OS-LEVEL ATTACKS AND DEFENSES: FROM SOFTWARE TO HARDWARE-BASED EXPLOITS

Dissertation zur Erlangung des akademischen Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
genehmigt durch den Fachbereich Informatik (FB 20)
der Technischen Universtität Darmstadt

von

DAVID GENS
aus Wiesbaden, Deutschland

Gutachter:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. Dr. Thorsten Holz (Zweitreferent)

Tag der Einreichung: 14. Dezember 2018
Tag der Disputation: 13. Februar 2019

TECHNISCHE
UNIVERSITÄT
DARMSTADT

CYSEC/System Security Lab
Intel Collaborative Research Institute (ICRI-CARS)
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

PHD REFEREES:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st PhD Referee)
Prof. Dr. Thorsten Holz (2nd PhD Referee)

FURTHER PHD COMMISSION MEMBERS:
Prof. Dr. Sebastian Faust
Prof. Dr. Guido Salvaneschi
Prof. Dr.-Ing. Thomas Schneider

Darmstadt, Germany December 2018

ABSTRACT

Run-time attacks have plagued computer systems for more than three decades, with control-flow hijacking attacks such as return-oriented programming representing the long-standing state-of-the-art in memory-corruption based exploits. These attacks exploit memory-corruption vulnerabilities in widely deployed software, e.g., through malicious inputs, to gain full control over the platform remotely at run time, and many defenses have been proposed and thoroughly studied in the past. Among those defenses, control-flow integrity emerged as a powerful and effective protection against code-reuse attacks in practice. As a result, we now start to see attackers shifting their focus towards novel techniques through a number of increasingly sophisticated attacks that combine software and hardware vulnerabilities to construct successful exploits. These emerging attacks have a high impact on computer security, since they completely bypass existing defenses that assume either hardware or software adversaries. For instance, they leverage physical effects to provoke hardware faults or force the system into transient micro-architectural states. This enables adversaries to exploit hardware vulnerabilities from software without requiring physical presence or software bugs.

In this dissertation, we explore the real-world threat of hardware and software-based run-time attacks against operating systems. While memory-corruption-based exploits have been studied for more than three decades, we show that data-only attacks can completely bypass state-of-the-art defenses such as Control-Flow Integrity which are also deployed in practice. Additionally, hardware vulnerabilities such as Rowhammer, CLKScrew, and Meltdown enable sophisticated adversaries to exploit the system remotely at run time without requiring any memory-corruption vulnerabilities in the system's software. We develop novel design strategies to defend the OS against hardware-based attacks such as Rowhammer and Meltdown to tackle the limitations of existing defenses. First, we present two novel data-only attacks that completely break current code-reuse defenses deployed in real-world software and propose a randomization-based defense against such data-only attacks in the kernel. Second, we introduce a compiler-based framework to automatically uncover memory-corruption vulnerabilities in real-world kernel code. Third, we demonstrate the threat of Rowhammer-based attacks in security-sensitive applications and how to enable a partitioning policy in the system's physical memory allocator to effectively and efficiently defend against such attacks. We demonstrate feasibility and real-world performance through our prototype for the popular and widely used Linux kernel. Finally, we develop a side-channel defense to eliminate Meltdown-style cache attacks by strictly isolating the address space of kernel and user memory.

ZUSAMMENFASSUNG

Softwarebasierte Laufzeitangriffe stellen Rechnerplattformen seit mehr als drei Jahrzehnten vor große Sicherheitsprobleme. In Form weit verbreiteter Offensivtechniken wie Return-Oriented Programming, die Programmierfehler durch bösartige Eingaben gezielt ausnutzen, können Angreifer im Extremfall so die vollständige Kontrolle über die Plattform erlangen. Daher wurden über die Jahre eine Vielzahl von Defensivmaßnahmen wie Control-Flow Integrity und Fine-Grained Randomization vorgeschlagen und die Effektivität dieser Schutzmechanismen war lange Zeit Gegenstand intensiver Forschungsarbeit. In jüngster Zeit wurden jedoch eine Reihe zunehmend ausgefeilterer Angriffe auf Hardwareschwachstellen aus Software heraus vorgestellt, die bei Beibehaltung des Angreifersmodells zur kompletten Kompromittierung dieser Systeme führen können. Diese neuartige Entwicklung stellt bestehende Verteidigungen, die traditionelle Softwareangriffe annehmen, somit in der Praxis vor große Herausforderungen.

