

To confirm the real-world applicability of NEWTON, we successfully implemented the above attack in practice. Using gdb to mimic an attacker's arbitrary read and write memory primitive, we recorded a video that shows how one can use our attack to mark libc memory pages as readable, writable, *and* executable. The video, accompanied with annoted details, is available on our project webpage.[3]

5.7.2 CPI

In this case study, we target CPI on nginx. CPI enforces a ¬Ptr write constraint, protecting code and data pointers. Thus, we use NEWTON's results in Table 5.3 to find callsites tainted by a non-pointer value, and select callsite 32. The callsite is in the function `ngx_http_get_indexed_variable`, and selects its callee from an array of structures with function pointers, as follows:

```
v[index].get_handler(r, &r->variables[index], v[index].data)
```

NEWTON's output pinpoints the taint source that we need to corrupt to control the `get_handler` function pointer: the `data` field in an `ngx_http_log_op_s` structure. It is worth noting how little effort it takes to find this dependency with NEWTON, as inspecting the source code reveals a complex data flow. The tainted data flows through multiple nginx-specific data structures and functions—none of which our low-effort attacker needs to know.

NEWTON also reveals that the taint source for the three arguments (Table 5.3) are all tainted by a non-pointer value. The last argument is controllable via the tainted `index`. The first two arguments are controllable by corrupting the allocator state much earlier in the execution. For example, the taint of the first `ngx_http_request_t*` argument originates 11 functions earlier in the execution, in `ngx_http_process_request_headers`. Again, NEWTON hides this complexity from the user.

With simple manual inspection, we also found that (1) the request data pointed to by the first argument is controllable by sending an incomplete `HTTP` request (which we complete later to trigger the exploit), (2) controlling the target and arguments with an arbitrary memory write allows request handling to complete without crashes, and (3) execution continues if the `get_handler` call is diverted to a different target, making it possible to chain calls via repeated interactions with the server.

Other than information on how to effect an arbitrary memory write and divert control flow, NEWTON also provides us with a list of the 767 usable targets

[3] https://vusec.net/projects/newton

stored in memory. This reflects CPI's *Live* target constraint. A complication is that we only control the `index` into the `v` array of `ngx_http_variable_t` structures. Since each structure contains 6 word-sized fields, only 1/6 of memory can be used to select live code pointer targets. Fortunately, this alignment restriction is bypassable using memory massaging (on the heap, stack, etc.) [19]. Moreover, Newton found the address of `dlopen` live in memory, allowing us to load arbitrary shared objects on the victim system and expand the set of available live targets.

For example, if we call `dlopen` on `"/bin/ed"` or other shared objects which use the `system` library call, we force the linker to bind the `system` code pointer in memory (GOT). This is easier after corrupting the linker configuration (`LD_LIBRARY_PATH`, `LD_BIND_NOW`). At that point, we again corrupt the `index` integer to redirect `get_handler` to the newly created live code pointer. Subsequently, we send another request to chain an invocation of the (now live) `system` library call, allowing us to execute arbitrary commands on the victim system. To "massage" the GOT to obtain a correctly aligned `system` code pointer, we carefully choose the `system`-dependent shared object to load.

We note that, other than CPI, the above attack bypasses all the defenses in the bottom-left quadrant marked by the $<\neg$Ptr, *Live*$>$ data point in Figure 5.2, including CCFI, TASR, PtrRR, XoM, and TypeArmor. Thus, an important lesson learned is that we must combine CPI with other strong defenses to further limit the attack surface. CPI combined with a secure implementation of CsCFI, for instance, would prevent us from controlling callsite 32.

When crafting the above attack in practice, we found that GNU libc enforces strict constraints on the flags provided to `dlopen`: unused bits should be zero, or else an error is returned.[4] This limits our attack, as it means that `index` should now be chosen such that the address of `r->variables[index]` is a valid flag for `dlopen` (e.g., `RTLD_NOW`), while `v[index].get_handler` still points to `dlopen`. Successful exploitation thus depends on the libc version. Musl libc, for example, does not enforce these constraints. Running nginx with musl libc, however, voids `dlopen` pointers in memory. Instead, we found code pointers to many functions of the `exec()` family, opening alternative ways for bypassing CPI.

5.8 Related Work

As we already discussed code-reuse defenses at length in this chapter, this section discusses the literature on code-reuse attacks only.

[4] https://sourceware.org/git/?p=glibc.git;h=3e539cb47e9fabfdda295926b4270b0f...

RETURN-TO-LIBC [221] represents the first generation of code-reuse attacks. Traditionally targeting the 32-bit x86 ISA, RETURN-TO-LIBC uses a memory corruption vulnerability to inject a return address on the stack pointing to an existing (libc) function, followed by function arguments. Thus, a subsequent `ret` instruction transfers control to the prepared function, thwarting DEP [185]. By preparing multiple call frames, function calls can be chained. On the x86-64 architecture, most function arguments are passed in CPU registers, making RETURN-TO-LIBC more challenging.

Return-Oriented Programming (ROP) [167] generalizes RETURN-TO-LIBC, and is now the de-facto standard in real-world code-reuse attacks. ROP also manipulates the stack, but doesn't chain complete functions. Instead, ROP uses small code fragments ending in return instructions, called *gadgets*. ROP is an extremely potent attack technique, which allows attackers to implement arbitrary Turing-complete computations in most practical programs [122].

The initial ROP attack signaled the start of an arms race around a third-generation of code-reuse attacks. Several defense techniques were developed, only to be shown susceptible to improved code-reuse attacks. *Jump-Oriented Programming* (JOP) [17] bypasses some execution monitoring defenses [44] and *Counterfeit Object-Oriented Programming* (COOP) [120] and related attacks [25, 26, 43, 49, 57, 58] bypass many existing Control-Flow Integrity (CFI) [2]-based defenses. Other attacks such as JIT ROP [42, 126], SROP [18], and AOCR [116] bypass information hiding defenses, including leakage-resistant variants [116]. The "gadget-stitching" model extends even beyond code reuse, also adopted by state-of-the-art techniques to craft data-only attacks [64, 65]. Note that although these recent efforts on Data-Oriented Programming (DOP) show similar weaknesses in modern defenses as outlined in this chapter, a key difference is that most of those defenses were never designed to mitigate data-only attacks. Attacks crafted with NEWTON, on the other hand, fall within the defenses' threat models.

Although the way NEWTON finds gadgets is somewhat similar to how ACICS gadgets are found [49], the latter are more constrained: only attacks where the function pointer and arguments are directly corruptible on the heap or in global memory are considered. As shown in Section 5.7, NEWTON finds more sophisticated attacks, where these elements may be corrupted in complex, indirect ways.

The focus on (manual or automatic) static analysis makes code reuse increasingly complex given increasingly sophisticated defenses. With NEWTON, we show that a switch to a simple and natural dynamic analysis approach significantly simplifies the discovery and stitching of gadgets, even in the face of state-

of-the-art defenses. Moreover, we argue that RETURN-TO-LIBC-style attacks on 64-bit architectures are not only practical, but also much easier, if an attacker piggybacks on the benign data flows of the application.

5.9 Conclusion

The "*geometry*" of innocent flesh on the bone has characterized ten years of code-reuse research: an attacker statically analyzes binary code to find gadgets, chains them together, and "calls" into security-sensitive syscalls. This model is simple to understand, but scales poorly as we assume increasingly sophisticated defenses.

In this chapter, we showed that, by also considering the "*dynamics*" of innocent flesh on the bone, even a low-effort attacker can easily find useful defense-aware gadgets to craft practical attacks. We implemented NEWTON, a gadget-discovery framework based on simple static and dynamic (taint) analysis. Using NEWTON, we found gadgets compatible with state-of-the-art defenses in many real-world programs. We also presented an nginx case study, showing that a NEWTON-armed attacker can find useful gadgets and craft attacks that comply with the restrictions of strong defenses such as CPI and context-sensitive CFI.

Our effort ultimately shows that, to sufficiently reduce the attack surface against a dynamic attack model, we must combine multiple state-of-the-art code-reuse defenses or, alternatively, deploy more heavyweight defenses at the cost of higher overhead.

6 Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

Recent work shows that the Rowhammer hardware bug can be used to craft powerful attacks and completely subvert a system. However, existing efforts either describe probabilistic (and thus unreliable) attacks or rely on special (and often unavailable) memory management features to place victim objects in vulnerable physical memory locations. Moreover, prior work only targets x86 and researchers have openly wondered whether Rowhammer attacks on other architectures, such as ARM, are even possible.

We show that *deterministic* Rowhammer attacks are feasible on commodity *mobile platforms* and that they cannot be mitigated by current defenses. Rather than assuming special memory management features, our attack, Drammer, relies on the predictable memory reuse patterns of standard physical memory allocators. We implement Drammer on Android/ARM, demonstrating the practicability of our attack, but also discuss a generalization of our approach to other Linux-based platforms. Furthermore, we show that x86-based Rowhammer exploitation techniques no longer work on mobile platforms and address the resulting challenges towards practical mobile Rowhammer attacks.

To support our claims, we present the first Rowhammer-based Android root exploit relying on *no software vulnerability*, and requiring *no user permissions*. In addition, we present an analysis of several popular smartphones and find that many of them are susceptible to our Drammer attack. We conclude by discussing potential mitigation strategies and urging our community to address the concrete threat of faulty DRAM chips in widespread commodity platforms.

6.1 Introduction

The Rowhammer hardware bug allows an attacker to modify memory without accessing it, simply by repeatedly accessing, i.e., "hammering", a given physical memory location until a bit in an adjacent location flips. Rowhammer has been used to craft powerful attacks that bypass all current defenses and completely subvert a system [59, 114, 235, 146]. Until now, the proposed exploitation techniques are either probabilistic [59, 235] or rely on special memory management features such as memory deduplication [114], MMU paravirtualization [146], or the `pagemap` interface [235]. Such features are often unavailable on commodity platforms (e.g., all are unavailable on the popular Amazon EC2 cloud, despite recent work explicitly targeting a cloud setting [114, 146]) or disabled for security reasons [219, 227]. Recent JavaScript-based attacks, in turn, have proven capable to reliably escape the JavaScript sandbox [19], but still need to resort to probabilistic exploitation to gain root privileges and to completely subvert a system [59].

Probabilistic Rowhammer attacks [59, 235] offer weak reliability guarantees and have thus more limited impact in practice. First, they cannot reliably ensure the victim object, typically a page table in kernel exploits [59], is surgically placed in the target vulnerable physical memory location. This may cause the Rowhammer-induced bit flip to corrupt unintended data (rather than the victim page table) and crash the whole system. Second, even when the victim page table is corrupted as intended, they cannot reliably predict the outcome of such an operation. Rather than mapping an attacker-controlled page table page into the address space as intended, this may cause the Rowhammer-induced bit flip to map an unrelated page table, which, when modified by the attacker, may also corrupt unintended data and crash the whole system.

This chapter makes two contributions. First, we present a *generic* technique for *deterministic* Rowhammer exploitation using *commodity* features offered by modern operating systems. In particular, we only rely on the predictable behavior of the default physical memory allocator and its memory reuse patterns. Using this technique (which we term Phys Feng Shui), we can reliably control the layout of physical memory and deterministically place security-sensitive data (e.g., a page table) in an attacker-chosen, vulnerable physical memory location.

Second, we use said technique to mount a deterministic Rowhammer attack (or Drammer) on mobile platforms, since they present different and unexplored hardware and software characteristics compared to previous efforts, which focus only on x86 architectures, mainly in desktop or server settings. Concerning the

hardware, mobile platforms mostly use ARM processors. However, all known Rowhammer techniques target x86 and do not readily translate to ARM. Moreover, researchers have questioned whether memory chips on mobile devices are susceptible to Rowhammer at all or whether the ARM memory controller is fast enough to trigger bit flips [232, 235]. Concerning the software, mobile platforms such as Android run different and more limited operating systems that implement only a subset of the features available in desktop and server environments. For example, unless explicitly specified by a device vendor, the Android kernel does currently not support huge pages, memory deduplication, or MMU paravirtualization, making it challenging to exploit the Rowhammer bug and impossible to rely on state-of-the-art exploitation techniques.

DRAMMER is an instance of the *Flip Feng Shui* (*FFS*) exploitation technique (abusing the physical memory allocator to surgically induce hardware bit flips in attacker-chosen sensitive data) [114], which for the first time relies only on always-on commodity features. For any Rowhammer-based *Flip Feng Shui* attack to be successful, three primitives are important. First, attackers need to be able to "hammer sufficiently hard"—hitting the memory chips with high frequency. For instance, no bits will flip if the memory controller is too slow. Second, they need to find a way to massage physical memory so that the right, exploitable data is located in the vulnerable physical page. Third, they need to be able to target specific contiguous physical addresses to achieve (1) double-sided Rowhammer [8, 235], a technique that yields more flips in less time than other approaches, and (2) more control when searching for vulnerable pages (important when mounting deterministic attacks). We show that, when attacking mobile platforms, none of these primitives can be implemented by simply porting existing techniques.

In this chapter, we present techniques to implement aforementioned primitives when attacking mobile platforms. We detail the challenges towards reliable exploitation on Android/ARM and show how to overcome its limited feature set by relying on DMA buffer management APIs provided by the OS. To concretely demonstrate the effectiveness of our DRAMMER attack on mobile platforms, we present the first deterministic, Rowhammer-based Android *root* exploit. Our exploit can be launched by any Android app with no special permission and without relying on any software vulnerability.

Finally, we present an empirical study and investigate how widespread the Rowhammer bug is on mobile devices. We investigate how fast we can exploit these bugs in popular smartphones and identify multiple phones that suffer from faulty DRAM: 17 out of 21 of our tested 32-bit ARMv7 devices—still the most dominant platform with a market share of over 97% [224]—and 1 out of our 6

tested 64-bit ARMv8 phones are susceptible to Rowhammer. We conclude by discussing how state-of-the-art Rowhammer defenses are ineffective against our DRAMMER attack and describe new mitigation techniques.

In summary, we make the following contributions:

- We present the first technique to perform *deterministic* Rowhammer exploitation using only *commodity* features implemented by modern operating systems.
- We demonstrate the effectiveness of our technique on mobile platforms, which present significant hardware and software differences with respect to prior efforts. We implement our DRAMMER attack on Android/ARM and present the first deterministic, Rowhammer-based Android root exploit. Our exploit cannot be mitigated by state-of-the-art Rowhammer defenses.
- We evaluate the effectiveness of DRAMMER and our Android root exploit and complement our evaluation with an empirical Rowhammer study on multiple Android devices. We identify ARMv7 and ARMv8 smartphones that suffer from faulty DRAM.
- To support future research on mobile Rowhammer, we release our codebase as an open source project and aim to build a public database of known vulnerable devices.[1]

6.2 Threat Model

We assume that an attacker has control over an unprivileged Android app on an ARM-based device and wants to perform a privilege escalation attack to acquire root privileges. We do not impose any constraints on the attacker-controlled app or the underlying environment. In particular, we assume the attacker-controlled app has no permissions and the device runs the latest stock version of the Android OS with all updates installed, all security measures activated, and no special features enabled.

6.3 Rowhammer Exploitation

Rowhammer is a software-induced hardware fault that affects dynamic random-access memory (DRAM) chips. In practice, this has the net effect that a piece of

[1] https://www.vusec.net/projects/drammer/

software can flip some bits in physical memory by solely performing memory read operations. It is important to note that triggering the Rowhammer bug is different than using (i.e., exploiting) it in a security-relevant manner. In fact, an exploit usually needs to trick a victim component (e.g., another process, the OS, or another VM hosted on the same physical node) to use a vulnerable physical memory location to store security-sensitive content. In the general case, software exploitation of this kind proved to be challenging.

In this section, we first provide general background information on memory hardware and the Rowhammer bug. Then, we summarize existing exploitation techniques and describe the three distinct primitives that Rowhammer exploits need to implement.

6.3.1 Memory Hardware

In order to understand the root cause of the Rowhammer bug, it is important to understand the architecture and components of DRAM chips. DRAM works by storing charges in an array of *cells*, each of which consists of a capacitor and an access transistor. A cell represents a binary value depending on whether it is charged or not. Cells are further organized in *rows*, which are the basic unit for memory accesses. On each access, a row is "activated" by copying the content of its memory cells to a *row buffer* (thereby discharging them), and then copying the content back to the memory cells (thereby charging them). A group of rows that is serviced by one row buffer is called a *bank*. Finally, multiple banks further form a *rank*, which spans across multiple *DRAM chips*. A *page frame* is the smallest fixed-length contiguous block of physical memory into which the OS maps a *memory page* (a contiguous block of virtual memory). From a DRAM perspective, a page frame is merely a contiguous collection of memory cells, aligned on the page-size boundary (typically 4 KB).

Memory cells naturally have a limited retention time and leak their charge over time. Therefore, they have to be refreshed regularly in order to keep their data. Thus, the DDR3 standard [177] specifies that the charge of each row has to be refreshed at least every 64 ms. This memory refresh interval is a trade-off between memory integrity on the one hand, and energy consumption and system performance on the other. Refreshing more often consumes more power and also competes with legitimate memory accesses, since a specific memory region is unavailable during the refresh [160].

6.3.2 The Rowhammer Bug

In a quest to meet increasing memory requirements, hardware manufacturers squeeze more and more cells into the same space. Unfortunately, Kim et al. [76] observed that the increasing density of current memory chips also makes them prone to disturbance errors due to charge leaking into adjacent cells on every memory access. They show that, by repeatedly accessing, i.e., "hammering," the same memory row (the aggressor row) over and over again, an attacker can cause enough of a disturbance in a neighboring row (the victim row) to cause bits to flip. Thus, triggering bit flips through Rowhammer is essentially a race against the DRAM internal memory refresh in performing enough memory accesses to cause sufficient disturbance to adjacent rows. Relying on activations of just one aggressor row to attack an adjacent row is called *single-sided Rowhammer*, while the more efficient *double-sided Rowhammer* attack accesses the two rows that are directly above and below the victim row [235].

6.3.3 Exploitation Primitives

While it was originally considered mostly a reliability issue, Rowhammer becomes a serious security threat when an attacker coerces the OS into storing security-sensitive data in a vulnerable memory page (a virtual page that maps to a page frame consisting of at least one cell that is subject to the Rowhammer bug). Depending on the underlying hardware platform, OS, and already-deployed countermeasures, prior efforts developed different techniques to perform a successful end-to-end Rowhammer attack. This section summarizes prior techniques and describes the three required primitives to exploit the Rowhammer bug.

P1. Fast Uncached Memory Access This primitive is the prerequisite to flip bits in memory and refers to the ability of activating rows in each bank *fast enough* to trigger the Rowhammer bug. In practice, this can be non-trivial for two reasons. First, the CPU memory controller might not be able to issue memory read commands to the memory chip fast enough. However, most often the challenge relates to the presence of several layers of caches, which effectively mask out all the CPU memory reads (after the first one). Thus, all known exploitation techniques need to implement a mechanism to bypass (or nullify) the cache.

P2. Physical Memory Massaging This primitive consists of being able to trick the victim component to use—in a *predictable* or, in a weaker form, *probabilistic*

way—a memory cell that is subject to the Rowhammer bug. More importantly, the attacker needs to be able to *massage* the memory precisely enough to push the victim to use the vulnerable cell to store security-sensitive data, such as a bit from a page table entry. This primitive is critical for an attacker to mount a privilege escalation attack, and it is also the most challenging one to implement in a fully *deterministic* way.

P3. Physical Memory Addressing This last primitive relates to understanding how physical memory addresses are used in the virtual address space of an unprivileged process. While this primitive is not a hard requirement for Rowhammer exploitation in general, it is crucial to perform double-sided Rowhammer: to access memory from two aggressor rows, an attacker must know which virtual addresses map to the physical addresses of these rows.

6.4 The First Flip

This section documents our efforts on hammering memory chips of mobile devices. We focus on testing the hardware without attempting to mount any exploit. In the next section, we then discuss how going from "flipping bits" to mounting a root privilege escalation attack is challenging, given that there are several aspects that make successful exploitation on mobile devices fundamentally different compared to previous efforts on desktop and server settings.

6.4.1 RowhARMer

Researchers have speculated that Rowhammer on ARM could be impossible, one of the main reasons being that the ARM memory controller might be *too slow* to trigger the Rowhammer bug [232, 235]. Not surprisingly, no existing work from academia or industry currently documents any success in reproducing the Rowhammer bug on mobile devices.

We set up an experiment to test whether memory chips used in mobile devices are subject to bit flips induced by Rowhammer and whether the ARM memory controller can issue *memory read operations* fast enough. Since this is a preliminary feasibility analysis, we perform the Rowhammer attack from a kernel module (i.e., with full privileges), which allows us to cultivate optimal conditions for finding bit flips: we disable CPU caching and perform double-sided Rowhammer by using the `pagemap` interface to find aggressor rows for each victim address.

We hammer rows by performing one million read operations on their two aggressors. To determine the minimum memory access time that still results

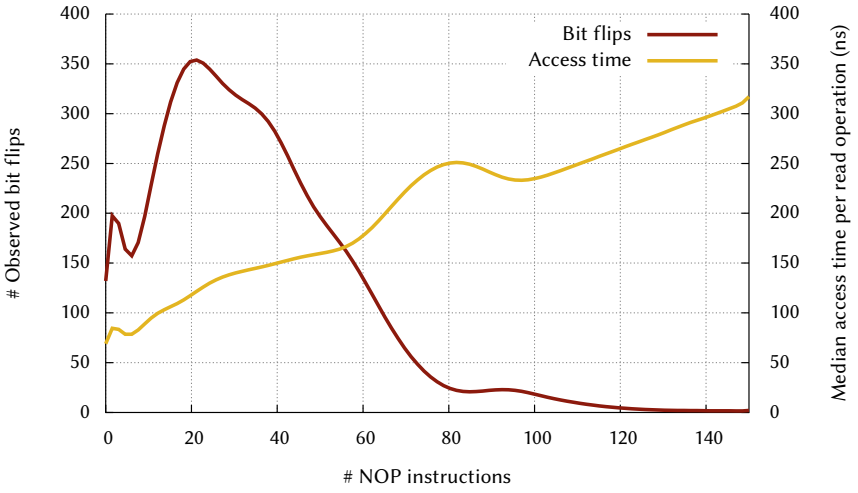


Figure 6.1. Dependency between observed bit flips (y_1) and memory access time (y_2) when repeatedly hammering the same 5 MB memory chunk, while increasing the number of NOP instructions (x) to simulate slower access times. The spike in access time around 60 to 80 NOP instructions may be caused by a background process during our analysis. The spike in observed bit flips around 20 NOP instructions indicates a "sweet spot" of memory access time.

in bit flips, we repeatedly hammer the same 5 MB of physical memory while artificially increasing the time between two read operations by inserting NOP instructions. We measure the time it takes to complete a single read operation and report the median over all hammered pages. We initiate all bytes in the victim row to `0xff` (all bits are set) and once the hammering phase finishes, we scan the victim row for bit flips—i.e., any byte that has a value different than `0xff`. Since we only perform *read* operations, any such modification to the memory content can be directly attributed to Rowhammer.

For this experiment, we used an LG Nexus 5 device running Android 6.0.1 (the latest version at the time of writing). The results of this experiment are encouraging: not only do bits flip, but it is also relatively simple to obtain them. In fact, we triggered flips in a matter of seconds, and observed up to 150 flips per minute. Figure 6.1 depicts the results of our experiment. It shows the dependency between the access time and the number of bit flips found when scanning a 5 MB memory chunk. Moreover, it shows that access times of 300 ns or higher are unlikely to trigger bit flips and that, surprisingly, the "sweet spot" for triggering the most flips on this particular DRAM chip is not reading at full speed (70 ns per read), but throttling to 100 ns per read. However, note that throttling does not

necessarily result in a lower rate of actual accesses to DRAM cells: the memory controller may reorder accesses internally.

6.5 Exploitation on the x86 Architecture

Even when a memory chip is vulnerable to Rowhammer attacks, it is challenging to perform a successful end-to-end exploitation—we need to implement the three primitives described earlier. We now review how existing Rowhammer exploitation techniques, which all target the x86 architecture, implement those primitives.

6.5.1 P1. Fast Uncached Memory Access

First and foremost an attacker needs the capability to activate alternating rows in each bank *fast enough* to trigger the Rowhammer bug. The main challenge here is to bypass the CPU cache. For this purpose, state-of-the-art attacks rely on one of the following techniques:

Explicit cache flush. This technique is based on using the `clflush` instruction, which flushes the cache entry associated to a given address. One can execute this instruction after accessing a particular memory address, so that subsequent *read* operations on that same address also trigger DRAM accesses. On x86 architectures, the `clflush` instruction is particularly useful because it can be executed even by a non-privileged process. This technique is used, for example, by Seaborn et al. [235, 216] and is based on the findings from Kim et al. [76].

Cache eviction sets. This technique relies on repeatedly accessing memory addresses that belong to the same *cache eviction set* [8, 19, 59]. A cache eviction set is defined as a set of *congruent* addresses, where two addresses are congruent if and only if they *map* to the same cache line. Thus, accesses to a memory address belonging to the same congruent set will automatically flush the cache while reading (because the associated cache line contains the content of the previously-read memory address). This observation is the basis for the several access patterns described by Gruss et al. [59] and is particularly useful when the `clflush` instruction is not available (e.g., when triggering Rowhammer from JavaScript).

Non-temporal access instructions. This technique relies on accessing memory using CPU instructions or APIs that, by design, do not use the cache. Previous efforts rely on non-temporal write instructions (e.g., `MOVNTI`, `MOVNTDQA`),

which are also used in some `memset()` and `memcopy()` implementations [112]. In this context, *non-temporal* means that the data will not likely be reused soon and thus does not have to be cached. As a result, these operations cause the CPU to directly write the content to memory, thus bypassing the cache.

6.5.2 P2. Physical Memory Massaging

This primitive is essential to *trick* the victim component into storing security-sensitive data (e.g., a page table) in an attacker-chosen, vulnerable physical memory page. Existing efforts have mainly relied on the following techniques for this purpose:

Page-table spraying Previous work exploits the Rowhammer bug to achieve root privilege escalation by flipping bits in page table entries (PTEs) [235, 216]. This attack suggests a *probabilistic* exploitation strategy that *sprays* the memory with page tables, hoping that at least one of them lands on a physical memory page vulnerable to Rowhammer. The next step is then to flip a bit in the vulnerable physical memory page, so that the victim page table points to an arbitrary physical memory location. Given the sprayed physical memory layout, such location should *probabilistically* contain one of the attacker-controlled page table pages (PTPs) which allows attackers to map their own page tables in the controlling address space. At that point, they can overwrite their own PTEs and access arbitrary (e.g., kernel) pages in physical memory to escalate privileges.

Memory deduplication Razavi et al. [114] abuse memory deduplication to perform deterministic Rowhammer exploitation [19, 114]. They show that an attacker can use memory deduplication to trick the OS into mapping two pages, an attacker-controlled virtual memory page and a victim-owned virtual memory page, to the same attacker-chosen vulnerable physical memory page. While such an exploitation strategy is powerful and has been successfully demonstrated in a cross-VM setting, it relies on memory deduplication, which is not an always-on feature, even in modern operating systems (e.g., off by default on Linux).

MMU paravirtualization Xiao et al. [146] leverage Xen MMU paravirtualization to perform deterministic Rowhammer exploitation from a guest VM. This exploits the property that Xen allows a guest VM to specify the physical location of a (read-only) PTP, allowing a malicious VM to trick the VM monitor into mapping a page table into a vulnerable location to "hammer." Similar to memory deduplication, this is not an always-on feature and only available inside Xen

MMU paravirtualized VMs. In addition, MMU paravirtualization is no longer the common case in popular cloud settings, with MMU virtualization becoming much more practical and efficient.

6.5.3 P3. Physical Memory Addressing

Processes have direct access only to virtual memory which is then mapped to physical memory. While the virtual memory layout is known to processes in userland, the physical memory layout is not. As discussed in the previous section, to perform double-sided Rowhammer, an attacker needs to repeatedly access specific physical memory pages. For this purpose, previous efforts suggest the following techniques:

Pagemap interface This technique relies on `/proc/self/pagemap` which provides *complete* information about the mapping of virtual to physical addresses. Clearly, having access to this information is sufficient to repeatedly access specific rows in physical memory.

Huge pages Another option is to use *huge (virtual) pages* that are backed by physically contiguous physical pages. In particular, a huge page covers 2 MB of contiguous physical addresses. Although this is not as fine-grained as knowing absolute physical addresses, one can use relative offsets to access specific physical memory pages for double-sided Rowhammer. In fact, it guarantees that two rows that are contiguous in virtual memory are also contiguous in physical memory.

6.5.4 Challenges on Mobile Devices

When assessing whether one can exploit Rowhammer bugs on mobile devices, we attempted, as a first step, to reuse known exploitation techniques described above. We found, however, that none of the primitives are applicable to mobile devices. Table 6.1 presents an overview of our analysis.

Explicit cache flush (P1) On ARMv7, the cache flush instruction is privileged and thus only executable by the kernel. Since our threat model assumes an unprivileged app, we cannot use this instruction to implement P1. Although the Android kernel exposes a `cacheflush()` system call to userland, this system call flushes only up to the Level 2 cache and thus fails to force repetitive DRAM accesses for a single address. Interestingly, ARMv8 does provide unprivileged cache flush instructions, but they may be disabled by the kernel.

Table 6.1. Techniques previously used for x86-based Rowhammer attacks and their availability on ARM-based mobile devices in unprivileged mode (●), privileged mode (○), or not at all (–). Some techniques are available in unprivileged mode, but are not practical enough to use in our setting (◐). Note how none of these techniques can be generally applied on all modern versions of mobile devices.

Primitive	x86 platforms	Mobile devices
Fast Uncached Memory Access		ARMv7/ARMv8
Explicit cache flush	●	○/◐
Cache eviction sets	●	–/–
Non-temporal access instructions	●	–/◐
Physical Memory Massaging		
Page-table spraying	●	◐
Memory deduplication	●	–
MMU paravirtualization	●	–
Physical Memory Addressing		
Pagemap interface	●	○
Huge pages	●	–

Cache eviction sets (P1) In principle, it is possible to use cache eviction sets to flush addresses from the cache. Unfortunately, this technique proved to be too slow in practice to trigger bit flips on both ARMv7 and ARMv8.

Non-temporal access instructions (P1) ARMv8 offers non-temporal load and store instructions, but they only serve as a hint to the CPU that caching is not useful [174]. In practice, we found that memory still remains cached, making these instructions not usable for our exploitation goals.

Page-table spraying (P2) As documented by Seaborn et al. [216], the page-table spraying mechanism is probabilistic and may crash the OS. We aim to implement a deterministic Rowhammer exploit and thus cannot rely on this technique.

Special memory management features (P2) Although device vendors may enable memory deduplication for *Low RAM* configurations [199], it is not enabled by default on stock Android. Moreover, MMU paravirtualization is not available and we can thus not rely on existing special memory management features.

Pagemap interface (P3) The Linux Kernel no longer allows unprivileged access to `/proc/self/pagemap` since version 4.0 [219]. This change was ported to the Android Linux kernel in November 2015 [214], making it impossible for us to use this interface for double-side Rowhammer.

Huge pages (P3) Although some vendors ship their mobile devices with huge page support (Motorola Moto G, 2013 model, for example), stock Android has

this feature disabled by default. We thus cannot rely on huge pages to perform double-sided Rowhammer in our setting.

Additional challenges In addition, there are further characteristics that are specific to mobile devices and that affect mobile Rowhammer attacks. First, the ARM specifications [173, 174] do not provide memory details and, for example, it is not clear what the size of a row is. Second, mobile devices do not have any swap space. Consequently, the OS—the Low Memory Killer in particular on Android—starts killing processes if the memory pressure is too high.

6.6 The Drammer Attack

We now describe how we overcome the limited availability of known techniques on mobile devices and how we mount our DRAMMER attack in a deterministic fashion. In contrast to most primitives discussed in the previous section, DRAMMER relies on general memory management behavior of the OS to perform deterministic Rowhammer attacks. For simplicity, we first describe our attack for Android/ARM (focusing on the more widespread ARMv7 platform) and later discuss its applicability to other platforms in Section 6.8.

6.6.1 Mobile Device Memory

One prerequisite to implement useful exploitation primitives is to understand the memory model of the chip we are targeting. One of the key properties to determine is the *row size*. Previous x86-based efforts ascertain the row size either by consulting the appropriate documentation or by running the `decode-dimms` program. Unfortunately, ARM does not document row sizes, nor does its platform provide instructions for fingerprinting DRAM modules. As such, we propose a *timing-based side channel* to determine a DRAM chip's row size.

We can apply our technique independently from the chosen target architecture. It relies on the observation that accessing two memory pages from the same bank is slower than reading from different banks: for same-bank accesses, the controller has to refill the bank's row buffer for each read. In particular, accessing physical pages n and $n+i$ while increasing i from 0 to x, shows a slower access time when page $n+i$ lands in the same bank as page n. Such increase in access time indicates that we walked over all the pages in a row and that $n+i$ now points to the first page in the second row, falling in the same bank. By setting x large enough (e.g., to 64 pages, which would indicate a row size of 256 KB), we ensure that we always observe the side channel, as it is not expected that the row size is 256 KB or larger. We evaluate our side channel in Section 6.9.

6.6.2 DMA Buffer Management

Modern (mobile) computing platforms consist of several different hardware components: besides the CPU or System-on-Chip (SoC) itself, devices include a GPU, display controller, camera, encoders, and sensors. To support efficient memory sharing between these devices as well as between devices and userland services, an OS needs to provide direct memory access (DMA) memory management mechanisms. Since processing pipelines that involve DMA buffers bypass the CPU and its caches, the OS must facilitate explicit cache management to ensure that subparts of the pipeline have a coherent view of the underlying memory. Moreover, since most devices perform DMA operations to physically contiguous memory pages only, the OS must also provide allocators that support this type of memory.

We refer to the OS interface that provides all these mechanisms as a *DMA Buffer Management API* which essentially exports "DMA-able" memory to userland. By construction, userland-accessible DMA buffers implement two of our attack primitives: (P1) providing uncached memory access and (P3) (relative) physical memory addressing.

6.6.3 Physical Memory Massaging

For the remaining and most crucial primitive (P2), we need to arrange the physical memory in such a way that we can control the content of a vulnerable physical memory page and *deterministically* land security-sensitive data therein. For this purpose, we propose Phys Feng Shui, a novel technique to operate physical memory massaging that is solely based on the predictable *memory reuse patterns* of standard physical memory allocators. In addition to being deterministic, this strategy does not incur the risk of accidentally crashing the system by causing bit flips in unintended parts of physical memory.

On a high level, our technique works by exhausting available memory chunks of different sizes to drive the physical memory allocator into a state in which it has to start serving memory from regions that we can reliably predict. We then force the allocator to place the target security-sensitive data, i.e., a page table, at a position in physical memory which is vulnerable to bit flips and which we can hammer from adjacent parts of memory under our control.

Memory Templating Since our attack requires knowledge about which exact memory locations are susceptible to Rowhammer, we first need to probe physical memory for flippable bits—although the number and location of vulnerable

memory regions naturally differs per DRAM chip, once found, the large majority of flips is reproducible [76]. We refer to this process as *memory templating* [114]. A successful templating session results in a list of *templates* that contain the location of vulnerable bits, as well as the direction of the flip, i.e., whether it is a 0-to-1 or 1-to-0 flip.

Physical Memory Allocator Linux platforms manage physical memory via the buddy allocator [170]. Its goal is to minimize external fragmentation by efficiently splitting and merging available memory in power-of-2 sized blocks. On each allocation request, it iteratively splits larger blocks in half as necessary until it finds a block matching the requested size. It is important to note that the buddy allocator always prioritizes the smallest fitting block when splitting—e.g., it will not attempt to split a block of 16 KB if a fitting one of 8 KB is also available. As a result, the largest contiguous chunks remain unused for as long as possible.

On each request for deallocation, the buddy allocator examines neighboring blocks of the same size to merge them again if they are free. To minimize the internal fragmentation produced by the buddy allocator for small objects, Linux implements a slab allocator abstraction on top of it. The default *SLUB* allocator implementation organizes small objects in a number of pools (or *slabs*) of commonly used sizes to quickly serve allocation and deallocation requests. Each slab is expanded on demand as necessary using physically contiguous chunks of a predetermined per-slab size allocated through the buddy allocator.

6.6.4 Phys Feng Shui

Phys Feng Shui lures the buddy allocator into reusing and partitioning memory in a predictable way. For this purpose, we use three different types of physically contiguous chunks: large chunks (`L`), medium-sized chunks (`M`), and small chunks (`S`). The size of small chunks is fixed at 4 KB (the page size). Although other values are possible (see also Section 6.7), for simplicity, we set the size of `M` to the row size and use the size of the largest possible contiguous chunk of memory that the allocator provides for `L`. As illustrated in Figure 6.2, our attack then includes the following steps:

Preparation and templating We first exhaust (i.e., allocate until no longer possible) all available physically contiguous chunks of size `L` (step 1) and probe them for vulnerable templates which we later can exploit. We then exhaust all chunks of size `M` (step 2), leaving the allocator in a state where blocks of size `M` and larger are no longer available (until existing ones are released).

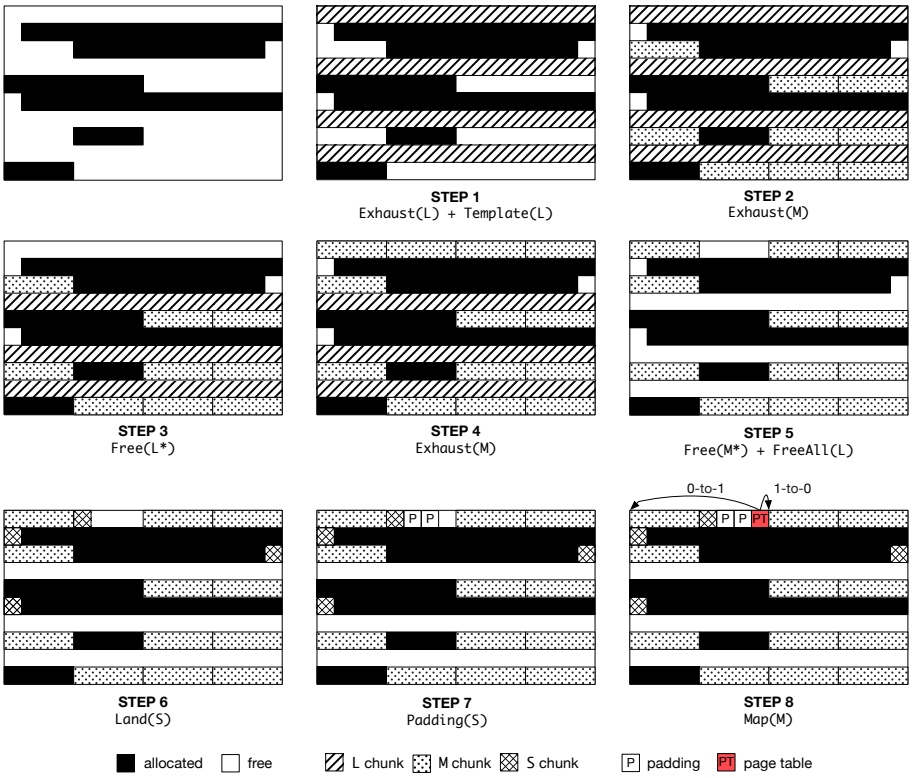


Figure 6.2. Physical memory layout before and after each step of Phys Feng Shui. Depending on the direction of the targeted bit flip, we map either the chunk before or after the vulnerable one in the last step.

Selective memory reuse Next, we select one of the templates generated in the previous step as the target for our exploit and refer to its corresponding L block as L*. We then release L* (step 3), and immediately exhaust all M chunks again (step 4). Since we depleted all the free chunks of size M or larger in the previous step, this forces the allocator to place them in the region we just released (i.e., predictably reuse the physical memory region of the released L* chunk). We refer to the M chunk that now holds the exploitable template as M*.

Finally, in preparation of landing the page table (PT) in the vulnerable page of M*, in the next step we release M* (step 5). Note that we restrict our choice of M* to chunks that are not at the edge of L*, since we need access to its surrounding memory in order to perform double-sided Rowhammer later.

Our technique naturally increases memory pressure. In practice, the OS handles low memory or out of memory (OOM) conditions by freeing up unused memory when the available memory falls under a certain threshold. This is especially

critical on mobile devices, which do not utilize swap space. In preparation of the next steps, which need to allocate several S chunks and would risk bumping the amount of available memory below the threshold, we now free the remaining L chunks to avoid triggering memory cleanup by the OS (or worse: a system crash).

Landing the first page table in the vulnerable region We now steer the memory allocator to place a S chunk in the vulnerable chunk M* that was released. For this purpose, we deplete the allocator of available blocks of size S…M/2 by repeatedly allocating S chunks. This guarantees that subsequent S allocations land in M* (step 6). We allocate S chunks by forcing (4 KB) page table allocations: we repeatedly map memory at fixed virtual addresses that mark page table boundaries (i.e., every 2 MB of the virtual address space on ARMv7). Since the maximum number of page tables per process is 1024, we spawn a number of worker processes to allocate as many as we need. Once all smaller chunk sizes are depleted, our next S allocation predictably lands in the vulnerable region (no other smaller block is available).

Determining when allocations reach the vulnerable region is trivial on Linux: the proc filesystem provides `/proc/zoneinfo` and `/proc/pagetypeinfo`. These special files are world-readable and detail information on the number of available and allocated memory pages and zones. In case these files are not available, we can exploit a timing or instruction-count (via the Performance Monitoring Unit) side-channel to detect when S lands in M*: depending on whether the allocator can serve a request from a pool of available chunks of a matching size, or whether it has to start splitting larger blocks, the operation takes more time and instructions. We can use this observation by measuring the time between an allocation and deallocation of an M chunk every time we force the allocation of a new S chunk. Once this time falls below a certain (adaptively computed) threshold, we know that we are filling the vulnerable region with S, since the allocator could no longer place a new M chunk there and had to start breaking down blocks previously occupied by one of the former L chunks.

Aligning the victim page table Finally, we map a page p in the former L* chunk that neighbors M* on the left (in case of a 0-to-1 flip), or on the right (in case of a 1-to-0 flip), at a fixed location in the virtual memory address space to force a new PTP allocation (step 8). Depending on the virtual address we pick, the page table entry (PTE) that points to p is located at a different offset within the PTP—essentially allowing us to align the victim PTE according to the vulnerable template.

We can similarly align the victim PTP according to the vulnerable page to make sure that we can flip selected bits in the victim PTE. For this purpose, we force the allocation of a number of padding PTPs as needed before or after allocating the victim PTP (step 7).

We further need to ensure that the vulnerable PTP allocated in `M*` and the location of p are 2^n pages apart: flipping the n lowest bit of the physical page address in the victim PTE deterministically changes the PTE to point to the vulnerable PTP itself, mapping the latter into our address space. To achieve this, we select any page p in the `M` chunk adjacent to `M*` to map in the victim PTP, based on whether it satisfies this property.

Exploitation Once we selected and aligned the victim PTP, PTE, and n according the vulnerable template, we perform double-sided Rowhammer and replicate the bit flip found in the templating phase. Once we trigger the desired flip, we gain write access to the page table as it is now mapped into our address space. We can then modify one of our own PTPs and gain access to any page in physical memory, including kernel memory.