Diese Dissertation erforscht eine Reihe solcher neuartigen Angriffsszenarien, um die reale Bedrohung auf das Betriebssystem trotz bestehender Verteidigungsmechanismen abschätzen zu können. Insbesondere geht die Arbeit im ersten Teil auf die Problematik sogenannter Data-Only Angriffe im Kontext von Defensivmaßnahmen wie Control-Flow Integrity ein und demonstriert, wie diese unter Ausnutzung von Softwarefehlern vollständig ausgehebelt werden können. Der zweite Teil erforscht die Bedrohung von Laufzeitangriffen durch Hardwarefehler, auch in Abwesenheit von Softwarefehlern auf welche sich bisherige Verteidigungen beschränken. Um die bisherigen Problem in den vorhandenen Schutzmechanismen anzugehen wurden neue Designstrategien entwickelt, mit Hilfe derer sich das Betriebssystem vor solch weiterführenden Angriffen durch geeignete Maßnahmen in Software schützen kann. Zunächst demonstrieren wir eine randomisierungsbasierte Verteidigung gegen Data-Only Angriffe auf Seitentabellen. Des weiteren wird ein Framework zur automatisierten Identifikation von Softwarefehlern im Betriebssystem anhand des Quelltexts auf Basis des LLVM Compilers vorgestellt. Außerdem erforscht die Arbeit eine Absicherung des Betriebbsystems vor Seitenkanalangriffen durch geeignete Isolation des Addressraumes. Ferner entwickeln wir eine Schutzmaßnahme vor Rowhammer-basierten Angriffen auf das OS, indem der physische Speicherallokator des Systems um eine Partitionierungsstrategie erweitert wird.

ACKNOWLEDGMENTS

First and foremost I would like to thank my thesis advisor Professor Ahmad-Reza Sadeghi for his excellent supervision and continuous support. I am grateful to have Professor Thorsten Holz as thesis co-advisor. I thank Professor Sebastian Faust, Professor Guido Salvaneschi, and Professor Thomas Schneider for being members of my thesis committee.

Additional thanks go to Professor Lucas Davi with whom I had the privilege to collaborate and work with while he was still based in Darmstadt and who offered me an initial position in his Junior Research Group funded by DFG's Emmy Noether program.

My research work and travels were funded in part by the Federal Ministry of Education and Research (BMBF) through SAL and by the DFG through CROSSING, both of which provided a great working environment and allowed me to collaborate with renowned and leading researchers from academia and industry worldwide.

Further, I thank all my co-authors besides Ahmad and Lucas: Orlando Arias, Ferdinand Brasser, Poulami Das, Ghada Dessouky, Lisa Eckey, Rolf Egert, Sebastian Faust, Marc Fischlin, Tommaso Frassetto, Jason M. Fung, Patrick Haney, Kristina Hostakova, Sven Jacob, Patrick Jauernig, Yier Jin, Arun Karthik Kanuparthi, Hareesh Khattri, Christopher Liebchen, Garrett Persyn, Jeyavijayan Rajendran, Simon Schmitt, Matthias Senker, Emmanuel Stapf, Dean Sullivan, Jörg Tillmans, and Shaza Zeitouni.

Thanks to all my colleagues of the System Security Lab at Technische Universität Darmstadt for fruitful and interesting technical discussion, as well as their help in organizing courses, and a great time.