Note that the exploit is fully reliable and may only fail if the flip discovered in the templating phase is not reproducible (e.g., if a 0-to-1 flip is now applied to a 1-bit content). Since the buddy allocator provides chunk alignment by design, however, we can address this issue. By exploiting knowledge about relative offsets inside the `L*` chunk, we can predict the lower bits of physical addresses in the vulnerable PTE. For example, if `L*` is of size 4 MB, meaning that it must start at a physical address that is a multiple of 2^{22}, we can predict the lower $2^{22}/4096 = 2^{10} = 10$ bits of all 1024 page frames that fall in `L*`. Thus, if our templating phase on ARMv7 reports a 0-to-1 flip in page 426, at bit offset 13 of a 32-bit word—a potential PTE, where offsets 1–12 are part of its properties field—we can immediately conclude that this flip is not exploitable: if, after Rowhammer, bit 13 is 1, the PTE may never point to its own page 426 (in fact, it could only point to uneven pages). In contrast, a 1-to-0 flip in page 389, at bit offset 16 of a 32-bit word is exploitable if we ensure that a PTE at this location points to page 397: `____ ____ __01 1000 1101 | ppro pert iess` flips to `____ ____ __01 1000 0101 | ppro pert iess` (= 389).

6.6.5 Exploitable Templates

The number of templates that an attacker can use for exploitation is determined by a combination of (1) the number of flips found in potential PTEs, and (2) the relative location of each flip in `L*`.

As discussed earlier, a bit flip in a PTE is exploitable if it flips one of the lower bits of the address part. For ARMv7, this means that flips found in the lowest 12 bits of each 32-bit aligned word are not exploitable as these fall into the properties field of a PTE. Moreover, a flip in one of the higher bits of a PTE is also not exploitable in a deterministic matter: a 0-to-1 flip in the highest bit would require the PTE to point to a page that is, physically, 2 GB to the right of its PTP. Without access to absolute physical addresses, we can only support bit flips that trigger a page offset shift of at most the size of $\mathsf{L} - 1$. For example, if L is 4 MB (512 page frames), a 0-to-1 flip in bit 9 of a possible 32-bit word in the first page of L, is exploitable: the exploit requires a PTE that points to a page that is $2^9 = 256$ pages away from the vulnerable page. The same flip in page 300 of L, however, is not exploitable, as it would require an entry pointing to a page outside of L.

In addition, ARMv7's page tables are, unlike x86's ones, of size 1 KB. Linux, however, expects page tables to fill an entire page of 4 KB and solves this by storing two 1 KB ARM hardware page tables, followed by two shadow pages (used to hold extra properties that are not available in the hardware page tables), in a single 4 KB page. This design further reduces the number of exploitable flips by a factor two: only flips that fall in the first half of a page may enclose a hardware PTE.

To conclude, for ARMv7, with a maximum L size of 4 MB, a template is not exploitable if (1) it falls in the second half of a page (a shadow page) (2) it falls in the lowest 12 bits of a 32-bit word (the properties field of a PTE), or (3) it falls in the highest 11 bits of a 32-bit word. Consequently, for each word, at most 9 bits are exploitable, and since there are only 256 hardware PTEs per page, this means that, *at most*, $2,304$ bits out of all possible $32,768$ bits of a single page are exploitable (around 7.0%).

6.6.6 Root Privilege Escalation

Once we have control over one of our own PTPs, we can perform the final part of the attack (i.e., escalate privileges to obtain *root* access). For this purpose, we repeatedly map different physical pages to scan kernel memory for the security context of our own process (`struct cred`), which we identify by using a unique 24-byte signature based on our unique (per-app) UID. We discuss more details and evaluate the performance of our Android root exploit in Section 6.9.

6.7 Implementation

To demonstrate that deterministic Rowhammer attacks are feasible on commodity mobile platforms, we implemented our end-to-end DRAMMER attack on Android. Android provides DMA Buffer Management APIs through its main memory manager called ION, which allows userland apps to access uncached, physically contiguous memory. Note, however, that DRAMMER extends beyond ION and we discuss how to generalize our attack on other platforms in Section 6.8.

6.7.1 Android Memory Management

With the release of Android 4.0, Google introduced ION [229] to unify and replace the fragmented memory management interfaces previously provided by each hardware manufacturer. ION organizes its memory pools in at least four different in-kernel heaps, including the `SYSTEM_CONTIG` heap, which allocates physically contiguous memory allocated via `kmalloc()` (slab allocator). Furthermore, ION supports buffer allocations with explicit cache management, i.e., cache synchronization is left up to the client and memory access is essentially direct, uncached. Userland apps can interact with ION through `/dev/ion`—allowing uncached, physically contiguous memory to be allocated by any unprivileged app without any permissions.

Our implementation uses ION to allocate `L` and `M` chunks—and maps such chunks to allocate `S` page table pages (4 KB on Android/ARM). Given that SLUB's `kmalloc()` resorts directly to the buddy allocator for chunks larger than 8 KB, we can use ION to reliably allocate 16 KB and larger chunks. We set `L` to 4 MB, the largest size `kmalloc()` supports. This gives us the most flexibility when templating and isolating vulnerable pages. Although more complex configurations are possible, for simplicity we set `M` to the row size (always larger than 16 KB). Intuitively, this allows us to release a single vulnerable row for page table allocations, while still controlling the aggressor rows to perform double-sided Rowhammer. Supporting other `M` values yields more exploitable templates, at the cost of additional complexity.

6.7.2 Noise Elimination

To ensure reliability, an attacker needs to eliminate interference from other activity in the system (e.g., other running apps) during the PHYS FENG SHUI phase. The risk of interference is, however, minimal. First, our PHYS FENG SHUI phase is designed to be extremely short-lived and naturally rule out interference. Second, interference is only possible when the kernel independently allocates memory

via the buddy allocator in the low memory zone. Since most kernel allocations are served directly from slabs, interference is hard to find in practice.

Nonetheless, the attacker can further minimize the risk of noise by scheduling the attack during low system activity, e.g., when no user is interacting with the device or when the system enters low power mode with essentially no background activity. Android provides notifications for both scenarios through the intents `ACTION_SCREEN_OFF` and `ACTION_BATTERY_LOW`.

6.8 Generalization

ION facilitates a Rowhammer attack on Android/ARM by readily providing DMA buffer management APIs to userland, but it is not yet available on every Linux platform (although there are plans to upstream it [236, 222]). Nonetheless, we describe how one can generalize our deterministic attack to other (e.g., x86) platforms by replacing ION with other standard capabilities found in server and desktop Linux environments. More specifically, the use of ION in DRAMMER can be replaced with the following strategies:

(1) Uncached memory Rather than having ION map uncached memory in userland, one can employ `clflush` or any of the other cache eviction techniques that have previously been used for Rowhammer [19, 59, 76, 182, 183, 112, 114, 235, 146].

(2) `L` chunks Transparent hugepages (THP) [179] supported by Linux (enabled by default on recent distributions for better performance [187, 213] and available on some Android devices) can replace the physically contiguous `L` chunks allocated by ION. By selectively unmapping part of each `L` chunk backed by a THP, one can also directly create a `M`*-sized hole without filling it with `M` chunks first. To learn when a THP allocation fails (i.e., when `L` chunks have been exhausted), one can read statistics from `/proc/vmstat` [203]. To force the kernel to allocate THPs (normally allocated in high memory) in low memory (normally reserved to kernel pages, e.g., PTPs), one can deplete the `ZONE_HIGHMEM` zone before starting the attack, as detailed in prior work [73]. Note that THPs contribute to the ability to mount deterministic attacks, not just to operate double-sided Rowhammer [114, 235, 146], in our setting.

(3) `M` chunks While using THPs to exhaust `M` chunk allocations is infeasible (they are larger), one can abuse the SLUB allocator and force it to allocate chunks

of a specific size through the buddy allocator. This can be done by depleting a slab cache of a carefully selected size (e.g., depleting `kmalloc-256` would force one 4 KB allocation) by triggering multiple allocations via particular system calls, e.g., `sendmmsg()`, as described by Xu et al. [147, 148].

(4) Predictable PTE content To ensure that our victim PTE points to an attacker-controlled physical page at a predictable offset, we rely on mapping neighboring chunks (the first and the last fragment of a THP in our generalized attack) multiple times in userland (the first time at allocation time, the second time when creating the victim page table). To implement this strategy with THPs, one can create a shared memory segment associated to the vulnerable THP (shared memory support for THPs is being merged mainline [220]) and attach it multiple times at fixed addresses to force page table allocations with PTEs pointing to it. As an alternative (until THP shared memory support is available mainline), one can simply map a new anonymous (4 KB) user page when forcing the allocation of the page table. As the `ZONE_HIGHMEM` zone is depleted, such user page will also end up in low memory right next to the PTP.

6.9 Evaluation

In this section, we evaluate various aspects of DRAMMER. We (1) evaluate our proposed side channel to detect the row size on a given device. Using its results, we (2) perform an empirical study on a large set of Android smartphones to investigate to what extent consumer devices are affected by susceptible DRAM chips. Finally, we (3) combine results from our study with the final exploitation step (i.e., from page table write access to root privilege escalation) and compute how fast we can perform our end-to-end attack.

6.9.1 Mobile Row Sizes

We evaluate the row size detection side channel described in Section 6.6 by constructing a heatmap for page-pair access times. We set our upper limit in both directions to 64 pages, and thus read from page pairs $(1, 1)$, $(1, 2)$, …, $(1, 64)$, $(2, 1)$, $(2, 2)$, …, $(64, 64)$.

Figure 6.3 shows such a heatmap for a LG Nexus 5 phone. We determined that the row size is 16 pages, which is $16 \times 4K = 64K$. This is somewhat surprising, given that most previous x86 efforts report a row size of 128K.

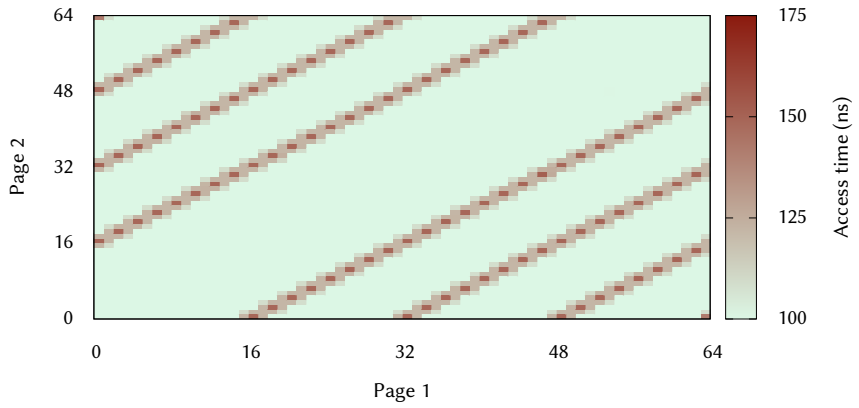


Figure 6.3. Heatmap representing the time required to access a given pair of pages on a LG Nexus 5. The diagonal pattern clearly indicates that the row size is 16 pages = 64K.

6.9.2 Empirical Study

For our empirical study, we acquired the following ARMv7-based devices: 15 LG Nexus 5 phones, a Samsung Galaxy S5, two One Plus Ones, two Motorola Moto G devices (the 2013 and 2014 model), and a LG Nexus 4. We further analyzed a number of ARMv8-based devices: a LG Nexus 5X, a Samsung Galaxy S6, a Lenovo K3 Note, a Xiaomi Mi 4i, a HTC Desire 510, and a LG G4.

We subject each device to a Rowhammer test, which (1) exhausts all the large ION chunks to allocate a maximum amount of hammerable memory (starting at 4 MB chunks, down to chunks that are 4 times the row size); (2) performs double-sided Rowhammer on each victim *page* for which aggressor pages are available twice and checks the entire victim *row* for flips (i.e., we hammer once with all victim bits set to 1—searching for 1-to-0 flips—and once with all bits set to 0); and (3) for each induced flip, dumps its virtual (and physical, if `/proc/self/pagemap` is available) address, the time it took to find it (i.e., after n seconds), its direction (1-to-0 or 0-to-1), and whether it is exploitable by our specific Drammer attack, as discussed in Section 6.6.

To hammer a page, we perform 2x 1M read operations on its aggressor pages. Although prior work shows that 2.5M reads yields the optimal amount of flips, lowering the *read count* allows us to finish our experiments faster, while still inducing plenty of bit flips.

Table 6.2. Empirical analysis results. For each device, the table shows the amount of hammered DRAM (*MB*), the median access time for a single read operation (*ns*), the number of unique flips found (*#flips*), the average amount of KB that contain a single flip (*KB*), the number of *#1-to-0* and *#0-to-1* flips, the number of exploitable templates according our attack (*#exploitable*), and after how many seconds of hammering we found the first exploitable flip (1^{st}). For same model devices, we use a subscript number to identify them individually. The top half of the table shows ARMv7-based (32-bit) smartphones, while the lower rows are ARMv8 (64-bit).

		Analysis results							
	Device	*MB*	*ns*	*# flips*	*KB*	*# 1-to-0*	*# 0-to-1*	*# exploitable*	1^{st}
ARMv7	Nexus 5_1	441	70	1,058	426	1,011	47	62 (5.86%)	116s
	Nexus 5_2	472	69	284,428	2	261,232	23,196	14,852 (5.22%)	1s
	Nexus 5_3	461	69	547,949	1	534,695	13,254	32,715 (5.97%)	1s
	Nexus 5_4	616	71	0	–	–	–	–	–
	Nexus 5_5	630	69	747,013	1	704,824	42,189	46,609 (6.24%)	1s
	Nexus 5_6	512	69	215,233	3	207,856	7,377	13,365 (6.21%)	3s
	Nexus 5_8	485	70	32,328	15	28,500	3,828	1,894 (5.86%)	4s
	Nexus 5_9	569	69	476,170	2	434,086	42,084	30,190 (6.34%)	0s
	Nexus 5_{10}	406	69	160,245	3	150,485	9,760	8,701 (5.43%)	1s
	Nexus 5_{11}	613	70	0	–	–	–	–	–
	Nexus 5_{12}	600	70	17,384	35	16,767	617	1,241 (7.14%)	16s
	Nexus 5_{13}	575	69	161,514	4	160,473	1,041	10,378 (6.43%)	355s
	Nexus 5_{14}	576	69	295,537	2	277,708	17,829	18,900 (6.40%)	1s
	Nexus 5_{15}	573	69	38,969	15	35,515	3,454	2,775 (7,12%)	11s
	Nexus 5_{17}	621	70	0	–	–	–	–	–
	Galaxy S5	207	82	0	–	–	–	–	–
	OnePlus One_1	292	71	3,981	75	2,924	1,057	242 (6.08%)	942s
	OnePlus One_2	1,189	69	1,992	611	942	1,050	94 (4.72%)	326s
	Moto G_{2013}	134	127	429	275	419	10	30 (6.99%)	441s
	Moto G_{2014}	151	127	1,577	98	1,523	54	71 (4.66%)	92s
	Nexus 4	82	18	1,328	64	1,061	267	104 (7.83%)	7s
ARMv8	Nexus 5x	271	63	0	–	–	–	–	–
	Galaxy S6	234	82	0	–	–	–	–	–
	K3 Note	423	218	0	–	–	–	–	–
	Mi 4i	327	159	0	–	–	–	–	–
	Desire 510	186	122	0	–	–	–	–	–
	G4	833	64	117,496	8	117,260	236	6,560 (5.58%)	5s

Our results (presented in Table 6.2) show that many tested ARMv7 devices are susceptible to the Rowhammer bug, while our ARMv8 devices seem somewhat more reliable (although still vulnerable). However, due to our small current ARMv8 sample size, we cannot conclude that ARMv8 devices are more resilient by design. Moreover, it should be noted that our Nexus 5 devices have been used extensively in the past for a variety of research projects. This may indicate a

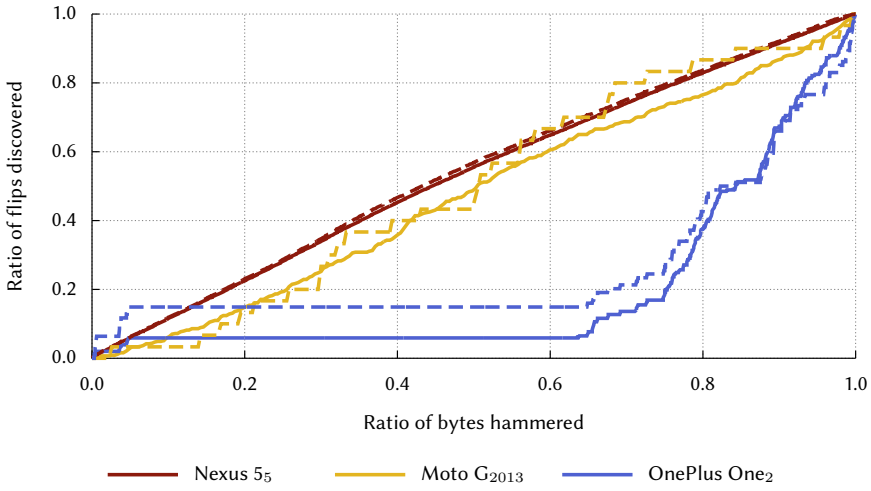


Figure 6.4. Template distribution over memory for three cases—one with many flips (Nexus 5_5), one with few flips (Moto G_{2013}), one with a large memory region without flips (OnePlus One_2).

correlation between (heavy) use and the ability to induce (more) bit flips. Future research is required, however, to confirm whether *DRAM wearing* actually contributes to more observable bit flips.

A second key finding presented in Table 6.2 is that a device from 2013 (Moto G, 1^{st} generation) is vulnerable to the Rowhammer bug. This shows that ARM memory controllers have been capable in performing Rowhammer attacks for at least three years. Moreover, the MSM8226 SoC (Qualcomm Snapdragon 400), which both Moto Gs are shipped with, is still a popular SoC for smartwatches (e.g., Motorola Moto 360 2^{nd} generation or Huawei Watch), suggesting that smartwatches may be susceptible to Rowhammer-based exploits as well. Additionally, while the majority of flips are induced on LPDDR3-devices, the reported flips on the Nexus 4 show that even LPDDR2 is be vulnerable.

Third, when looking at the exploitability of observed flips (*exploitable* and *1st* columns of Table 6.2), we conclude that *once we see flips, we will eventually find exploitable flips.* Although only around 6% of all observed flips are exploitable, we always find enough flips to also find those that we can use in our end-to-end Android root exploit.

To get a better understanding of the template distribution over memory, we display a CDF for three interesting devices in Figure 6.4: one with many flips (Nexus 5_5), one with few flips (Moto G_{2013}), and the OnePlus One_2. For most

other devices, the distribution trend is similar as that of the Nexus 5 and Moto G. For the OnePlus One, however, it is interesting to notice that a large region of the DRAM exposes no flips at all (from 5% to 60%).

6.9.3 Root Privilege Escalation

Armed with bit flips from the memory templating step, we rely on PHYS FENG SHUI to place the victim page table in a vulnerable template-matching location and proceed to reproduce an exploitable bit flip. This step allows us to control one of our own page tables, enabling root privilege escalation. To complete our Android root exploit, we overwrite the controlled page table to probe kernel memory for our own `struct cred` structure.

The `struct cred` structure represents a process' security context and holds, among others, its real, effective, and saved user and group IDs (6 UIDs). Since Android provides each app—and thus each running process—a unique UID, we can fingerprint a security context by comparing $6 \times 4 = 24$ bytes. In our experiments on the latest kernel, the physical page that stores a specific `struct cred` has 20 bits of entropy, placed between `0x30000000` and `0x38000000`. Moreover, the structure is always aligned to a 128 byte boundary, that is there are $\frac{4096}{128} = 32$ possible locations within a page on which a `struct cred` can be found. To successfully find our own, we thus have to map and scan 2^{20} different physical pages in the worst-case scenario, and for each page perform 32 different compare operations, resulting in, at most, $2^{20} * 32 = 33,554,432$ calls to `memcmp`. Since we control only a single page table—on ARMv7 capable of storing PTEs to 512 physical pages—we also need to flush the TLB every 512 tries. Thus, in order to read all possible pages that may contain own our `struct cred`, we must perform $\frac{2^{20}}{512} = 2,048$ TLB flushes.

We flush the TLB by reading from $8,196$ different pages in a 32 MB memory region, which takes approximately 900 μs. Comparing 24 bytes using `memcmp()` takes at most 600 ns, limiting the upper bound time of the final exploitation step to $2,048 \times 900\,\mu s + 33,554,432 \times 600\,ns \sim 22$ seconds (measured on a Nexus 5). Note that having to break 20 bits of entropy does not make our attack less deterministic: we will always be able to find our own `struct cred`.

Based on the results from our empirical study, we find that, for the most vulnerable phone, an end-to-end attack (including the final exploitation step) takes less than 30 seconds, while in the worst-case scenario, it takes a little over 15 minutes, where templating is obviously the most time-consuming phase of the attack. To confirm that our exploit is working, we successfully exploited our Nexus 5_8 in less than 20 seconds.

Finally, to support future research on mobile Rowhammer and to expand our empirical study to a broader range of devices, we release our codebase as an open source project and aim to build a public database of known vulnerable devices on our project website.

6.10 Mitigation and Discussion

In this section, we investigate the effectiveness of current Rowhammer defenses and discuss potential design improvements of the memory management process that could mitigate our attack.

6.10.1 Existing Rowhammer Defenses

Countermeasures against Rowhammer have already been proposed, both in software and hardware, but very few are applicable to the mobile domain or effective against a generic attack such as the one we proposed.

Software-based *Instruction "blacklisting"*, i.e., disallowing or rewriting instructions such as `CLFLUSH` [235, 216] and non-temporal instructions [112] has been proposed as a countermeasure and is now deployed in Google Native Client (NaCl). Similarly, access to the Linux `pagemap` interface is now prohibited from userland [214, 219]. However, these countermeasures have already proven insufficient, since Rowhammer has been demonstrated in JavaScript [19, 59], where neither these special instructions nor the `pagemap` interface are present. As a more generic countermeasure, ANVIL [8] tries to detect Rowhammer attacks by monitoring the last-level cache miss rate and row accesses with high temporal locality. Similarly, Herath et al. [233] propose to monitor the number of last-level cache misses during a given refresh interval. Both approaches rely on CPU performance counters specific to Intel/AMD. Furthermore, our attack bypasses the cache completely, thus producing no cache misses that could raise red flags.

Hardware-based Memory with *Error Correcting Codes (ECC)* corrects single bit flip errors, and reports other errors. However, Lanteigne [183] studied Rowhammer on server settings with ECC and reported surprising results, as some server vendors implement ECC to report bit flip errors only upon reaching a certain threshold—and one vendor even failed to report any error. Likewise, ECC often does not detect multiple flips in a single row. *Doubling DRAM refresh rates* has been the response of most hardware vendors via EFI or BIOS updates [186, 202,

204]. It severely limits most attacks. However, Kim et al. [76] show that the refresh rate would need to be increased up to eight times to completely mitigate the issue. Aweke et al. [8] mention that both doubling the DRAM refresh rate and prohibiting the `CLFLUSH` instruction defeat Rowhammer attacks, but no system currently implements it. As increased refresh rates have severe consequences for both power consumption and performance [160], this countermeasure does not seem well-suited for mobile devices. In addition, it aligns poorly with the direction taken by the LPDDR4 standard [178], which requires the refresh rate to drop at low temperatures to conserve battery life.

Further mitigations rely on the *Detection of Activation Patterns* to refresh targeted rows and need support from the DRAM chip or the memory controller. The LPPDR4 standard proposes Target Row Refresh (TRR) [178], which seems to be an effective countermeasure, but we need to expand our study to more devices shipped with this type of memory. Probabilistic Adjacent Row Activation (PARA) [76] refreshes neighboring rows on each activation with a low probability (and thus very likely during repeated activations in a Rowhammer attack), but requires modifications of the memory controller to do so. ARMOR [197] introduces an extra cache buffer for rows with frequent activations, i.e., hammered rows, but again needs to be implemented in the memory controller.

6.10.2 Countermeasures Against Drammer

We now elaborate on countermeasures that are more specific to our Drammer attack on mobile platforms.

Restriction of userland interface Since DMA plays an important part in the deterministic Rowhammer attack on mobile devices, the question arises whether userland apps should be allowed unrestricted access to DMA-able memory. On Android, the motivation for doing so via ION is device fragmentation: vendors need to define custom heaps depending on the specific hardware requirements of each product, and provide a mapping of use cases to heaps in their custom implementation of `gralloc()`. It is Google's policy to keep the vendors' product-specific code in user rather than in kernel mode [223].[2] Linux implements similar DMA-support with the dma-buf buffer sharing API [217], but with a more restricted interface. However, ION seems to fill a gap in this regard [231] and efforts are underway to upstream it [236, 222].

[2]Full discussion at the Linux Plumbers Conference 2013: https://www.youtube.com/watch?v=8okc75j5cKk

Concurrently to our work Google has adopted several defenses from the Linux kernel in Android [226] concerning memory protection and attack surface reduction. While the majority of defenses do not affect our attack, Android now provides mechanisms to enforce access restrictions to `ioctl` commands and added seccomp to enable system call filers. These mechanisms could be used to restrict the userland interface of ION.

However, we note that disabling ION is not the enough to stop Rowhammer-based attacks: (1) as discussed in Section 6.8, it is possible to generalize our attack to other Linux-based platforms without ION; (2) since a large number of DRAM chips are vulnerable to bit flips and an attacker might still be able to exploit them through other means. Nevertheless, improvements to the interface of ION and memory management in general could significantly raise the bar for an attacker. For example, a possible improvement is to adopt constraint-based allocation [218], where the (ION) allocator picks the type of memory for an allocation based on constraints on the devices sharing a buffer and defers the actual allocation of buffers until the first device attaches one (rather than upon request from a userland client).

Memory isolation and integrity In the face of an OS interface that provides user applications access to DMA-able memory, stricter enforcement of memory isolation may be useful. Specifically, it may be possible to completely isolate DMA-able memory from other regions. Currently, ION readily breaks the isolation of memory zones by allowing userland to allocate physically contiguous memory in low memory regions usually reserved for the kernel and page tables. One option is to isolate ION regions controlled by userland from kernel memory. In principle, ION can already support isolated heaps (e.g., ION carveout), but such heaps are statically preallocated and do not yet provide a general buffer management primitive. Furthermore, even in the absence of ION, an attacker can force the buddy allocator to allocate memory (e.g., huge or regular pages) in kernel memory zones by depleting all the memory available to userland [73]. Thus, the design of isolation and integrity measures for security-critical data such as page tables also needs improvements.

For instance, the characteristics of the underlying DRAM cells could be taken into account when allocating memory regions for security-critical data. Flikker proposes to allocate critical data in memory regions with higher refresh rates than non-critical data [87]. RAPID suggests that the OS should prefer pages with longer retention times, i.e., that are less vulnerable to bit flips [137]. Even without a DRAM-aware allocator, isolating security-critical data (e.g., page tables) in

zones that the system never uses for data that can be directly (e.g., ION buffers) or indirectly (e.g., slab buffers) controlled would force attackers to resort to a probabilistic attack with low chances of success (no deterministic or probabilistic memory reuse). However, enforcing strict isolation policies is challenging as, when faced with high memory pressure, the physical page allocator naturally encourages cross-zone reuse to eliminate unnecessary OOM events—opening up again opportunities for attacks [73]. In addition, even strict isolation policies may prove insufficient to completely shield security-sensitive data. For example, ION is also used by the media server, which is running at a higher privilege than normal apps. Hence, an attacker controlling a hypothetically isolated ION region could still potentially corrupt security-sensitive data, i.e., the media server's state, rather than, say, page tables.

Prevention of memory exhaustion Per-process memory limits could make it harder for an attacker (1) to find exploitable templates and (2) exhaust all available memory chunks of different sizes during PHYS FENG SHUI. Android already enforces memory limits for each app, but only at the Dalvik heap level. As a countermeasure, we could enforce this limit at the OS level (accounting for *both* user and kernel memory), and per-user ID (to prohibit collusion).

6.11 Related Work

The Rowhammer bug has gathered the attention of the scientific community for two years, beginning with the work of Kim et al. [76], who studied the possibility and the prevalence of bit flips on DDR3 for x86 processors. Aichinger [4] later analyzed the prevalence of the Rowhammer bug on server systems with ECC memory and Lanteigne performed an analysis on DDR4 memory [182]. In contrast to these efforts, we are the first to study the prevalence of the Rowhammer bug on ARM-based devices. Several efforts focused on finding new attack techniques [8, 19, 59, 76, 112, 114]. However, all these techniques only work on x86 architectures and, as discussed in Section 6.5.4, are not applicable to our setting.

DRAMMER is an instance of the *Flip Feng Shui* (*FFS*) exploitation technique [114]. Rather than using memory deduplication, DRAMMER relies on PHYS FENG SHUI for physical memory massaging on Linux. This shows that *FFS* can be implemented with always-on commodity features.

Lipp et al. [86] demonstrated cache eviction on ARM-based mobile devices, but did not evaluate the possibility of Rowhammer attacks based on cache eviction. Other attack techniques focus on the DRAM itself. Lanteigne [182, 183] ex-

amined the influence of data and access patterns on bit flip probabilities on DDR3 and DDR4 memory on Intel and AMD CPUs. Pessl et al. [106] demonstrated that reverse engineering the bank DRAM addressing can reduce the search time for Rowhammer bit flips. These techniques are complementary to our work.

Another line of related work uses the predictable memory allocation behavior of Linux for the exploitation of use-after-free vulnerabilities. Kemerlis et al. [73] showed in their ret2dir attack how kernel allocators can be forced to allocate user memory in kernel zones. Xu et al. [147, 148] used the recycling of slab caches by the SLUB allocator to craft the PingPongRoot root exploit for Android. Finally, Lee Campbell [188] relied on kernel object reuse to break out of the Chrome sandbox on Android.

In concurrent work, Zhang et al. [155] perform a systematic analysis of the design and implementation of ION, although without studying the topic of cache coherency. They found similar security flaws as the ones we exploit for our attack, but described different attack scenarios: (1) due to the unlimited access to ION heaps, both concerning access restrictions and memory quotas, an attacker can perform a denial-of-service attack by exhaustively allocating all device memory; (2) the recycling of kernel memory for userland apps—and in this case missing buffer zeroing logic in between—makes ION vulnerable to sensitive information leakage. Consequently, their proposed redesign of ION contains some of the countermeasures we discussed in the previous section, e.g., enforcing memory quotas and restricting the userland interface. However, they observe that implementing these changes is challenging, as they incur performance penalties, break backward-compatibility, and add complexity by introducing new security mechanisms specifically for the access to ION heaps.

6.12 Conclusion

In this chapter, we demonstrated that powerful *deterministic* Rowhammer attacks that grant an attacker root privileges on a given system are possible, even by only relying on always-on features provided by commodity operating systems. To concretely substantiate our claims, we presented an implementation of our Drammer attack on the Android/ARM platform. Not only does our attack show that practical, deterministic Rowhammer attacks are a real threat for billions of mobile users, but it is also the first effort to show that Rowhammer is even possible at all (and reliably exploitable) on any platform other than x86 and with a much more limited software feature set than existing solutions. Moreover, we demonstrated that several devices from different vendors are vulnerable to Rowhammer.

To conclude, our research shows that practical large-scale Rowhammer attacks are a serious threat and while the response to the Rowhammer bug has been relatively slow from vendors, we hope our work will accelerate mitigation efforts both in industry and academia.

7 GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM

Over the last two years, the Rowhammer bug transformed from a hard-to-exploit DRAM disturbance error into a fully weaponized attack vector. Researchers demonstrated exploits not only against desktop computers, but also used single bit flips to compromise the cloud and mobile devices, all without relying on any software vulnerability.

Since hardware-level mitigations cannot be backported, a search for software defenses is pressing. Proposals made by both academia and industry, however, are either impractical to deploy, or insufficient in stopping all attacks: we present RAMPAGE, a set of DMA-based Rowhammer attacks against the latest Android OS, consisting of (1) a root exploit, and (2) a series of app-to-app exploit scenarios that bypass all defenses.

To mitigate Rowhammer exploitation on ARM, we propose GUARDION, a light-weight defense that prevents DMA-based attacks—the main attack vector on mobile devices—by isolating DMA buffers with guard rows. We evaluate GUARDION on 22 benchmark apps and show that it has a negligible memory overhead. We further show that we can improve system performance by re-enabling higher order allocations after Google disabled these as a reaction to previous attacks.

7.1 Introduction

For decades, defensive research on memory corruption could brush aside the threat of exploitation via hardware bugs as "outside the threat model," if not science fiction entirely. The frightening list of devastating Rowhammer attacks, however, published at one security venue after another [19, 50, 59, 114, 134, 146], suggests that we are in urgent need of practical defenses. In this chapter, we propose a practical, isolation-based protection that stops DMA-based Rowhammer attacks by carefully surrounding DMA buffers with DRAM-level guard rows. We focus our work on mobile devices as here, the problem is even more worrisome: unlike desktop and server machines, it is impossible to perform hardware upgrades.

Rowhammer on mobile devices The Rowhammer hardware bug at its core consists of the leakage of charge between adjacent memory cells on a densely packed DRAM chip [76]. Thus, whenever the CPU reads or writes one row of bits in the DRAM module, the neighboring rows are ever so slightly affected. Normally, this does not create problems as DRAM periodically refreshes the charge in its cells, well in time to preserve data integrity. However, an attacker who deliberately hits the same rows many times within a refresh interval may cause the charge leakage to accumulate to the point that a bit flips in an adjacent row and modify memory that she does not own. Initially considered a curiosity of relatively minor importance, researchers have shown that attackers can harness Rowhammer to completely subvert a system's security [19, 24, 59, 114, 235, 134, 146].

Clearly, the threat of Rowhammer attacks for smartphones and tablets is particularly serious, as replacing the memory chips of such devices is not an option. In addition, power consumption is a prime concern in the mobile world, and many of the hardware-level solutions (such as ECC memory or higher DRAM refresh rates) consume more power. Furthermore, even though newer standards such as LPDDR4 [178] discuss the adoption of Rowhammer mitigations, i.e., Target Row Refresh (TRR), they do so only as an *optional* protection mechanism, thus making LPDDR4 chips vulnerable as well [183, 134].

Existing software defenses are not effective Given the challenges of deploying hardware solutions, the development of effective software-based defenses is particularly important to protect mobile users against Rowhammer attacks. In our analysis, we systematically explore existing proposals, which fall into two

categories: techniques that attempt to prevent attackers from triggering bit flips, and those that focus on making it impossible for a bit flip to bring physical memory into an exploitable state (Section 7.4). We argue that both directions have limitations, either in terms of practicality (for instance because they require specific hardware features), or worse, in terms of effectiveness (as they still allow for Rowhammer exploitation). We demonstrate this ineffectiveness by presenting novel attacks that circumvent all existing proposed and implemented defense techniques (Section 7.5).

The need for *practical* solutions Security solutions need to strike a balance between security and practicality—a defense against Rowhammer attacks should not incur unacceptable performance overhead, nor should it severely reduce the amount of available memory. Conversely, it should be effective and hard to bypass. In this work, we propose GUARDION, which effectively and efficiently blocks all known DMA-based Rowhammer attacks against mobile devices (Section 7.6).

GUARDION builds on the observation that triggering bit flips on ARM-based mobile platforms is facilitated by using uncached memory, accessible through DMA allocations [134]. Albeit other techniques exist, most are either impractical or easily addressable on ARM. For example, the `cacheflush()` system call that is exposed to userland by the Android kernel, only flushes up to the Level 2 cache, and thus fails to force repetitive DRAM accesses for a single address. Additionally, ARMv8's unprivileged cache flush instruction can easily be disabled by the kernel and thus do not pose a security risk.

We thus explicitly limit our defense to the more generic class of DMA-based Rowhammer attacks that rely on uncached memory. Doing so has an important implication for our design: instead of attempting to isolate all sensitive information, which is impractical, we can instead isolate only DMA allocations. As we will show, DMA allocations constitute only a very small fraction of all allocations in the system, and we can hence afford to apply expensive fine-grained isolation for *each* DMA allocation using guard rows. In our design, we isolate DMA allocations from the rest of the system by using two guard rows, one at the top and another at the bottom. With this scheme, an attacker can no longer use DMA allocations to trigger bit flips in any memory page in the system except in the guard rows. In effect, this design defends against Rowhammer by eradicating the ability to inject bit flips in sensitive data.

Can GuardION defend against any Rowhammer exploit? No. GUARDION only enforces that DMA-based Rowhammer attacks can no longer flip bits in another process or kernel memory. Attacks that induce bit flips by means of cache eviction sets—another popular Rowhammer technique on x86—are still possible. The (1) lento, and (2) idiosyncratic nature of these attacks, however, make them harder to launch in practice. First, increased access times will result in less flipped bits at a slower rate. Second, a substantial amount of reverse engineering is required for such attacks, and this work must be repeated for each target architecture [50, 134]. Thus, although not stopping all possible attacks, GUARDION reduces the attack surface significantly.

Contributions In summary, we make the following contributions:

- We systematically explore the design of software defenses, and show that existing proposals are either not practical or not effective.
- To back our claims, we present RAMPAGE, a set of DMA-based Rowhammer attack variants on ARM. RAMPAGE consists of (1) a root exploit, and (2) a series of app-to-app attacks.
- We introduce GUARDION, a software-based defense that prevents DMA-based Rowhammer attacks. GUARDION is simple, efficient, and has low memory overhead.

In the spirit of open science, we provide our modifications to the Android source code for implementing GUARDION at https://github.com/vusec/guardion.

7.2 Threat Model

We consider an attacker with full control over a zero-permissions holding, unprivileged Android app that is running on the victim's device. She seeks to mount a DMA-based Rowhammer attack, similar to recent work [134], to either (1) escalate her privileges to root, or (2) compromise other apps present on the device. The victim device is hardened against other classes of Rowhammer attacks (e.g., GLitch [50]) and has the latest Android security updates installed.

7.3 Background

This section describes the relevant background information about the Rowhammer vulnerability and its exploitation. This is meant to provide only a brief intro-

duction, for a more in-depth discussion, we point the interested reader to papers exclusively focusing on this topic [76, 106, 146].

7.3.1 The Rowhammer Vulnerability

Rowhammer is a hardware fault in dynamic random-access memory (DRAM) chips. DRAM chips work by storing charges in an array of *cells*. The charge state of a given cell encodes a binary value, a memory bit. Cells are organized in *rows*, which, at the hardware level, is the smallest unit for a memory access. When a memory row is accessed, the content of its cells is copied to a so-called *row buffer*. During this copy operation, the row's cells are discharged, and they are then recharged with their initial values.

Independently from the row access process, memory cells tend to leak their charged state (due to their nature), and their content thus needs to be refreshed regularly. Kim et al. [76] observed that the increasing density of memory chips makes them prone to disturbance errors due to charge leaking into adjacent cells on every memory access. In particular, they show that, by repeatedly accessing, i.e., "hammering," the same memory row (the aggressor row), an attacker can cause enough of a disturbance in a neighboring row (the victim row) to cause bits to flip. The Rowhammer vulnerability is thus a race against the DRAM memory refresh: if an attacker can cause sufficient disturbance, the refresh process may not be fast enough to recharge the cells with their initial values. Kim et al. show that it is possible to flip bits in memory by solely performing software-induced memory read operations, bypassing common memory isolation mechanisms.

7.3.2 Rowhammer Exploitation

Triggering the Rowhammer bug is different than exploiting it. Most bits in memory are irrelevant for an attacker, as flipping them would often just trigger a memory corruption, without obtaining any concrete security advantage. For successful exploitation, the attacker must first land a security-sensitive memory page (e.g., a page owned by the operating system or by another privileged process) into a vulnerable physical memory page. In the general case, software exploitation of this kind is challenging, and, as outlined by van der Veen et al. [134], requires the implementation of the following primitives:

Fast uncached memory access The attacker must access the DRAM chip "fast enough." One of the biggest challenges here is bypassing CPU caches, which, if not handled properly, would "block" any read attempt. The attacker thus needs

to either flush them (to make sure that the next memory read access propagates to DRAM), or use uncached DMA memory to bypass CPU caches altogether.

Physical memory massaging For successful exploitation, the attacker must land a security-sensitive page into a physical memory location that is vulnerable to Rowhammer. This entails that the attacker somehow massages the physical memory so that she can probabilistically [235] or deterministically [114] determine where security-sensitive memory pages would land in physical memory.

Physical memory addressing To make Rowhammer exploitation more practical, the attacker can mount a so-called double-sided Rowhammer attack in which the victim row gets hammered by not one, but both adjacent rows. While this increases the chances of triggering bit flips, it is more challenging: the attacker must either be able to allocate physically contiguous memory, or determine how virtual addresses of an unprivileged process are mapped to physical addresses. In other words, the attacker must determine which virtual addresses map to the two physical rows adjacent to the victim row. We note that, while this primitive is not strictly necessary to implement Rowhammer attacks, its implementation is often required to make these attacks practical.

Security researchers demonstrated that a variety of different system environments are vulnerable to Rowhammer exploitation. Seaborn et al. [235] were the first to demonstrate two practical attacks: one to gain local privilege escalation, and another to escape native client sandboxes. Other researchers then used Rowhammer to bypass in-browser JavaScript sandboxes [19, 59], and even to perform cross-VM attacks [114, 146]. While most work focuses on the x86 platform, Van der Veen et al. show that also ARM-based mobile devices are vulnerable to the Rowhammer bug [134]. This last attack, Drammer, is the most problematic as it does not rely on any special hardware or software features. It shows that it is possible to mount a deterministic privilege escalation technique by relying only on basic memory management functions available in typical modern operating systems that cannot easily be turned off.

7.3.3 Android Memory Management

Android, as any other Linux platform, manages physical memory via the buddy allocator, whose goal is to minimize memory fragmentation [170]. In addition, starting from Android 4.0, Google introduced ION [229], a high-level interface that aims at replacing and unifying the several memory management interfaces exposed by each hardware manufacturer. One of the main features implemented

Table 7.1. Summary of existing defenses and their limitations when deployed to prevent DMA-based Rowhammer attacks on ARM.

Class	Defense	Practical	Secure
¬**flips**	ANVIL [8]	✗	✓†
	B-CATT [161]	✗	✗
	Disabling the contiguous heap [198]	✓	✗
	Pool size reduction [198]	✓	✗
¬**massage**	CATT [22]	✗	✗
	Separation of lowmem/highmem[198]	✓	✗
	Our approach (GUARDION)	✓	✓

†When implemented to monitor DRAM accesses instead of cache misses.

by ION is a number of DMA Buffer Management APIs, which allows userland apps to obtain uncached memory. ION organizes its memory pools in several in-kernel heaps, such as the *kmalloc heap* (`SYSTEM_CONTIG`) and the *system heap* (`SYSTEM`). These heaps allocate memory at different memory locations and, in general, behave differently. For example, van der Veen et al. [134] observed how an app can use the *kmalloc heap* to obtain physically contiguous memory (now disabled by Google [198]), while this is not possible when using the *system heap*.

7.4 Overview of Software-based Defenses

Proposed software-level Rowhammer mitigations try to (1) prevent Rowhammer from triggering bit flips, or (2) prevent massaging of physical memory into an exploitable state (i.e., bit flips in security-sensitive data structures). We now discuss these defenses in more detail and expose their limitations in terms of practicality—*What are the limitations for deploying this technique in practice?*—and security—*Does this technique stop all attacks?* Table 7.1 summarizes our discussion and shows that no previous solution is both practical *and* secure.