Finally, I would like to thank my family and in particular my parents Monika and Andreas for their continuing encouragement. Special thanks to Miriam for her enduring sympathy, support, and friendship.

CONTENTS

I INTRODUCTION

1 OVERVIEW 3
- 1.1 Goals and Scope 5
- 1.2 Summary of Contributions 5
- 1.3 Organization 7
- 1.4 Publications 8

2 BACKGROUND 9
- 2.1 Basic OS-level Defenses 9
  - 2.1.1 Privilege-Level Separation 9
  - 2.1.2 Virtual-Memory Protection 10
- 2.2 The Traditional Threat Models 11
- 2.3 Memory-Corruption Attacks 12
  - 2.3.1 Causes of Memory Corruption 12
  - 2.3.2 A Model for Exploitation 13
  - 2.3.3 Typical Attack Workflow 14
- 2.4 State-of-the-art Attacks and Defenses 15
  - 2.4.1 Return-to-libc and Return-Oriented Programming 15
  - 2.4.2 Code-Reuse Defenses and Advanced Attacks 16
  - 2.4.3 Data-only attacks 17
- 2.5 Modern Hardware Platforms 18
  - 2.5.1 Overall System Architecture 18
  - 2.5.2 Processor Design Principles 19
  - 2.5.3 Remote Hardware Exploits 21

II MEMORY CORRUPTION: THE THREAT OF DATA-ONLY ATTACKS

3 BYPASSING CFI IN MODERN BROWSERS. 25
- 3.1 Data-Only Attacks on Dynamic Code Generation 25
- 3.2 Assumptions and Threat Model 26
- 3.3 Generically Bypassing CFI by Exploiting JIT-Compilers 27
  - 3.3.1 Attack Overview 27
  - 3.3.2 Corrupting the Intermediate Representation 28
  - 3.3.3 Attack Framework 29
- 3.4 On the Prevalence of Data-Only Attacks 29
- 3.5 Possible Mitigations 30

4 BREAKING AND FIXING CFI IN OS KERNELS. 31
- 4.1 OS Kernels as Targets of Data-Only Attacks 31
- 4.2 Assumptions and Threat Model 31
- 4.3 Our Page Table Exploit 32
  - 4.3.1 Attack Overview 32
  - 4.3.2 Exploiting the Kernel by Corrupting the Page Table 33
- 4.4 PT-Rand: Mitigating Data-Only Attacks against the Page Tables 34
  - 4.4.1 Design and Overview 34

4.4.2 Challenges 36
4.5 Implementation 37
4.5.1 Page Table Allocations 37
4.5.2 Generating a Randomized Area 39
4.5.3 Page Table References 39
4.5.4 Handling of the one-to-one mapping 40
4.5.5 Translation of Physical Addresses 40
4.6 Evaluation 41
4.6.1 Methodology 41
4.6.2 Leakage Resilience and Randomization Entropy 41
4.6.3 Performance 42
4.6.4 Compatibility and Stability 44
5 AUTOMATICALLY UNCOVERING MEMORY CORRUPTION IN KERNEL CODE. 47
5.1 Run-time defenses vs. compile-time verification 47
5.2 Data-Flow Analysis 48
5.3 Problem Description 50
5.4 Design of K-Miner 50
5.4.1 Goals and assumptions 51
5.4.2 Overview 51
5.4.3 Uncovering Memory Corruption 52
5.4.4 Challenges 53
5.5 Implementation 54
5.5.1 Global Analysis Context 55
5.5.2 Analyzing Kernel Code Per System Call 56
5.5.3 Minimizing False Positives 57
5.5.4 Efficiently Combining Multiple Analyses 58
5.6 Evaluation 59
5.6.1 Security 59
5.6.2 Performance 61
5.6.3 Usability 62
5.7 Possible Extensions 64
6 RELATED WORK 67
6.1 Data-only Attacks and Defenses for Static Code 67
6.2 Data-only attacks and defenses for JIT code 68
6.3 Kernel and Page Table Protection 69
6.4 Static Analysis Frameworks 71
6.4.1 Kernel Static Analysis Frameworks 71
6.4.2 User Space Static Analysis 72