7.4.1 Preventing Bit Flips (¬flips)

ANVIL [8] is a two-step mitigation technique that relies on the processor's performance monitoring unit (PMU) to (1) monitor last-level cache misses (LLC misses). If the number of LLC misses per time period exceeds a predefined value, it marks the offending load/store instructions as a potential Rowhammer attack. It then (2) instructs the PMU to also record virtual addresses accessed, and data sources used by those instructions. ANVIL analyzes the results of the latter, and,

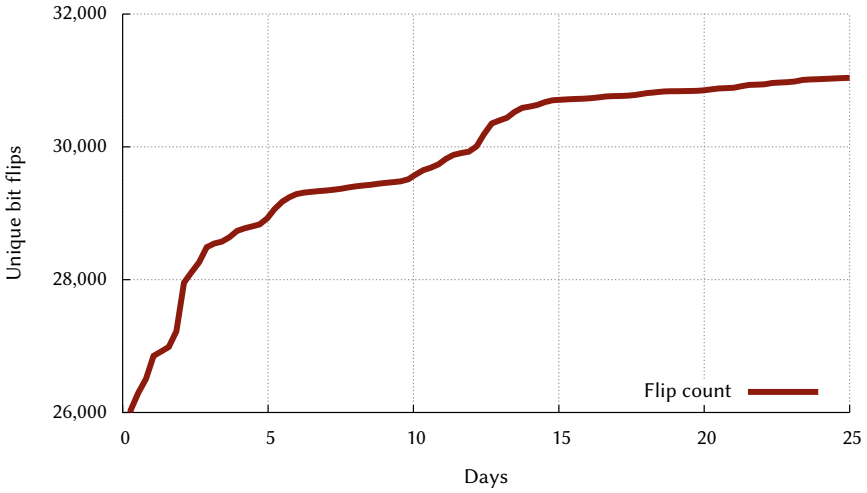


Figure 7.1. Number of unique bit flips found while repeatedly hammering the same 4 MB chunk using double-sided Rowhammer on a Nexus 5 over a time-period of 25 days. Results were obtained by using Drammer's source code [244].

if it concludes that a Rowhammer attack is ongoing, it accesses neighboring rows to force an early refresh, effectively preventing any bit from flipping.

ANVIL could prevent DMA-based Rowhammer attacks by monitoring DRAM accesses instead of LLC misses. Such a defense would be secure, as it would successfully prevent bits from flipping. We were unsuccessful, however, in our search for PMU features on ARM that would allow an efficient implementation of ANVIL's second stage: we were unable to locate any feature that allows us to keep track of which virtual or physical addresses are read from or written to. As such, we conclude that ANVIL is impractical as a mitigation against DMA-based Rowhammer attacks.

B-CATT [161] instructs the bootloader to run a Rowhammer test over the entire physical memory to identify memory pages with vulnerable cells. It then instructs the operating system to mark these pages as unavailable, forcing the system to never use them. This effectively removes the ability of an attacker to induce bit flips.

To evaluate B-CATT's security guarantees, we ran an experiment in which we search for bit flips on a Nexus 5 device by repeatedly performing double-sided Rowhammer on the same 4 MB chunk of contiguous memory. We ran our test for little less than a month while keeping track of each bit flip. Figure 7.1 shows our results: it depicts the number of unique bit flips over time, where a bit flip

is unique if the physical address at which it was reported has not been flipped during an earlier round. Our results show that unique flips indeed do increase over time, and proves that mitigations based on blacklisting vulnerable memory, such as B-CATT, do not scale and are inherently insecure. Furthermore, it shows that any technique that relies on observations and thresholds derived during a testing period, can be subverted as the attacker can trigger different bit flips during runtime. This is on par with related work that discusses the importance of different bit patterns unique to every device when hammering [183, 119]), which makes generic hammering techniques not always effective in finding all vulnerable memory regions.

In parallel, there are many issues that make B-CATT impractical. First, since devices may average close to one bit flip per page [134, 146], B-CATT would have to disable *all* of physical memory for those. Second, blacklisted pages contribute to physical memory fragmentation, making it harder—or impossible—for apps that require physically contiguous memory to run properly. Third, doing a single sweep of a device's physical memory may take over a day to complete—as we experienced when scanning the entire 4 GB of LPDDR4 memory of a Google Pixel, which is in line with observations of related work [146].

Disabling the contiguous heap was Google's first reaction to Drammer [134] and is a third defense that tries to prevent the attacker from flipping bits. In the November 2016 security update for Android, Google disabled the *kmalloc heap*, removing an attacker's primitive to allocate contiguous memory [198]. Without access to the `pagemap` interface—a special file in `procfs` for retrieving physical addresses—this update effectively disables an attacker's ability of performing double-sided Rowhammer, greatly reducing the number of bits she can flip [19].

As this was Google's first attempt at mitigating Drammer, disabling the contiguous heap is proven to work in practice on a variety of devices. We will show in Section 7.5, however, that it is not secure: it possible to implement primitives for obtaining contiguous memory allocations even when using the regular *system heap* (which does not guarantee the allocation of contiguous memory). In concurrent work, Frigo et al. [50] present another side channel for detecting contiguous memory.

Pool size reduction was part of a second round of Drammer mitigations by Google which reduced the number of internal *system heap* pools to two. Before, ION could allocate and pool memory using many different pool sizes (4 KB, 8 KB, 16 KB, …, 4 MB). If one requested a large chunk of memory, say 4 MB, from an empty pool, ION would request these large chunks from the underlying allocator directly, increasing the likelihood for an attacker to obtain physically contiguous

memory. By reducing the maximum pool size to 64 KB, the attacker is more likely to obtain fragmented memory pieces that are not physically contiguous.

Although this is a proven practical solution, it does not eradicate the problem at its root. We show in Section 7.5 how a determined attacker can still force the system to allocate contiguous memory to launch double-sided Rowhammer. Moreover, we show how limiting system allocations to low orders (up to 64 KB) is not effective when memory is not heavily fragmented. In fact, a request for 200 MB would get split up in many 64 KB allocations, some of which will very likely be allocated right next to each other in physical memory.

7.4.2 Preventing Physical Memory Massaging (¬massage)

CATT proposes a static partitioning of physical memory between different security domains [161]. In principle, its design allows for an arbitrary *finite* number of security domains. However, only a prototype for the special case of two security domains was implemented and evaluated: *lowmem* (kernel memory) and *highmem* (user memory). By design, this system guarantees that, under any circumstance, the kernel never touches userland memory, and vice versa.

Modern operating system kernels are designed to make all possible resources available to an app or to the kernel itself. For example, in Linux, the memory management code moves physical memory between zones (e.g., highmem and lowmem) to alleviate memory pressure in them, as a function of the current workload. Due to its *static* partition, CATT severely limits this capability, making it unlikely to be used in practice. Moreover, as acknowledged by the authors, the generalization of CATT to more than two security domains presents a number of significant practicality and complexity challenges. For example, to prevent app-to-app attacks that we will discuss in Section 7.5, CATT must enable as many domains as there are apps installed. To support this, the prototype must be able to pass additional arguments to the kernel's memory allocator to specify the security domain of the process requesting the allocation. This would result in memory fragmentation, which would in turn lead, among other problems, to performance issues: the memory allocator must scan the memory to find a "suitable" memory region for each memory allocation.

Not only is CATT impractical, recent work also demonstrates that so-called *double-ownership* kernel buffers (e.g., video buffers that are shared between user and kernel) allow an attacker to bypass CATT's security guarantees [163].

Separation of highmem/lowmem was part of Google's mitigations against Drammer. Android now enforces that the *system heap*—which is exposed to

userland apps—only returns memory pages from highmem, separating attacker-controlled memory for critical data structures in lowmem.

The highmem/lowmem separation suffers from the same issues as described before. Additionally, we show in the next section that an attacker can allocate many ION chunks to deplete the highmem pool and force the kernel to serve new requests from lowmem. Thus, despite Android's latest security updates, an unprivileged app can still force the system to allocate userland pages in lowmem.

7.5 RAMpage: Breaking the State-of-the-Art

This section elaborates on the security limitations of existing defenses as discussed in the previous section. We document new attack strategies, showing that the defense mechanisms that appear to be practical are not effective for preventing Rowhammer attacks. We first show how it is possible to mount Rowhammer-based attacks even when ION memory allocations are not contiguous and served from highmem. Next, we discuss several app-to-app attack scenarios that show kernel-owned data is not the only target memory to protect.

7.5.1 Exploiting Non-Contiguous Memory

Before Drammer, the ION subsystem allowed userland apps to allocate a large number of contiguous chunks. As described previously, to mitigate Drammer, Google disabled the ION *kmalloc heap*. The ION *system heap*, however, is still available. This heap has two features that make the Drammer attack more challenging: (1) ION allocations from this heap are no longer guaranteed to be physically contiguous, preventing attackers from performing double-sided Rowhammer; (2) the system heap allocates memory from a different zone (highmem, as opposed to lowmem for the kmalloc heap).

We now detail our first RAMPAGE variant, R0: a reliable Drammer implementation that shows how disabling contiguous memory allocations does not prevent Rowhammer-based privilege escalation attacks.

Exhausting the system heap We observe that once ION's internal pools are drained, subsequent allocations are handled directly by the buddy allocator. In this state, we rely on the predictable behavior of the buddy allocator to get contiguous pages [134]. With access to contiguous chunks of memory, we then perform double-sided Rowhammer to find exploitable bit flips. However, as mentioned, the system heap initially allocates memory from highmem while the interesting data structures reside in the lowmem zone. To force lowmem alloca-

tions, we simply continue allocating memory until no highmem is left (which we detect by monitoring `procfs`). Once this is the case, the kernel serves subsequent requests from lowmem, allowing us to find bit flips in physical memory that may later hold a page table.

Shrinking the cache pool Armed with an exploitable bit flip in lowmem, we perform Phys Feng Shui to trick the kernel in storing a page table in the vulnerable page. For this, we need to free the vulnerable row so that the buddy allocator may use it as a page table later. Simply releasing the chunk, however, is not sufficient: after freeing, it ends up in the ION memory pool. We thus require a primitive to shrink system heap pools.

On Android, the low-memory killer (LMK) [228] handles low-memory conditions that arise in the system before the more severe Linux Out-of-Memory (OOM) killer is triggered. The LMK works similarly to the OOM killer, but keeps track of additional information, such as various shrinkers that are available. Shrinkers are registered by memory subsystems or drivers that reserve an amount of memory from the system RAM [191]. When the system is close to running out of memory, the LMK calls the registered shrinkers to release and regain cached memory.

We now construct a primitive to release physical memory of the system heap pools back to the kernel: (1) we read from `/proc/meminfo` to learn how much free memory is available and use this to (2) trigger a `mmap` allocation from userland which is large enough to trigger the LMK. This indirectly forces the ION subsystem to release its preallocated cached memory, including the row with the vulnerable page.

Rooting a mobile device By combining our primitives with the Phys Feng Shui methodology of Drammer [134], we implement the remaining steps of the attack (i.e., finding exploitable chunks and landing page tables in vulnerable locations) and develop a root exploit. We were successful in mounting our proof of concept against an LG G4 running the latest version of Android (7.1.1. at the time of our experiments).

The implementation of these steps involves solving a number of engineering challenges that, from a conceptual point of view, are similar to what was presented in Drammer.

7.5.2 Exploiting System-wide Isolation

In this section, we detail how defense solutions that only protect specific parts of system memory (e.g., the CATT prototype) do not provide a comprehensive protection mechanism. We present three more RAMPAGE variants that illustrate how one can bypass these defenses.

ION-to-ION (R1) In this scenario, we use ION allocations to corrupt ION buffers that belong to another app or process. We start with allocating ION memory to search for exploitable bit flips. Next, we release the vulnerable page to which this bit belongs so that our victim may reuse it. Depending on our victim process, we must then either wait for it to allocate ION memory, or we can trigger allocations by sending an app-specific *intent*.

To investigate the feasibility of this attack, we developed a proof-of-concept in which we trigger bit flips in ION memory that is in use by a victim process. During a real attack, an attacker will likely target a privileged app, such as the media server, in which case she must investigate what bits are sensitive to flip. We acknowledge that it may not be trivial to perform such an attack, and we believe this to be an interesting direction for future research. However, we argue that this scenario and our proof of concept provide a concrete example showing how current defense mechanisms are not comprehensive enough.

CMA-to-CMA attack (R2) The Contiguous Memory Allocator (CMA) is another kernel mechanism to implement DMA-like primitives [189, 190, 207] and thus provides another venue for attackers. Mounting CMA attacks is technically more challenging since it uses a bit map for deciding allocations: depending on the internal state of the map, the victim may not get the same chunk of memory that the attacker releases after the templating, i.e., the probing for vulnerable memory locations. However, the attacker can exhaust the CMA bit map before releasing the vulnerable range to ensure that the victim reuses a vulnerable chunk.

CMA-to-system attack (R3) Although challenging, it is also possible to corrupt system memory from CMA-allocated memory, leading to our last RAMPAGE variant. In fact, the buddy allocator is designed to migrate pages from the CMA heap when the system is close to out-of-memory situations. These pages can be claimed back at any time by the CMA heap (in other words, they are movable). We note that this attack cannot directly target page tables (because they are unmovable); however, the attacker might be able to target other sensitive system-owned data structures (e.g., `struct cred`).

7.6 GuardION: Fine-grained Memory Isolation

As discussed, the main reason for which defenses fail in practice is because they aim to protect *all* sensitive information by making sure that they are not affected by Rowhammer bit flips. Hence, they are either impractical or they miss cases (e.g., variants R1-R3). As RAMPAGE shows, ARM devices are still widely exposed, and providing an adequate software protection is particularly pressing.

We propose GUARDION, a mitigation against DMA-based Rowhammer exploits on mobile devices. Instead of trying to protect all physical memory, we focus on limiting the capabilities of an attacker's uncached allocations. As we will show in Section 7.7, these only constitute a small fraction of all allocations in the system. We can hence afford to apply expensive fine-grained isolation for *each* DMA allocation. GUARDION isolates such buffers with two *guard rows*, one at the 'top' (the first n bytes before an allocation), and another at the 'bottom' (n additional bytes starting at the last address of the allocation). This enforces a strict containment policy in which bit flips that are triggered by reading from uncached memory cannot occur outside the boundaries of that DMA buffer. In effect, this design defends against Rowhammer by eradicating the ability of the attacker to inject bit flips in sensitive data.

Note that GUARDION works under the assumption that bit flips never occur in memory pages that are physically more than one row 'away' from the aggressor rows. This is in the same spirit as other defenses and we believe a sane assumption: such flips have never been reported before, and the electrical properties of Rowhammer make this unlikely to ever occur. Additionally, our current prototype assumes that physical addresses are linearly mapped to DRAM addresses. While this is true for most ARM-based chipsets [106, 134], a next version of GUARDION should use a kernel allocator that is aware of DRAM geometry.

We now describe our implementation of this fine-grained isolation for the Android kernel. Specifically, we modify three allocators that potentially hand contiguous uncached memory to userland apps: the ION contiguous heap, the ION system heap, and the contiguous memory allocation heap (i.e., the CMA heap). In all cases, we need modifications in the allocation and deallocation routines, which we now discuss in more detail.

7.6.1 Isolating ION's Contiguous Heap

Google disabled ION's contiguous heap, the *kmalloc heap* (`SYSTEM_CONTIG`), as part of their efforts to thwart the Drammer attack. This was possible since most devices do not require physically contiguous memory allocations to be available

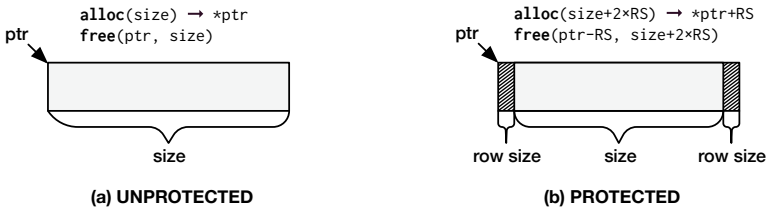


Figure 7.2. Allocations on ION's contiguous heap (a) without and (b) with GUARDION (RS = row size).

for regular userland apps. Device configurations that do require this, however, remain exposed. Since isolating these allocations is simple, we explore them first before describing our more elaborate efforts for isolating ION's system and CMA heaps.

For each request, the contiguous heap allocator takes the requested size and computes the smallest buddy order that satisfies this. The allocator then requests the required number of pages from that buddy order. To free a previously allocated buffer, the allocator simply returns the pages back to the buddy allocator. To isolate bit flips in these buffers, we allocate two guard rows that sandwich the allocation as shown in Figure 7.2. At allocation time for a given size s, we request two extra rows, i.e., $s + 2 \times RS$ (RS being the row size), and return the buffer starting after the guard row to the user. Note that the user process will not have access to these guard rows, as they are never mapped to virtual memory.

For this to work, we need to round up the allocation size to at least the row size. Hence, to protect a 4 KB buffer, we need to allocate 3×64 KB (assuming a row size of 64 KB). Fortunately, at runtime, many requested buffers have a larger size, amortizing the overhead of guard rows. Further, given that DMA buffers constitute a small fraction of an entire app's memory, this overhead becomes negligible as we will show in Section 7.7.

7.6.2 Isolating ION's System Heap

There are two main limitations with ION's contiguous heap: (1) it is not possible to satisfy requests if physically contiguous memory is not available due to fragmentation, and (2) the interaction with the buddy allocator is expensive. To address these limitations, the *system heap* (SYSTEM) provides its users with virtually contiguous memory backed by memory pools of various sizes.

Figure 7.3(a) shows how the system heap satisfies an allocation of a given size by stitching multiple smaller physically contiguous allocations together. These

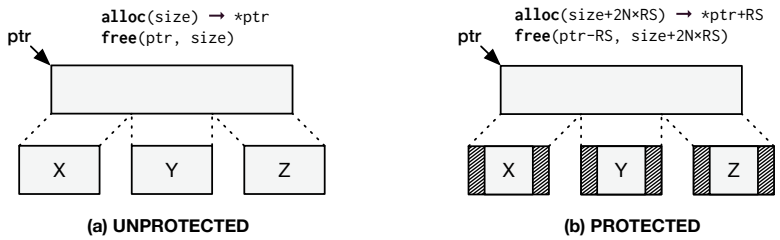


Figure 7.3. Allocations on ION's system heap (a) without and (b) with GUARDION (N = number of pool members, size = X+Y+Z, RS = row size).

smaller allocations are satisfied from pools with pre-defined sizes. The system heap makes an attempt to use pools of the largest suitable size before resorting to pools with smaller sizes to reduce management overhead for each allocation. These pools act as a cache of the buddy allocator in order to improve the allocation performance. Whenever the system is under memory pressure, free memory from these pools is reclaimed and given back to the buddy allocator.

Currently, in order to thwart Drammer, Android only enables pools of size 4 KB and 64 KB, instead of previously-supported larger sizes. Since the size of a memory row is usually 64 KB on ARM, it was expected that attackers cannot allocate large-enough physically contiguous buffers to perform the templating step reliably. We showed that this is not the case in Section 7.4 and we now discuss how we can protect the system heap with GUARDION.

We extend our design for the contiguous heap to protect each physically contiguous allocation from each pool, as depicted in Figure 7.3(b). We extended the pool allocation and deallocation routines to isolate every pool member, similar to how we isolated each allocation from the contiguous heap. Our modifications are mostly straightforward, given that we do not introduce an additional state in the pool allocator. During `free` operations, we return the extra guard rows back to the system.

The overhead of isolating uncached allocations in the system heap depends on the number of allocations from the pools for each request. Given that we are isolating each sub-allocation, we now safely re-enable pools with larger sizes. On top of reducing per-allocation management overhead in the system heap, enabling larger pools reduces the overhead of GUARDION given that it reduces the number of sub-allocations for each request.

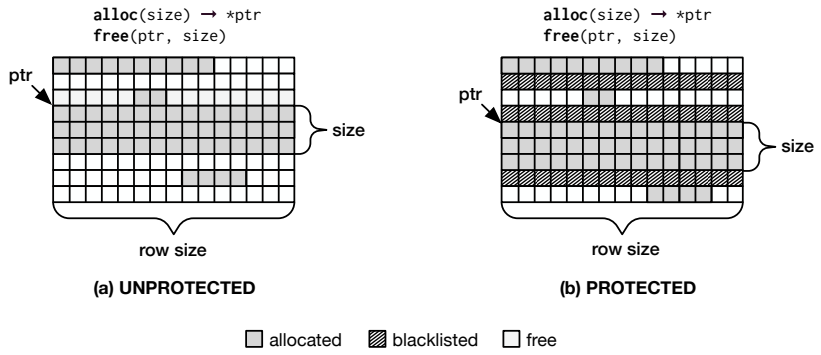


Figure 7.4. Allocations on ION's CMA heap (a) without and (b) with GUARDION.

7.6.3 Isolating ION's CMA Heap

While it was possible to disable ION's contiguous heap for newer mobile devices, there are still drivers that may require physically contiguous memory allocations. These allocations, however, mostly happen in the kernel and are handled by the *CMA heap* (`CMA`). The CMA heap has a statically-defined size which is reserved in physical memory at boot time. These pages may be used by other users when necessary but can always be claimed back by the kernel. While the CMA heap is currently only used by the kernel, we found that recent Android versions still expose it to unprivileged apps. Although we did not find any userland app on a Google Pixel that requires it, we still implement isolation for this heap to provide complete protection.

Figure 7.4(a) shows how the CMA allocator handles requests using a bit map that tracks free memory in the CMA region. The CMA allocator scans this bit map to find the first fit for a requested allocation size. This means that over time, this bit map gets fragmented. To provide isolation in this heap, we follow the following strategy: we blacklist all odd rows in the bit map during initialization. This provides isolation for all allocations that are smaller than the size of the row. To support allocations larger than the row size, we scan the bit map to find a first fit assuming we can allocate blacklisted rows. We use a secondary bit map for the rows to keep track of odd rows that are allocated as part of a large allocation and maintain it during free operations of these large allocations. Figure 7.4(b) shows an isolated allocation from the CMA heap with GUARDION in place.

7.7 Evaluation

We now evaluate GUARDION under several aspects: security, performance, and ease of adoption.

7.7.1 Security Evaluation

GUARDION provides an isolation primitive that makes it impossible for attackers to use uncached DMA allocations to flip bits in memory that is in use by the kernel or any userland app. Within our threat model, where attacks are only possible by attacking uncached memory, GUARDION protects all known Rowhammer attack vectors, and, to the best of our knowledge, no existing technique can bypass it. We verify this by mounting the exploits detailed in Section 7.5 which all failed: we were unable to flip bits in the memory of another process.

7.7.2 Performance and Memory Footprint

We now evaluate the overhead of GUARDION, focusing on both performance and memory overhead.

Dataset To evaluate GUARDION, we execute 22 Android benchmark apps that we selected as follows: (1) we built a dataset of 135 benchmark apps, obtained by searching for the *benchmark* keyword on Google Play; (2) by profiling the ION subsystem, we found that only 28 of them use uncached DMA memory; (3) we discarded two of them as they perform the same tests, one because it does not produce a score, and three because they do not produce reproducible numbers for our baseline.

We run each benchmark app thrice on a Google Pixel running Android 7.1.1 without GUARDION (baseline) and reboot the device after each execution. We then enable GUARDION and repeat this experiment. We compute the median over the three runs and use this as the benchmark score. Since some benchmarks report higher scores for better performance, while for others a lower score indicates better performance, we normalize the scores across benchmarks. Finally, we calculate the geometric mean (geomean) over all benchmark results.

Performance overhead Figure 7.5 shows the overall performance impact of GUARDION. In particular, the figure shows the relative performance compared to the baseline, where a positive value indicates an improvement, while a negative value indicates a performance degradation.

In the worst case, GUARDION results in a performance degradation of 6.6%,

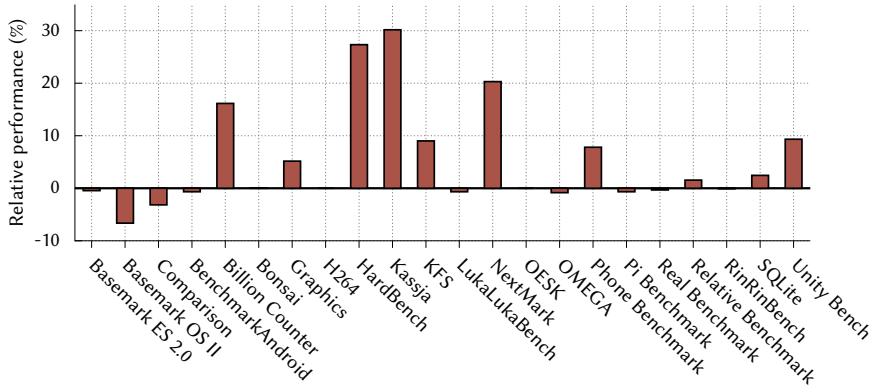


Figure 7.5. GUARDION performance analysis. The numbers show the relative improvement (positive number) or degradation (negative number) of the performance according to each benchmark app.

which we believe is still acceptable. In most cases, however, we see a performance improvement, likely caused by the fact that GUARDION allows us to revert Google's second series of patches, which reduced pool size as outlined in Section 7.4. This allows the ION subsystem to use higher order allocations again in case a process files a request for DMA memory larger than 64 KB: for example, the Kassja benchmark triggers many 2 MB uncached allocations. With large order allocations enabled, each such request triggers the ION subsystem to call the underlying page allocator only once, at most (if the request cannot be processed by the pool). Without GUARDION, however, the request would get split up in $\frac{2\,MB}{64\,KB} = 32$ allocations, each one introducing additional overhead.

Memory overhead Figure 7.6 shows the memory overhead of GUARDION. We determine the memory footprint of an app by modifying the ION subsystem to log every allocation and free operation, including the kernel affected virtual addresses. This way, we can map each allocation to its associated free operation.

In general, memory overhead is negligible, especially when considering that modern devices usually have at least 2 GB of RAM. Interestingly, the RealBench app shows a significant overhead of 46.2 MB, which is much higher than the average. Upon investigation, we determine that this is because the app pressures the uncached DMA, allocating about 190.2 MB during the test.

Impact on UI performance and everyday usage Google measures UI performance of apps mainly in terms of frames per second (fps) and number of "janky,"

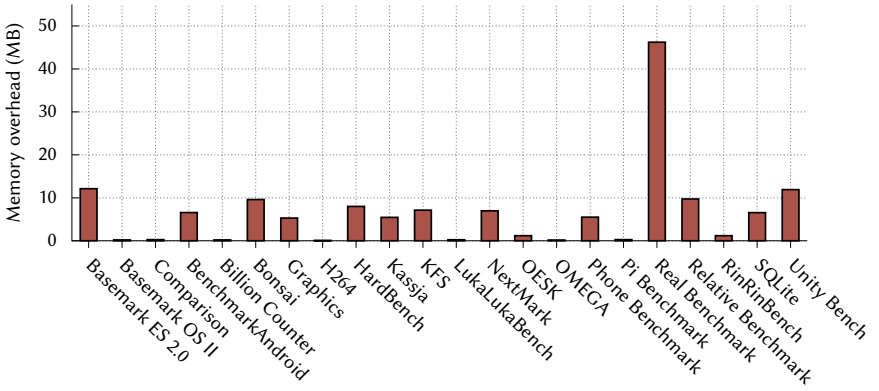


Figure 7.6. Memory overhead (in MB) of DMA isolation with GUARDION.

i.e., delayed or dropped, frames [200]. A consistent rate of 60fps is considered ideal for smooth UI animations. Android provides measurements of these values for a specific app through `adb shell dumpsys gfxinfo <packagename>`. We evaluate the performance of web browsing with Google Chrome on the Google Pixel with and without GUARDION and aggregated the results [241]. Averaged over five runs each, we did not notice any significant differences in UI performance: 22.4 out of 266.8 (8.4%) and 22.0 out of 279.0 (7.9%) frames were janky without and with GUARDION, respectively, with the frame rate remaining constant at 60fps.

Finally, we evaluated the impact of GUARDION on day-to-day device use. For this test, we performed several "everyday" operations on a Google Pixel with and without GUARDION. These operations include taking a photo, shooting and watching a video, watching a video through the YouTube app, making a phone call, making a video call through the Skype app, and browsing the web with Chrome. In all scenarios, we did not notice any difference. Moreover, we did not notice any sign of slowdown or instability.

7.7.3 Patch Complexity and Adoption

We believe that it is easy to integrate GUARDION with the current Android code base. In particular, our prototype implementation for Android 7.1.1 consists of only 844 lines of code. The patch is mostly contained in the ION subsystem, and adds functionality to the bit map data structure of the kernel. Touching only 9 files in the Android source code, it is thus well contained. We also note that a significant part of our patch (422 lines, 5 files) relates to augmenting the bit map

data structure to protect the CMA heap, which we believe should not be exposed to userland apps in the first place (thus possibly reducing the size of our patch even more).

We are currently in the process of submitting our patch to Google, and we hope that Google adopts our proposal either as a security patch for existing versions or in newer versions of Android.

7.8 Related Work

This section provides an overview of related work in the field of Rowhammer exploitation and prevention that were not yet discussed in Section 7.3.2 and Section 7.4.

7.8.1 Rowhammer Attacks

After Kim et al. [76] performed the first systematic study on the Rowhammer hardware fault, and Seaborn et al. [235] demonstrated the first practical attacks, Qiao et al. [112] showed how to use non-temporal access instructions such as `movnti`, to bypass the cache (instead of relying on `clflush`, which was disabled in the Google Native Client browser sandbox following the first Rowhammer attacks). Aweke et al. [8] showed that it is possible to trigger bit flips without using special instructions: they show how an attacker can force the cache to invalidate its content by accessing memory addresses belonging to the same cache eviction set. Another recent technique abuses Intel's Cache Allocation Technology (CAT) to reduce the number of active ways in the last-level cache, which in turn significantly decreases the number of memory accesses required to trigger a bit flip [3]. As discussed in Section 7.3.2, related work has also shown how the Rowhammer vulnerability can be exploited in a number of different scenarios [19, 59, 114, 134, 146].

7.8.2 Rowhammer Defenses

The aforementioned attacks have demonstrated the severity of the Rowhammer vulnerability and prompted the research community to propose a number of defense mechanisms, both in hardware and in software.

Hardware-level defenses One of the most obvious defense mechanism is the production of memory chips that do not suffer from the Rowhammer vulnerability. Kim et al. [76] discuss the various aspects that could be improved: for

example, one could increase the row refresh rate. The DDR3 standard [177] specifies that rows should be refreshed at least every 64 ms, while Kim et al. suggest to refresh the rows at least every 32 ms. Other proposals are Error-Correcting Code (ECC) memory and Target Row Refresh (TRR). One last protection mechanism is PARA [76], which probabilistically activates rows adjacent to a potential victim row.

Unfortunately, all these techniques have significant limitations. First, many of these techniques rely on hardware modifications: ECC and TRR require the production of new memory chips, while PARA requires a change in the memory controller. This makes their deployment less practical, mainly because these chips would be more expensive, would require new development and production pipelines, and they cannot be easily adopted by existing systems, especially mobile devices. Moreover, newer standards such as LPDDR4 [178] already discuss the adoption of TRR, but only as an *optional* protection mechanism, thus leaving LPDDR4 chips still vulnerable to Rowhammer. Protecting mobile devices through hardware protection mechanisms is even more challenging due to the energy consuming nature of these mechanisms and the importance minimizing the device's battery consumption.

Second, these mechanisms are not always effective, even when deployed. For example, Aweke et al. [8] show that they could perform Rowhammer exploitation under 32 ms, making the faster refresh rate ineffective. Instead, mechanisms like ECC memory have the limitations of protecting only from one-bit memory corruption, which is not enough since Kim et al. could induce multiple bit flips. These limitations provided strong incentives to develop software-level defenses.

Software-level defenses The research community started to propose software-based solutions only recently. The first concrete solution is ANVIL [8], which we discussed in Section 7.4. Unfortunately, ANVIL is not applicable to mobile devices. Furthermore, we discussed B-CATT and CATT [22, 161], as well as Google's patches in reaction to Drammer, in Section 7.4, and demonstrated why they are not effective in Section 7.5.

7.9 Conclusion

In recent years, the Rowhammer vulnerability gathered a lot of attention from both the academic and industrial community. While researchers have demonstrated exploits for a range of devices in a variety of settings [19, 59, 76, 114, 235, 134, 146], the Drammer attack on mobile devices [134] is particularly wor-

rying, since it allows for a deterministic attack on very popular systems, by just relying on basic memory management features. Given that it is impossible to perform hardware upgrades on these devices, there is a clear need for effective and efficient software-based defenses.

In this chapter, we showed that existing software mitigations do not solve the problem: they are either impractical to deploy, or do not provide adequate protection. To back our claims, we presented RAMPAGE, a set of DMA-based Rowhammer attacks against the latest Android OS. As a mitigation, we proposed GUARDION, a lightweight, software-only defense to prevent Rowhammer exploitation on mobile devices. Our evaluation shows that GUARDION introduces negligible memory overhead, improves performance compared to Google's mitigation in reaction to previous attacks, and prevents DMA-based Rowhammer attacks, even when considering app-to-app attacks. We release our modifications as open source, and are in the process of sharing our patches with Google, hoping they will adopt our proposal in newer versions of Android.

8 Conclusion

Today, three decades after Robert Morris used one to accidentally break the Internet, memory errors are still one of the primary threats to the security of our systems. Granted, years of security research attribute to the fact that attacks are now more sophisticated than ever — recent Pwn2Own exploits not rarely require a handful of vulnerabilities — they still occur on a regular basis. In this thesis, we studied, and tried to advance computer security defenses that focus on preventing exploits that stem from memory errors. We intersected this domain from two dimensions: (1) we studied code-reuse attacks and defenses, and (2) we dissected the Rowhammer bug and studied its impact on mobile platforms. In summary, this dissertation provides the following key results.

1. **Context-sensitive CFI.** We presented *practical Context-sensitive Control-Flow Integrity* (CCFI). While CCFI can significantly enhance the security of state-of-the-art defenses against control-flow diversion attacks, it has long been perceived as inefficient and impractical for real-world adoption. We showed how we can effectively address the three fundamental challenges towards fast and practical CCFI — efficient path monitoring, analysis, and verification — in a realistic way on commodity platforms.

2. **Forward-edge CFI.** We presented a *forward-edge Control-Flow Integrity* (CFI) and *Control-Flow Containment* (CFC) solution to stop advanced code-reuse attacks. Based on conservative static binary-level analysis to derive both target-oriented and callsite-oriented control-flow invariants, our approach applies strong security policies at runtime without the possibility of breaking the program's original intentions. Our work relies on target-oriented invariants to enumerate legal callsite targets and *detect* attacks that transfer control to illegal targets (akin to traditional CFI, but with much stronger binary-level invariants). In addition, we use callsite-oriented invariants to invalidate illegal function arguments at each callsite and *contain* attacks that rely on type-unsafe function argument reuse: the CFC pro-

tection technique. CFC further improves the quality of our target-oriented invariants, resulting in the strictest binary-level CFI solution to date.

3. **C++ vtable hijacking.** We presented a *practical binary-level defense mechanism against C++ vtable hijacking attacks*. Unlike prior work that restricts the target set of virtual callsites, our approach protects objects *at creation time* and restricts their usage to virtual calls that are reachable by the object. This sidesteps accuracy problems faced by prior work while simultaneously extending the threat model to include use-after-free attacks. Our work provides improved correctness guarantees by handling false positives at vcall verification time. We protect applications from modern C++ code-reuse attacks, including whole-function reuse.

4. **10 years of code-reuse attacks.** We presented a *runtime gadget-discovery framework based on constraint-driven dynamic taint analysis*. We showed that by considering dynamic analysis — opposed to the static analysis that we used for the past decade — even low-effort attackers can find useful defense-aware gadgets to craft practical code-reuse attacks. Our framework found gadgets compatible with state-of-the-art defenses in many real-world programs. We also presented an nginx case study, showing that an attacker armed with our framework can find useful gadgets and craft exploits that comply with the restrictions of strong defenses such as CPI and context-sensitive CFI. Our effort showed that, to sufficiently reduce the attack surface against a dynamic attack model, we must combine multiple state-of-the-art code-reuse defenses or, alternatively, deploy more heavyweight defenses at the cost of higher overhead.

5. **Mobile Rowhammer attacks.** We presented *deterministic Rowhammer attacks on mobile platforms*. We demonstrated that powerful *deterministic* Rowhammer attacks that grant an attacker root privileges on a given system are possible, even by only relying on always-on features provided by commodity operating systems. We presented an implementation of our attack on the Android/ARM platform. Not only does our attack show that practical, deterministic Rowhammer attacks are a real threat for billions of mobile users, but it is also the first effort to show that Rowhammer is even possible at all (and reliably exploitable) on any platform other than x86 and with a much more limited software feature set than existing attacks. Moreover, we demonstrated that several devices from different vendors are vulnerable to Rowhammer, showing that practical Rowhammer attacks are a serious threat.

6. **Mobile Rowhammer defenses.** We presented a *practical mitigation of DMA-based Rowhammer attacks on ARM*. By means of concrete examples, we showed that existing, and proposed software mitigations do not fix the problem of Rowhammer exploitation: they are either impractical to deploy, or do not provide adequate protection. We proposed a lightweight, software-only defense to prevent Rowhammer exploitation on mobile devices. Our mitigation works by isolating DMA allocations — those that allow efficient Rowhammer attacks — from other memory, ensuring that bit flips never occur in kernel-level data structures. Our approach is unique in that it does not try to mitigate *all* possible Rowhammer bugs, but rather aims at eliminating the easiest and most pressing attack vector. This allowed us to implement a solution that is fast and does not incur a large memory overhead, making it practical for deployment in real devices.

Future Directions

As we currently experience *the resurgence* of memory errors — the trend of explosive growth in number of reported vulnerabilities — it is clear that we are far from done with them. This dissertation only scratches the surface of memory error attacks and defenses; research on this topic remains relevant until we *at least* reverse the trend and see a *decrease* in the number of issues that are opened every day. There are numerous directions for future research, ranging from *non-control-data attacks* to *low-overhead bounds checkers*, from *legacy systems and patching behavior* to *static and dynamic analysis for vulnerability detection*, and from *type-confusion bugs* to *(full) memory safety solutions*. In the following, we limit ourselves to the topics presented in this dissertation to expose some concrete future research directions.

1. **Context-sensitive CFI polices.** Our binary-level forward-edge CCFI policy is straightforward: we simply propagate function pointers that are passed in call arguments. This stimulates research on more sophisticated policies — which Patharmor can serve as a basis for. Binary-level data flow analysis may be able to better reconstruct where code pointers are defined and what legal paths of function calls can propagate them to indirect callsites. This would help to significantly reduce the number of allowed targets for forward edges in a program's CFG, yielding even better security guarantees than currently available.

   Aside from strong CCFI policies on binaries, source-level protections may also benefit from better Data Structure Analysis (DSA). The current LLVM

DSA design is flow-insensitive and unification-based and thus aggressively merges data-flow information, resulting in overly conservative results when searching for forward-edge context-sensitive policies. Future work should aim for more progressive results, while keeping analysis overhead to a minimum.

2. **Evaluating defenses.** Our gadget-discovery framework evaluates code-reuse defenses that are *only* applicable to general programs. We do not evaluate C++-specific approaches like vtable protection mechanisms. Future work should extend our efforts and incorporate such proposals, yielding a better understanding of the security guarantees of language-specific defenses. Similarly, it may be possible to cover an even wider area of memory error mitigations by also including data-only attacks in our framework.

   Additionally, as one of our conclusions is that we must combine state-of-the-art code-reuse defenses to provide better security guarantees, we should evaluate (1) what the best combinations of code-reuse defenses are, and (2) how their combination affects system performance.

3. **Protecting the Internet of Things**. The Internet of Things, or IoT, is a collective name for embedded devices — basically every device beyond servers, desktops, laptops, and smartphones — that communicate with one another. Characterized by their small size, these gadgets are often low powered and consist of 'exotic' architectures. This makes it hard to port existing memory error mitigations: some embedded systems do not even provide virtual memory, rendering many x86-based defenses useless. If IoT applications continue being written in unsafe low-level languages like C and C++, the estimated 30 billion operable devices in 2020 cause a drastic increase in the memory-error attack surface. Research should focus on addressing such major challenges in securing our future systems.

4. **Pointer Authentication.** ARM recently announced *Pointer Authentication instructions*, an extension to the ARMv8 Instruction Set Architecture (ISA) that allows programs to efficiently encrypt and decrypt pointers in memory. A whitepaper from Qualcomm describes how a return address gets 'encrypted' with a secret key and a *context*. By using the stack pointer (SP) for the latter, return instructions can only return to callsites with a matching SP.

   While this extension is a step towards eliminating code-reuse attacks in hardware, the use-case scenario as proposed in the whitepaper is still vul-

nerable to *replay* attacks: an attacker with arbitrary read capabilities can collect valid authentication codes and use these in a later stage to launch a ROP chain. We envision three directions to improve precision of Pointer Authentication and thus limit the attack surface even further. First, by including the callee's function address in the context, we guarantee that functions must always return to their original caller. Second, as this still allows for in-function ROP chains, we suggest to use unique stack alignments for each callsite in a function, ensuring a unique (SP + caller function) context per callsite. Third, since only a limited number of bits are available for stack alignment, we suggest to use the instruction pointer as a third input to the context: by cloning large functions accross virtual memory, we increase singularity for each callsite.

5. **Large-scale Rowhammer study.** Ever since its discovery, the Rowhammer bug has been used in a variety of attack scenarios, targeting mobile phones, desktop platforms, and even the cloud. It remains unclear, however, how prevalent the Rowhammer-bug is in reality, and how easy it is to trigger bit flips in practice; as we still await the first use of Rowhammer in the wild, all existing exploits never left the academic field. DRAM manufacturers tell us that Rowhammer has been resolved, and that new memory chips should no longer be exposed, but without a widespread analysis, we cannot know for sure.

   Future work should initiate a large-scale study on Rowhammer in uncontrolled environments. Such research should include (1) an evaluation on how we can reverse engineer the required (DRAM) hardware properties — the rowsize and how adresses are mapped to channels, banks, and ranks — in an uncontrolled environment, i.e., without permitting root access and infinite trials; and (2) a detailed breakdown of vulnerable devices and/or DDR modules. Such work would enable affected industrial companies to perform better risk management strategies, helping them to answer the question whether Rowhammer requires more (or less) attention and whether we should consider it to be a viable and pratical attack vector for the future.

6. **Hardware-based Rowhammer mitigations.** Hardware vendors have proposed and are deploying features that are specifically designed to mitigate Rowhammer attacks. Implementing them in hardware has the advantage that performance overhead remains limited, making such designs suitable for daily use. Evaluating said technologies, however is of paramount;

future work should focus on analyzing their security guarantees, and where necessary try to improve them.

We envision three directions for research on (the effectiveness of) hardware-based Rowhammer mitigations. A first direction involves designing a secure error-correcting code (ECC) memory version that protects against Rowhammer — akin to a recently proposed software-based mitigation technique [79]. ECC was originally introduced to protect server systems from bit flips induced by cosmic rays, but is now often suggested as a sound defense against Rowhammer bit flips as well. Recent work, however, shows that an attacker can bypass ECC protection by flipping multiple bits in a 64-bit word, rendering the error-correcting code valid again [32]. A second direction comprehends the evaluation of the Target Row Refresh (TRR) mitigation that is available in recent DDR chips. By keeping track of row activations, TRR detects rows that are being hammered and can refresh them more frequently. Deciding on the threshold, however, is hard: setting it too low will cause an unnecessary increase in power consumption and decrease in performance, while setting it too high may result in not refreshing an attacked row in time. A third direction involves exploring how an additional cache in the memory controller could thwart Rowhammer attacks. Chip makers may decide to add DRAM caches that are physically *just* before physical memory, aiming at improving DMA performance. Such caches have a side-effect for Rowhammer as well since reads from the aggressor rows no longer propagate to DRAM directly. Future Rowhammer attacks may need to reverse engineer such cache's eviction policy to find if eviction-based Rowhammer remains a threat.