III REMOTE HARDWARE EXPLOITS: AN EMERGING ATTACK PARADIGM
7 HAMMER TIME: REMOTE ATTACKS ON DRAM AND INITIAL DEFENSES. 77
7.1 DRAM and the Rowhammer Bug 77
7.2 Rowhammer in the Context of Security-sensitive Applications 78
7.3 Rowhammer as an Exploit Primitive 80
7.4 On the Distribution of Bit Flips 82
7.5 Initial Rowhammer Defense 83
8 MITIGATING ROWHAMMER ATTACKS AGAINST OS KERNELS. 85

8.1 On the Necessity of Software Defenses against Rowhammer 85
8.2 Assumptions and Threat Model 86
8.3 Design of CATT 87
8.3.1 Overview 87
8.3.2 Security Domains 88
8.3.3 Challenges 89
8.4 Implementation 89
8.4.1 Mapping Page Frames to Domains 90
8.4.2 Tracking Security Domains 90
8.4.3 Modifying the Physical Page Allocator 91
8.4.4 Defining DRAM Partitioning Policies 91
8.5 Security Evaluation 92
8.5.1 Rowhammer Testing Tool 93
8.5.2 Real-world Rowhammer Exploit 94
8.6 Performance Evaluation 96
8.6.1 Run-time Overhead 96
8.6.2 Memory Overhead 97
8.6.3 Robustness 97
8.7 Discussion 98
8.7.1 Applying CATT to Mobile Systems 98
8.7.2 Single-sided Rowhammer Attacks 98
8.7.3 Benchmarks Selection 98
8.7.4 Vicinity-less Rowhammering 99
9 SIDE-CHANNEL RESILIENT KERNEL-SPACE RANDOMIZATION. 101
9.1 Side-channel attacks against KASLR 101
9.2 Adversary Model and Assumptions 103
9.3 Our Side-Channel Defense for the Kernel 103
9.3.1 Overview 104
9.3.2 Challenges for Fine-grained Address Space Isolation 105
9.4 Prototype Implementation 106
9.4.1 Initialization 107
9.4.2 Interrupts 107
9.4.3 Fine-grained Page Table Switching 108
9.5 Evaluation 108
9.5.1 Security 108
9.5.2 Performance 111
9.6 Discussion 113
9.6.1 Applying LAZARUS to different KASLR implementations 113
9.6.2 Speculative Execution and Side-channel Attacks 114
9.6.3 Other side-channel attacks on KASLR 114
10 THE GROWING PROBLEM OF SOFTWARE-EXPLOITABLE HARDWARE BUGS. 117
10.1 Hardware Vulnerabilities from a Software Perspective 117
10.1.1 The Origin of the Hardware Bug 117
10.1.2 Quality Assurance Techniques for Hardware 118
10.2 Assessing the state-of-the-art 119
10.3 Exploiting Hardware Bugs From Software 119
10.3.1 Threat Model and Assumptions 120

10.3.2 Attack Details 121
11 RELATED WORK 125
11.1 Rowhammer 125
11.1.1 Attacks 125
11.1.2 Defenses 126
11.2 Side-channel attacks against the OS 127
11.2.1 Paging-based Side-channel Attacks on KASLR 127
11.2.2 Software Mitigations 128
11.2.3 Hardware Mitigations 129
11.3 Recent Hardware Exploits 129
IV DISCUSSION AND CONCLUSION
12 DISSERTATION SUMMARY 135
13 FUTURE WORK 137
V APPENDIX
A ABOUT THE AUTHOR 141
B LIST OF FIGURES 145
C LIST OF TABLES 149
BIBLIOGRAPHY 151