References

The references in this thesis are organized in different sections: conference proceedings, (journal) articles, books, technical reports and documentation, online articles, talks, and source code. All online references were archived and are available in the *Internet Archive Wayback Machine*.[1] They were all accessed on June 1, 2018, unless stated otherwise.

Conference Proceedings

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. **A theory of secure control-flow**. In *Proceedings of the* 7^{th} *International Conference on Formal Engineering Methods (ICFEM)*. Nov. 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. **Control-flow integrity**. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2005.

[3] M. T. Aga, Z. B. Aweke, and T. Austin. **When good protections go bad: Exploiting anti-DoS measures to accelerate rowhammer attacks**. In *Proceedings of the* 10^{th} *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. May 2017.

[4] B. Aichinger. **DDR memory errors caused by row hammer**. In *Proceedings of the* 19^{th} *IEEE High Performance Extreme Computing Conference (HPEC)*. Sep. 2015.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. **Preventing memory error exploits with WIT**. In *Proceedings of the* 29^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2008.

[6] P. Akritidis, M. Costa, M. Castro, and S. Hand. **Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors**. In *Proceedings of the* 18^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2009.

[7] D. Andriesse, X. Chen, V. van der Veen, A. Słowińska, and H. Bos. **An in-depth analysis of disassembly on full-scale x86/x64 binaries**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[8] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. **ANVIL: Software-based protection against next-generation rowhammer attacks**. In *Pro-*

[1] https://web.archive.org

ceedings of the 21^{st} *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Apr. 2016.

[9] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. **You can run but you can't read: Preventing disclosure exploits in executable code**. In *Proceedings of the* 21^{st} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2014.

[10] M. Backes and S. Nürnberger. **Oxymoron: Making fine-grained memory randomization practical by allowing code sharing**. In *Proceedings of the* 23^{rd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2014.

[11] A. R. Bernat and B. P. Miller. **Anywhere, any-time binary instrumentation**. In *Proceedings of the* 10^{th} *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Sep. 2011.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar. **Address obfuscation: An efficient approach to combat a broad range of memory error exploits**. In *Proceedings of the* 12^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2003.

[13] S. Bhatkar, R. Sekar, and D. C. DuVarney. **Efficient techniques for comprehensive protection from memory error exploits**. In *Proceedings of the* 14^{th} *USENIX Security Symposium (USENIX SEC)*. Jul. 2005.

[14] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. **Timely rerandomization for mitigating memory disclosures**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[15] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. **Hacking blind**. In *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2014.

[16] T. Bletsch, X. Jiang, and V. W. Freeh. **Mitigating code-reuse attacks with control-flow locking**. In *Proceedings of the* 27^{th} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2011.

[17] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. **Jump-oriented programming: A new class of code-reuse attack**. In *Proceedings of the* 6^{th} *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Mar. 2011.

[18] E. Bosman and H. Bos. **Framing signals—a return to portable shellcode**. In *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2014.

[19] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. **Dedup est machina: Memory deduplication as an advanced exploitation vector**. In *Proceedings of the* 37^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[20] D. Bounov, R. G. Kıcı, and S. Lerner. **Protecting C++ dynamic dispatch through VTable interleaving**. In *Proceedings of the* 23^{rd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2016.

[21] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A. Sadeghi. **Leakage-resilient layout randomization for mobile devices**. In *Proceedings of the* 23^{rd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2016.

[22] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi. **CAn't Touch This: Practical and generic software-only defenses against rowhammer attacks**. In *Proceedings of the* 26^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[23] N. Burow, D. McKee, S. A. Carr, and M. Payer. **CFIXX: Object type integrity for C++ virtual dispatch**. In *Proceedings of the* 24^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2018.

[24] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch. **Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques**. In *Proceedings of the* 23^{rd} *International on High-Performance Computer Architecture (HPCA)*. Feb. 2017.

[25] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. **Control-flow bending: On the effectiveness of control-flow integrity**. In *Proceedings of the* 24^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2015.

[26] N. Carlini and D. Wagner. **ROP is still dangerous: Breaking modern defenses**. In *Proceedings of the* 23^{rd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2014.

[27] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. **Return-oriented programming without returns**. In *Proceedings of the* 17^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2010.

[28] X. Chen, H. Bos, and C. Giuffrida. **CodeArmor: Virtualizing the code space to counter disclosure attacks**. In *Proceedings of the* 2^{nd} *IEEE European Symposium on Security and Privacy (EuroS&P)*. Apr. 2017.

[29] X. Chen, A. Słowińska, D. Andriesse, H. Bos, and C. Giuffrida. **StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries**. In *Proceedings of the* 22^{nd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2015.

[30] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. **ROPecker: A generic and practical approach for defending against ROP attacks**. In *Proceedings of the* 21^{st} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2014.

[31] T. Chiueh and F. Hsu. **RAD: A compile-time solution to buffer overflow attacks**. In *Proceedings of the* 21^{st} *International Conference on Distributed Computing Systems (ICDCS)*. Apr. 2001.

[32] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos. **Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks**. In *Proceedings of the* 40^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2019.

[33] A. Coletta, V. van der Veen, and F. Maggi. **DroydSeuss: A mobile banking trojan tracker - short paper**. In *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016.

[34] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A. Sadeghi. **Losing control: On the effectiveness of control-flow integrity under stack attacks**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[35] M. L. Corliss, E. C. Lewis, and A. Roth. **Using DISE to protect return addresses from attack**. In *Proceedings of the Workshop on architectural support for security and anti-virus (WASSA)*. Mar. 2005.

[36] C. Cowan, C. Pu, D. Maier, H. Hintongif, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. **StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks**. In *Proceedings of the* 7^{th} *USENIX Security Symposium (USENIX SEC)*. Jan. 1998.

[37] S. Crane, A. Homescu, and P. Larsen. **Code randomization: Haven't we solved this problem yet?** In *Proceedings of the 2016 IEEE Cybersecurity Development Conference (SecDev)*. Nov. 2016.

[38] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. **Readactor: Practical code randomization resilient to memory disclosure**. In *Proceedings of the* 36^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[39] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. **It's a TRaP: Table randomization and protection against function-reuse attacks**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[40] J. Criswell, N. Dautenhahn, and V. Adve. **KCoFI: Complete control-flow integrity for commodity operating system kernels**. In *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2014.

[41] T. H. Dang, P. Maniatis, and D. Wagner. **The performance cost of shadow stacks and stack canaries**. In *Proceedings of the* 10^{th} *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Apr. 2015.

[42] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose. **Isomeron: Code randomization resilient to (just-in-time) return-oriented programming**. In *Proceedings of the* 22^{nd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2015.

[43] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. **Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection**. In *Proceedings of the* 23^{rd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2014.

[44] L. Davi, A. Sadeghi, and M. Winandy. **Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks**. In *Proceedings of the* 4^{th} *ACM Workshop on Scalable Trusted Computing (STC)*. Nov. 2009.

[45] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. **Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks**. In *Proceedings of the* 6^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jul. 2009.

[46] M. Elsabagh, D. Fleck, and A. Stavrou. **Strict virtual call integrity checking for C++ binaries**. In *Proceedings of the* 12^{th} *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Apr. 2017.

[47] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. **XFI: Software guards for system address spaces**. In *Proceedings of the* 7^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2006.

[48] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. C. Rinard, and H. Okhravi. **Missing the point(er): On the effectiveness of code pointer integrity**. In *Proceedings of the* 36^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[49] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. C. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. **Control jujutsu: On the weaknesses of fine-grained control flow integrity**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[50] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. **Grand pwning unit: Accelerating microarchitectural attacks with the gpu**. In *Proceedings of the* 39^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2018.

[51] R. Gawlik and T. Holz. **Towards automated integrity protection of C++ virtual function tables in binary programs**. In *Proceedings of the* 30^{th} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2014.

[52] X. Ge, W. Cui, and T. Jaeger. **GRIFFIN: Guarding control flows using intel processor trace**. In *Proceedings of the* 22^{nd} *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Apr. 2017.

[53] X. Ge, M. Payer, and T. Jaeger. **An evil copy: How the loader betrays you**. In *Proceedings of the* 24^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2017.

[54] J. Gionta, W. Enck, and P. Larsen. **Preventing kernel code-reuse attacks through disclosure resistant code diversification**. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS)*. Oct. 2016.

[55] J. Gionta, W. Enck, and P. Ning. **HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities**. In *Proceedings of the* 5^{th} *ACM Conference on Data and Application Security and Privacy (CODASPY)*. Mar. 2015.

[56] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. **Enhanced operating system security through efficient and fine-grained address space randomization**. In *Proceedings of the* 21^{st} *USENIX Security Symposium (USENIX SEC)*. Aug. 2012.

[57] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. **Out of control: Overcoming control-flow integrity**. In *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2014.

[58] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. **Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard**. In *Proceedings of the* 23^{rd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2014.

[59] D. Gruss, C. Maurice, Stefan, and Mangard. **Rowhammer.js: A remote software-induced fault attack in javascript**. In *Proceedings of the* 13^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jul. 2016.

[60] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. **PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace**. In *Proceedings of the* 7^{th} *ACM Conference on Data and Application Security and Privacy (CODASPY)*. Mar. 2017.

[61] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. **ShrinkWrap: VTable protection without loose ends**. In *Proceedings of the* 31^{st} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2015.

[62] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. **ILR: Where'd my gadgets go?** In *Proceedings of the* 33^{rd} *IEEE Symposium on Security and Privacy (S&P)*. May 2012.

[63] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. **Profile-guided automated software diversity**. In *Proceedings of the* 11^{nd} *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2013.

[64] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. **Automatic generation of data-oriented exploits**. In *Proceedings of the* 24^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2015.

[65] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. **Data-oriented programming: On the expressiveness of non-control data attacks**. In *Proceedings of the* 37^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[66] D. Jang, Z. Tatlock, and S. Lerner. **SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks**. In *Proceedings of the* 21^{st} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2014.

[67] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. **Cyclone: A safe dialect of c**. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC)*. Jun. 2002.

[68] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. **Recovering C++ objects from binaries using inter-procedural data-flow analysis**. In *Proceedings of the* 3^{rd} *ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*. Jan. 2014.

[69] R. W. M. Jones and P. H. J. Kelly. **Backwards-compatible bounds checking for arrays and pointers in c programs**. In *Proceedings of the* 3^{rd} *International Workshop on Automated Debugging (AADEBUG)*. May 1997.

[70] O. Katz, N. Rinetzky, and E. Yahav. **Statistical reconstruction of class hierarchies in binaries**. In *Proceedings of the* 23^{rd} *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Mar. 2018.

[71] O. Katz, R. El-Yaniv, and E. Yahav. **Estimating types in binaries using predictive modeling**. In *Proceedings of the* 43^{rd} *ACM Symposium on Principles of Programming Languages (POPL)*. Jan. 2016.

[72] G. S. Kc, A. D. Keromytis, and V. Prevelakis. **Countering code-injection attacks with instruction-set randomization**. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2003.

[73] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. **ret2dir: Rethinking kernel isolation**. In *Proceedings of the* 23^{rd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2014.

[74] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. **libdft: Practical dynamic data flow tracking for commodity systems**. In *Proceedings of the* 8^{th} *ACM Conference on Virtual Execution Environments (VEE)*. Mar. 2012.

[75] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. **Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software**. In *Proceedings of the* 22^{nd} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2006.

[76] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. **Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors**. In *Proceedings of the* 41^{st} *International Symposium on Computer Architecture (ISCA)*. Jun. 2014.

[77] V. Kiriansky, D. L. Bruening, and S. Amarasinghe. **Secure execution via program shepherding**. In *Proceedings of the* 11^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2002.

[78] K. Koning, H. Bos, and C. Giuffrida. **Secure and efficient multi-variant execution using hardware-assisted process virtualization**. In *Proceedings of the* 46^{th} *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Jul. 2016.

[79] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi. **Zebram: Comprehensive and compatible software protection against rowhammer attacks**. In *Proceedings of the* 13^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2018.

[80] R. K. Konoth, V. van der Veen, and H. Bos. **How anywhere computing just killed your phone-based two-factor authentication**. In *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016.

[81] H. Koo and M. Polychronakis. **Juggling the gadgets: Binary-level code randomization using instruction displacement**. In *Proceedings of the* 11^{nd} *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. May 2016.

[82] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan. **Tackling the path explosion problem in symbolic execution-driven test generation for programs**. In *Proceedings of the* 19^{th} *IEEE Asian Test Symposium (ATS)*. Dec. 2010.

[83] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. **Code-pointer integrity**. In *Proceedings of the* 11^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2014.

[84] C. Lattner, A. Lenharth, and V. Adve. **Making context-sensitive points-to analysis with heap cloning practical for the real world**. In *Proceedings of the* 28^{th} *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. Jun. 2007.

[85] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. **ANDRUBIS - 1,000,000 apps later: A view on current Android malware behaviors**. In *Proceedings of the* 3^{rd} *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. Sep. 2014.

[86] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard. **ARMageddon: Cache attacks on mobile devices**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[87] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. **Flikker: Saving DRAM refresh-power through critical data partitioning**. In *Proceedings of the* 16^{th} *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Mar. 2011.

[88] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. **Transparent and efficient CFI enforcement with Intel Processor Trace**. In *Proceedings of the* 23^{rd} *International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017.

[89] K. Lu, S. Nürnberger, M. Backes, and W. Lee. **How to make ASLR win the clone wars: Runtime re-randomization**. In *Proceedings of the* 23^{rd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2016.

[90] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. **ASLR-Guard: Stopping address space leakage for code reuse attacks**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[91] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. **Pin: Building customized program analysis tools with dynamic instrumentation**. In *Proceedings of the* 26^{th} *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. Jun. 2005.

[92] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. **CCFI: Cryptographically enforced control flow integrity**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[93] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. **SoftBound: Highly compatible and complete spatial memory safety for c**. In *Proceedings of the* 30^{th} *ACM*

SIGPLAN conference on Programming Language Design and Implementation (PLDI). Jun. 2009.

[94] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. **CETS: Compiler-enforced temporal safety for c**. In *Proceedings of the* 9^{th} *International Symposium on Memory Management (ISMM)*. Jun. 2010.

[95] S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Schmiedecker, and E. Weippl. **Enter sandbox: Android sandbox comparison**. In *Proceedings of the* 3^{rd} *IEEE Mobile Security Technologies Workshop (MoST)*. May 2014.

[96] B. Niu and G. Tan. **Monitor integrity protection with space efficiency and separate compilation**. In *Proceedings of the* 20^{th} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2013.

[97] B. Niu and G. Tan. **Modular control-flow integrity**. In *Proceedings of the* 35^{th} *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. Jun. 2014.

[98] B. Niu and G. Tan. **RockJIT: Securing just-in-time compilation using modular control-flow integrity**. In *Proceedings of the* 21^{st} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2014.

[99] B. Niu and G. Tan. **Per-input control-flow integrity**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[100] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. **Poking holes in information hiding**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[101] V. Pappas, M. Polychronakis, and A. D. Keromytis. **Smashing the gadgets: Hindering return-oriented programming using in-place code randomization**. In *Proceedings of the* 33^{rd} *IEEE Symposium on Security and Privacy (S&P)*. May 2012.

[102] V. Pappas, M. Polychronakis, and A. D. Keromytis. **Transparent ROP exploit mitigation using indirect branch tracing**. In *Proceedings of the* 22^{nd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2013.

[103] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. **MARX: Uncovering class hierarchies in C++ programs**. In *Proceedings of the* 24^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2017.

[104] A. Pawlowski, V. van der Veen, D. Andriesse, E. van der Kouwe, T. Holz, and C. Giuffrida. **VPS: Excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching**. In *Proceedings of the* 35^{th} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2019.

[105] M. Payer, A. Barresi, and T. R. Gross. **Fine-grained control-flow integrity through binary hardening**. In *Proceedings of the* 12^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jul. 2015.

[106] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. **DRAMA: Exploiting DRAM addressing for cross-CPU attacks**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[107] P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter. **Code pointer masking: Hardening applications against code injection attacks**. In *Proceedings of the* 8^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jul. 2011.

[108] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. **Comprehensive shellcode detection using runtime heuristics**. In *Proceedings of the* 26^{th} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2010.

[109] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis. **kRˆX: Comprehensive kernel protection against just-in-time code reuse**. In *Proceedings of the* 12^{th} *ACM European Conference on Computer Systems (EuroSys)*. Apr. 2017.

[110] A. Prakash, X. Hu, and H. Yin. **vfGuard: Strict protection for virtual function calls in COTS C++ binaries**. In *Proceedings of the* 22^{nd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2015.

[111] M. Prasad and T. Chiueh. **A binary rewriting defense against stack based buffer overflow attacks**. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC)*. Jun. 2003.

[112] R. Qiao and M. Seaborn. **A new approach for rowhammer attacks**. In *Proceedings of the* 9^{th} *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. May 2016.

[113] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. **VUzzer: Application-aware evolutionary fuzzing**. In *Proceedings of the* 24^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2017.

[114] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. **Flip Feng Shui: Hammering a needle in the software stack**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[115] B. G. Roth and E. H. Spafford. **Implicit buffer overflow protection using memory segregation**. In *Proceedings of the* 6^{th} *International Conference on Availability, Reliability and Security (ARES)*. Aug. 2011.

[116] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A. Sadeghi, and H. Okhravi. **Address oblivious code reuse: On the effectiveness of leakage resilient diversity**. In *Proceedings of the* 24^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2017.

[117] O. Ruwase and M. S. Lam. **A practical dynamic buffer overflow detector**. In *Proceedings of the* 11^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2004.

[118] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. **Vtpin: Practical vtable hijacking protection for binaries**. In *Proceedings of the* 32^{nd} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2016.

[119] A. Schaller, W. Xiong, M. U. Salee, N. A. Anagnostopoulos, S. Katzenbeisser, and J. Szefer. **Intrinsic rowhammer PUFs: Leveraging the rowhammer effect for improved security**. In *Proceedings of the* 10^{th} *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. May 2017.

[120] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. **Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications**. In *Proceedings of the* 36^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[121] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. **Evaluating the effectiveness of current anti-ROP defenses**. In *Proceedings of the* 17^{th} *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Sep. 2014.

[122] E. J. Schwartz, T. Avgerinos, and D. Brumley. **Q: Exploit hardening made easy**. In *Proceedings of the* 20^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2011.

[123] J. Seibert, H. Okhravi, and E. Söderström. **Information leaks without memory disclosures: Remote side channel attacks on diversified code**. In *Proceedings of the* 21^{st} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2014.

[124] H. Shacham. **The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)**. In *Proceedings of the* 14^{th} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2007.

[125] A. Słowińska, T. Stancescu, and H. Bos. **Howard: A dynamic excavator for reverse engineering data structures**. In *Proceedings of the* 18^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2011.

[126] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. **Just-In-Time code reuse: On the effectiveness of fine-grained address space layout randomization**. In *Proceedings of the* 34^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2013.

[127] M. L. Soffa, K. R. Walcott, and J. Mars. **Exploiting hardware advances for software testing and debugging (NIER track)**. In *Proceedings of the* 33^{rd} *International Conference on Software Engineering (ICSE)*. May 2011.

[128] M. Sun, J. C. S. Lui, and Y. Zhou. **Blender: Self-randomizing address space layout for android apps**. In *Proceedings of the* 19^{th} *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Sep. 2016.

[129] A. Tang, S. Sethumadhavan, and S. Stolfo. **Heisenbyte: Thwarting memory disclosure attacks using destructive code reads**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[130] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. **Enforcing forward-edge control-flow integrity in GCC & LLVM**. In *Proceedings of the* 23^{rd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2014.

[131] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Słowińska, H. Bos, and C. Giuffrida. **Practical context-sensitive CFI**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[132] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. **The dynamics of innocent flesh on the bone: Code reuse ten years later**. In *Proceedings of the* 24^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2017.

[133] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos. **Memory errors: The past, the present, and the future**. In *Proceedings of the* 15^{th} *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Sep. 2012.

[134] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. **Drammer: Deterministic rowhammer attacks on mobile platforms**. In *Proceedings of the* 23^{rd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2016.

[135] V. van der Veen, E. Göktaş, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. **A tough call: Mitigating advanced code-reuse attacks at the binary level**. In *Proceedings of the* 37^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[136] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi. **GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM**. In *Proceedings of the* 15^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jun. 2018.

[137] R. K. Venkatesan, S. Herr, and E. Rotenberg. **Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM**. In *Proceedings of the* 13^{th} *International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2006.

[138] D. Wagner and D. Dean. **Intrusion detection via static analysis**. In *Proceedings of the* 22^{nd} *IEEE Symposium on Security and Privacy (S&P)*. May 2001.

[139] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. **Efficient software-based fault isolation**. In *Proceedings of the* 14^{th} *ACM Symposium on Operating Systems Principles*. Dec. 1993.

[140] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W. Hsu, and Y. Cheng. **ReRanz: A light-weight virtual machine to mitigate memory disclosure attacks**. In *Proceedings of the* 13^{th} *ACM Conference on Virtual Execution Environments (VEE)*. Apr. 2017.

[141] Z. Wang and X. Jiang. **HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity**. In *Proceedings of the* 31^{st} *IEEE Symposium on Security and Privacy (S&P)*. May 2010.

[142] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. **Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code**. In *Proceedings of the* 19^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2012.

[143] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. **No-execute-after-read: Preventing code disclosure in commodity software**. In *Proceedings of the* 11^{nd} *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. May 2016.

[144] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. **Shuffler: Fast and deployable continuous code re-randomization**. In *Proceedings of the* 12^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2016.

[145] Y. Xia, Y. Liu, H. Chen, and B. Zang. **CFIMon: Detecting violation of control flow integrity using performance counters**. In *Proceedings of the* 42^{nd} *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Jun. 2012.

[146] Y. Xiao, X. Zhang, Y. Zhang, and M.-R. Teodorescu. **One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[147] W. Xu and Y. Fu. **Own your android! Yet another universal root**. In *Proceedings of the* 9^{th} *USENIX Workshop on Offensive Technologies (WOOT)*. Aug. 2015.

[148] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. **From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[149] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. **PAriCheck: an efficient pointer arithmetic checker for c programs**. In *Proceedings of the* 5^{th} *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Apr. 2010.

[150] Y. Younan, D. Pozza, F. Piessens, and W. Joosen. **Extended protection against stack smashing attacks without performance loss**. In *Proceedings of the* 22^{nd} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2006.

[151] B. Zeng, G. Tan, and Ú. Erlingsson. **Strato: A retargetable framework for low-level inlined-reference monitors**. In *Proceedings of the* 22^{nd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2013.

[152] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. **VTrust: Regaining trust on virtual calls**. In *Proceedings of the* 23^{rd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2016.

[153] C. Zhang, C. Songz, K. Z. Chen, Z. Chen, and D. Song. **VTint: Protecting virtual function tables' integrity**. In *Proceedings of the* 22^{nd} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2015.

[154] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. **Practical control flow integrity & randomization for binary executables**. In *Proceedings of the* 34^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2013.

[155] H. Zhang, D. She, and Z. Qian. **Android ION hazard: The curse of customizable memory management system**. In *Proceedings of the* 23^{rd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2016.

[156] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. **A platform for secure static binary instrumentation**. In *Proceedings of the* 10^{th} *ACM Conference on Virtual Execution Environments (VEE)*. Mar. 2014.

[157] M. Zhang and R. Sekar. **Control flow integrity for COTS binaries**. In *Proceedings of the* 22^{nd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2013.

[158] M. Zhang and R. Sekar. **Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks**. In *Proceedings of the* 31^{st} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2015.

Articles

[159] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. **Randomized instruction set emulation**. *ACM Transactions on Information and System Security (TISSEC)*. vol. 8. no. 1, pp. 3–40. Feb. 2005.

[160] I. Bhati, M.-T. Chang, Z. Chishti, S. Lu, and B. Jacob. **DRAM refresh mechanisms, penalties, and trade-offs**. *IEEE Transactions on Computers*. vol. 65. no. 1, pp. 108–121. Mar. 2015.

[161] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi. **CAn't touch this: Practical and generic software-only defenses against rowhammer attacks**. *arXiv Computing Research Repository*. vol. abs/1611.08396. Nov. 2016.

[162] B. Buck and J. K. Hollingsworth. **An API for runtime code patching**. *International Journal of High Performance Computing Applications (IJHPCA)*. vol. 14. no. 4, pp. 317–329. Nov. 2000.

[163] Y. Cheng, Z. Zhang, and S. Nepal. **Still hammerable and exploitable: On the effectiveness of software-only physical kernel isolation**. *arXiv Computing Research Repository*. vol. abs/1802.07060. Feb. 2018.

[164] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A. Sadeghi. **Selfrando: Securing the Tor browser against de-anonymization exploits**. *Proceedings on Privacy Enhancing Technologies (PoPETs)*. vol. 2016. no. 4, pp. 454–469. Jul. 2016.

[165] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. **Efficiently computing static single assignment form and the control dependence graph**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. vol. 13. no. 4, pp. 451–490. Oct. 1991.

[166] G. C. Necula, J. Condit, M. Harren, S. Mcpeak, and W. Weimer. **CCured: Type-safe retrofitting of legacy software**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. vol. 27. no. 3, pp. 477–526. May 2005.

[167] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. **Return-oriented programming: Systems, languages, and applications**. *ACM Transactions on Information and System Security (TISSEC)*. vol. 15. no. 1. Mar. 2012.

[168] S. Volckaert, B. Coppens, and B. de Sutter. **Cloning your gadgets: Complete ROP attack immunity with multi-variant execution**. *IEEE Transactions on Dependable and Secure Computing (TDSC)*. vol. 13. no. 4, pp. 437–450. Jul. 2015.

Books

[169] A. E. Anderson and W. J. Heinze. **C++ Programming and Fundamental Concepts**. Prentice Hall, 1992.

[170] M. Gorman. **Understanding the Linux Virtual Memory Manager**. Prentice Hall, 2007.

[171] U. Khedker, A. Sanyal, and B. Karkare. **Data Flow Analysis: Theory and Practice**. CRC Press, 2009.

[172] A. Słowińska. **Using Information Flow Tracking to Protect Legacy Binaries**. Vrije Universiteit Amsterdam, May 2012.

Technical Reports and Documentation

[173] ARM Limited. **ARM architecture reference manual. ARMv7-A and ARMv7-R edition**. 2012.

[174] ARM Limited. **ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile**. 2013.

[175] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. **Itanium C++ ABI (Revision: 1.83)**. http://refspecs.linuxbase.org/cxxabi-1.83.html. 2018.

[176] A. Fog. **Calling conventions for different C++ compilers and operating systems**. http://agner.org/optimize/calling_conventions.pdf. 2018.

[177] JEDEC Solid State Technology Association. **DDR3 SDRAM specification**. JESD79-3F. 2012.

[178] JEDEC Solid State Technology Association. **Low power double data 4 (LPDDR4)**. JESD209-4A. 2015.

[179] Linux kernel documentation. **Transparent hugepage support**. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[180] M. Fossi, E. Johnson, D. Turner, T. Mark, J. Blackbird, D. McKinney, M. K. Low, T. Adams, M. P. Laucht, and J. Gough. **Symantec report on the underground economy**. Symantec. Technical Report. Nov. 2008.

[181] I. Fratric. **Runtime prevention of return-oriented programming attacks**. University of Zagreb. Technical Report. Sep. 2012.

[182] M. Lanteigne. **A tale of two hammers: A brief rowhammer analysis of AMD vs. Intel**. Third I/O Inc. Technical Report. May 2016.

[183] M. Lanteigne. **How rowhammer could be used to exploit weaknesses in computer hardware**. Third I/O Inc. Technical Report. Mar. 2016.

[184] S. Sinnadurai, Q. Zhao, and W.-F. Wong. **Transparent runtime shadow stack: Protection against malicious return address modifications**. National University of Singapore, Singapore-MIT Alliance. Technical Report. Feb. 2004.

Online

[185] S. Andersen and V. Abella. **Part 3: Memory protection technologies**. Sep. 2004. [Online]. Available: http://technet.microsoft.com/en-us/library/bb457155.aspx.

[186] Apple. **About the security content of Mac EFI security update 2015-001**. Jun. 2015. [Online]. Available: https://support.apple.com/en-us/HT204934.

[187] A. Arcangeli. **Transparent hugepage support**. Aug. 2010. [Online]. Available: http://www.linux-kvm.org/images/9/9e/2010-forum-thp.pdf.

[188] L. Campbell. **Exploiting NVMAP to escape the Chrome sandbox - CVE-2014-5332**. Jan. 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/01/exploiting-nvmap-to-escape-chrome.html.

[189] J. Corbet. **Contiguous memory allocation for drivers**. Jul. 2010. [Online]. Available: https://lwn.net/Articles/396702/.

[190] J. Corbet. **A reworked contiguous memory allocator**. Jun. 2011. [Online]. Available: https://lwn.net/Articles/447405/.

[191] J. Corbet. **Smarter shrinkers**. May 2013. [Online]. Available: https://lwn.net/Articles/550463/.

[192] J. Corbet. **Memory protection keys**. May 2015. [Online]. Available: https://lwn.net/Articles/643797/.

[193] dcypher. **And the nominees for the dutch cyber security research award are...** Mar. 2016. [Online]. Available: http://www.ictopen.nl/news/and-the-nominees-for-the-dutch-cyber-security-research-award-are.

[194] dcypher. **Award winners DCSRP 2017 and BCMT 2017 (ICT.OPEN 2017)**. Mar. 2017. [Online]. Available: https://www.dcypher.nl/en/content/award-winners-dcsrp-2017-and-bcmt-2017-ictopen-2017.

[195] dcypher. **Announcement: Nominations for the DCSRP award 2018**. Feb. 2018. [Online]. Available: https://www.dcypher.nl/en/content/announcement-nominations-dcsrp-award-2018.

[196] J. Edge. **Building the kernel with clang**. Sep. 2017. [Online]. Available: https://lwn.net/Articles/734071/.

[197] M. Ghasempour, M. Lujan, and J. Garside. **ARMOR: A run-time memory hot-row detector**. 2015. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/.

[198] Google. **ion: Disable ION_HEAP_TYPE_SYSTEM_CONTIG**. Nov. 2016. [Online]. Available: https://android.googlesource.com/kernel/msm/+/android-6.0.1_r0.134%5C%5E%5C%21/.

[199] Google. **Low RAM configuration**. Dec. 2017. [Online]. Available: https://source.android.com/devices/tech/perf/low-ram.

[200] Google. **Testing UI performance**. Apr. 2018. [Online]. Available: https://developer.android.com/training/testing/performance.

[201] J. Hertz and T. Newsham. **Project triforce: Run AFL on everything!** Jun. 2016. [Online]. Available: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

[202] Hewlett Packard. **Moonshot component pack version 2015.05.0 release notes**. May 2015. [Online]. Available: https://support.hpe.com/hpsc/doc/public/display?docId=c04676483.

[203] C. Lameter. **Light weight event counters V4**. Jun. 2006. [Online]. Available: https://lwn.net/Articles/188327/.

[204] Lenovo. **Row hammer privilege escalation**. Mar. 2015. [Online]. Available: https://support.lenovo.com/us/en/product_security/row_hammer.

[205] Microsoft. **A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003**. Sep. 2006. [Online]. Available: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[206] MITRE Corporation. **Vulnerabilities by type**. Jan. 2018. [Online]. Available: https://www.cvedetails.com/vulnerabilities-by-types.php.

[207] M. Nazarewicz. **A deep dive into CMA**. Mar. 2012. [Online]. Available: https://lwn.net/Articles/486301/.

[208] NYU Tandon School of Engineering. **CSAW'17 Applied Research Winners**. Dec. 2017. [Online]. Available: https://csaw.engineering.nyu.edu/research/csaw17-applied-research-winners.

[209] PaX Team. **Address space layout randomization (ASLR)**. Mar. 2003. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt.

[210] Pwnie Awards LLC. **Nominees for the pwnie awards 2017**. Aug. 2017. [Online]. Available: https://pwnies.com/archive/2017/nominations/.

[211] Pwnie Awards LLC. **Pwnie awards winners**. Aug. 2017. [Online]. Available: https://pwnies.com/archive/2017/winners/.

[212] Pwnie Awards LLC. **Nominees for the pwnie awards 2018**. *Accessed: September 1, 2018.* Aug. 2018. [Online]. Available: http://pwnies.com/nominations/.

[213] Red Hat. **How to use, monitor, and disable transparent hugepages in Red Hat Enterprise Linux 6 and 7?** Sep. 2015. [Online]. Available: https://access.redhat.com/solutions/46111.

[214] M. Salyzyn. **AOSP commit 0549ddb9: "upstream: Pagemap: Do not leak physical addresses to non-privileged userspace**. Nov. 2015. [Online]. Available: https://android-review.googlesource.com/c/kernel/common/+/182766.

[215] SANS. **CWE/SANS top 25 most dangerous software errors**. Jun. 2011. [Online]. Available: https://www.sans.org/top25-software-errors/.

[216] M. Seaborn and T. Dullien. **Exploiting the DRAM rowhammer bug to gain kernel privileges**. Mar. 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[217] S. Semwal. **DMA buffer sharing API guide**. Apr. 2012. [Online]. Available: https://lwn.net/Articles/489703/.

[218] S. Semwal. **dma-buf constraints-enabled allocation helpers**. Oct. 2014. [Online]. Available: https://lwn.net/Articles/615892/.

[219] K. A. Shutemov. **Linux commit ab676b7d: "pagemap: Do not leak physical addresses to non-privileged userspace"**. Mar. 2015. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[220] K. A. Shutemov. **THP-enabled tmpfs/shmem using compound pages**. May 2016. [Online]. Available: https://lwn.net/Articles/687352/.

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

[222] J. Stultz. **Integrating the ION memory allocator**. Sep. 2013. [Online]. Available: https://lwn.net/Articles/565469/.

[223] J. Stultz. **The Android graphics microconference**. Oct. 2013. [Online]. Available: https://lwn.net/Articles/569704/.

[224] Unity. **Mobile (Android) hardware stats**. Mar. 2017. [Online]. Available: http://hwstats.unity3d.com/mobile/cpu-android.html.

[225] V. van der Veen. **Trends in memory errors**. Feb. 2017. [Online]. Available: https://vvdveen.com/memory-errors/.

[226] J. Vander Stoep. **Protecting Android with more Linux kernel defenses**. Jul. 2016. [Online]. Available: https://android-developers.googleblog.com/2016/07/protecting-android-with-more-linux.html.

[227] VMware. **Security considerations and disallowing inter-virtual machine transparent page sharing**. Oct. 2014. [Online]. Available: https://kb.vmware.com/s/article/2080735.

[228] A. Vorontsov. **Android low memory killer vs. memory pressure notifications**. Dec. 2011. [Online]. Available: https://lkml.org/lkml/2011/12/18/173.

[229] T. M. Zeng. **The Android ION memory allocator**. Feb. 2012. [Online]. Available: https://lwn.net/Articles/480055/.

[230] W. Zhiyuan and J. Criswell. **llvm DSA - reproduce the result in PLDI 07 paper**. May 2015. [Online]. Available: http://lists.cs.uiuc.edu/pipermail/llvmdev/2015-5/085390.html.

Talks

[231] L. Abbott. **Lessons from ION**. *Embedded Linux Conference (ELC)*. Apr. 2016.

[232] T. Dullien. **Three things that rowhammer taught me**. *Null Singapore*. Mar. 2016.

[233] N. Herath and A. Fogh. **These are not your grand daddy's CPU performance counters — CPU hardware performance counters for security**. *Black Hat USA*. Aug. 2015.

[234] T. de Raadt. **Exploit mitigation techniques**. *OpenCON*. Nov. 2005.

[235] M. Seaborn and T. Dullien. **Exploiting the DRAM rowhammer bug to gain kernel privileges**. *Black Hat USA*. Aug. 2015.

[236] S. Semwal. **Upstreaming ION features: Issues that remain**. *Linux Plumbers Conference*. Aug. 2015.

[237] C. Tice. **Improving function pointer security for virtual method dispatches**. *GNU Tools Cauldron*. Jul. 2012.

Source Code

[238] B. Aker. **memslap: Load testing and benchmarking a server**. 2013. [Online]. Available: http://docs.libmemcached.org/bin/memslap.html.

[239] Apache Software Foundation. **Apache benchmark**. 2013. [Online]. Available: http://httpd.apache.org/docs/2.0/programs/ab.html.

[240] A. Kopytov. **sysbench: Scriptable database and system performance benchmark**. May 2018. [Online]. Available: https://github.com/akopytov/sysbench.

[241] C. Mabee. **cookie-butter: Python script for making graphics performance charts for an Android app**. Apr. 2016. [Online]. Available: https://github.com/Turnsole/cookie-butter.

[242] G. Rodola. **pyftpdlib: Extremely fast and scalable python ftp server library**. May 2018. [Online]. Available: https://github.com/giampaolo/pyftpdlib.

[243] D. Tucker. **OpenSSH portable regression tests**. Dec. 2014. [Online]. Available: http://www.dtucker.net/openssh/regress.

[244] V. van der Veen. **Drammer: Native binary for testing Android phones for the rowhammer bug**. Oct. 2016. [Online]. Available: https://github.com/vusec/drammer.

[245] B. Zehm. **sendEmail: An email program for sending smtp mail from a command line**. Sep. 2009. [Online]. Available: http://caspian.dotconf.net/menu/Software/SendEmail.

Summary

30 years ago, in November 1988, Robert Morris wreaked havoc on the Internet. Exploiting already known, but poorly patched software vulnerabilities, his *worm* infected 10% of all connected machines. Parts of the Internet were unavailable for days, and costs were estimated to reach millions of dollars. Whether or not accidentally, Morris' actions had immense consequences: not only did he become the first person to be tried and convicted under the 1986 Computer Fraud and Abuse Act, Morris also profoundly shaped the field of computer security. Ever since the events of November 2, 1988, security researchers in both industry and academia have put endless of hours in studying and mitigating vulnerabilities, in particular those of the type that Morris exploited: *memory errors*.

Today, despite three decades of research, memory errors still undermine the security of our systems. Even if we consider only classic buffer overflows — a popular subset of memory error vulnerabilities — this class of memory errors has been lodged in the top-3 of the top 25 most dangerous software errors for years. Experience shows that attackers, motivated nowadays by profit rather than fun, have been effective at finding ways to circumvent protective measures. Many attacks today start with a memory error corruption that provides an initial foothold for further infection.

In this dissertation, we analyze and advance computer security defenses that aim to stop the exploitation of memory errors. We intersect this domain from two core dimensions where such errors occur: in software, and in hardware.

First, we analyze advanced code-reuse attacks — one of the most elaborate types of **software-based** memory error exploitation — and how we can defend legacy binaries against them. One of the most promising ways to mitigate code-reuse attacks is Control-Flow Integrity (CFI). Unfortunately, enforcing it without access to source code is hard in practice, and existing defenses often leave enough wiggle room for an attacker to launch successful exploits. In this dissertation, we propose new binary-level defenses that improve the precision of CFI — reducing

the wiggle room just enough to stop attacks from being successful. Moreover, we explore how much leeway an attacker still has after applying different types of code-reuse defenses, including our own.

Second, we study Rowhammer — a **hardware-based** memory error — and its impact on mobile platforms. This disturbance error is the result of the ever increasing density of memory chips, a necessity to be able to put more and faster DRAM memory in devices. It equips attackers with a powerful primitive: a single bit flip in memory that is not under control of the attacker. Ever since its discovery, Rowhammer-based memory corruption attacks have been used to exploit a variety of ecosystems, including the desktop, browser, and even the cloud. In this work, we show that mobile devices are also susceptible to Rowhammer. We demonstrate how an attacker can leverage this to escalate privileges. Moreover, we propose a lightweight countermeasure that can eradicate the majority of the Rowhammer attack surface.

Samenvatting

30 jaar geleden, in november 1988, richtte Robert Morris een ravage aan op het Internet. Door het uitbuiten van reeds bekende, maar niet-geüpdatete kwetsbaarheden, infecteerde zijn *worm* 10% van alle verbonden machines. Delen van het Internet waren dagen onbereikbaar en kosten werden geschat tot in de miljoenen. Al dan niet per ongeluk, Morris' daden hadden immense gevolgen: niet alleen werd hij de eerste persoon die veroordeeld werd onder de onlangs in werking getreden Wet Computervredebreuk, Morris heeft er vooral voor gezorgd dat het onderwerp computerbeveiliging definitief op de kaart is gezet. Sinds de gebeurtenissen van 2 november 1988 zijn onderzoekers uit zowel het bedrijfsleven als de academische wereld continu bezig met het bestuderen en bestrijden van kwetsbaarheden, in het bijzonder die van het type dat Morris misbruikte: geheugenfouten.

Vandaag de dag, ondanks drie decennia aan onderzoek, ondermijnen geheugenfouten nog steeds de veiligheid van onze systemen. Zelfs als we alleen de klassieke buffer overflow beschouwen — een populaire deelverzameling van geheugenfouten — dan zien we dat deze al jaren geparkeerd staat in de top 3 van de top 25 gevaarlijkste programmafouten. Ervaring leert dat aanvallers effectief zijn in het vinden van nieuwe manieren om beschermende maatregelingen te omzeilen. Veel aanvallen beginnen tegenwoordig met een geheugenfout als een eerste handvat voor verdere infectie.

In deze dissertatie analyseren en bevorderen we computerbeveiliging met als doel misbruik van geheugenfouten te voorkomen. We besnijden dit domein vanuit twee kerndimensies waarin dergelijke fouten kunnen voorkomen: in programmatuur (software) en in computerapparatuur (hardware).

We beginnen met een analyse van zogenaamde "geavanceerde code-hergebruik aanvallen" — een van de doordachtste manieren om geheugenfouten in programmatuur uit te buiten — en hoe we oudere binaire bestanden ertegen kunnen wapenen. Een veelbelovende maatregel tegen code-hergebruik aanvallen is *besturings-*

stroomintegriteit. Helaas is het in de praktijk lastig om besturingsstroomintegriteit toe te passen op programmatuur zonder toegang te hebben tot haar oorspronkelijke broncode; bestaande verdedigingen laten daarom vaak steken vallen waardoor een aanvaller alsnog kwetsbaarheden met succes kan misbruiken. In deze dissertatie stellen we nieuwe verdedigingen voor die de precisie van besturingsstroomintegriteit kunnen verbeteren voor binaire bestanden — we reduceren de speelruimte dermate dat aanvallen niet langer zullen slagen. Tevens kijken we naar hoeveel dergelijke speelruimte een aanvaller nog heeft wanneer verschillende maatregelen tegen code-hergebruik aanvallen worden toegepast, inclusief die van ons zelf.

Vervolgens bestuderen we rijhamer — een geheugenfout in de computerapparatuur — en de impact van rijhamer op mobiele platformen. Deze storingsfout is het resultaat van de steeds maar toenemende dichtheid van geheugenchips, een noodzaak om apparatuur uit te kunnen rusten met meer en sneller geheugen. Rijhamer geeft aanvallers een krachtig primitief: de waarde van een enkele bit in geheugen waar de aanvaller geen toegang tot heeft, kan worden omgegooid. Sinds haar ontdekking is rijhamer gebruikt om geheugenfout-gebaseerde aanvallen uit te voeren op een tal van ecosystemen, inclusief de desktop PC, de browser, en zelfs de cloud. In deze dissertatie laten we zien dat ook mobiele apparaten vatbaar zijn voor rijhamer. We presenteren hoe een aanvaller dit kan uitbuiten om privileges te escaleren. Daarnaast stellen we een goedkope maatregel voor die het grootste deel van het rijhamer-aanvalsoppervlak kan elimineren.