Part I

INTRODUCTION

1 OVERVIEW

Computer systems are an integral part of our society. We leverage these systems in controlling large parts of our infrastructure, such as the transportation networks and energy grids. We use computers on a daily basis to communicate with friends and family, and consume, create, and publish media. Today, we even entrust those systems with handling the financial markets and aiding in our democratic processes. Consequently, computing platforms represent an extremely valuable target for adversaries. At the same time computer systems are also highly diverse, ranging from embedded devices and sensors in planes and cars, to smart phones, laptops, desktop computers, and powerful servers running the cloud. To manage this broad array of hardware and abstract the underlying complexity, the Operating System (OS) provides application developers with a software interface for user-space programs. The OS is typically considered part of the system's *trusted computing base* (TCB) [8], and usually runs with elevated privileges to protect the platform from malicious software. To achieve this, the OS enforces strict isolation policies for user programs utilizing dedicated security mechanisms that are provided by the hardware.

However, modern platforms execute a large number of applications and many of these programs are written in low-level languages, which do not provide any memory safety guarantees in the case of an error. This means that bugs in such pieces of software leave computer systems vulnerable to run-time attacks based on memory corruption, e.g., through maliciously crafted inputs. In practice, this allows attackers to overwrite code pointers that are stored in unprotected memory, such as the stack or the heap of the program. One of the earliest examples for such a vulnerability is the *buffer overflow* [9]: many programs use buffers to temporarily store user-provided input such as a string of text characters in memory. If the software developer forgot to properly check the length of the input, a user can overflow this buffer by providing an input that exceeds the originally defined size of the storage. This allows an adversary to corrupt parts of the program memory that are located beyond the storage of the buffer. For instance, by writing beyond the boundaries of a stack buffer an adversary can overwrite code pointers such as return addresses, which are typically stored on the stack as well. This enables the attacker to hijack the control flow of the program and execute code that was not originally intended by the developer in a *code-reuse attack*. In this way, an adversary can in principle achieve arbitrary behavior at run time [10].

Stack-based buffer overflows are one example of a bug class that allows memory corruption. They are part of a much larger family of vulnerabilities that further include heap-based errors, format string vulnerabilities, type confusion, uninitialized data usage, use-after-free and double-free errors, dangling pointers, synchronization errors, and integer overflows [11, 12]. These *memory-corruption vulnerabilities* are introduced due to human error in low-level software code, such as mistakes in memory management or missing corner cases in manually crafted checks. Memory-corruption vulnerabilities allow adversaries to subvert and take control of the affected program through malicious inputs, and hence, continue to pose severe threats for the security of real-world software [13–19].

For this reason, operating systems treat applications as *untrusted* and the kernel is designed to act as a *reference monitor* [20, 21] for user processes. The OS mediates all accesses of application software and enforces a strict separation of privileges to protect the platform. To implement this isolation efficiently and achieve a strong policy enforcement, the OS usually leverages dedicated hardware mechanisms such as virtual memory and privilege separation. However, for legacy and performance reasons all major operating systems are written in low-level programming languages that leave the system vulnerable to memory corruption in case of an error. This means, that a user-space adversary who gained access to the platform through a vulnerable application can target bugs in OS kernel code in a second step, e.g., by launching attacks against memory-corruption vulnerabilities in system calls or drivers in low-level OS code [22–26].

Such multi-staged attacks are increasingly common in practice, as many examples demonstrate the severe impact of memory-corruption vulnerabilities in large and complex code bases of major OS kernels: in 2016 *Dirty CoW* [27] was discovered to be a widely exploited bug in the Linux kernel that also affected all Android-based phones [28]. In 2017 Project Zero demonstrated a kernel exploit for iOS based on a double-free bug in Apple's Mach kernel [29]. Every year we are seeing memory-corruption-based exploits across all vendors and platforms that affect billions of devices [30–39]. This is why a great amount of time and effort has been invested over the recent years to systematically develop efficient mitigations and analyze their effectiveness, with Control-Flow Integrity [40–45], Code-Pointer Integrity [46], and various randomization-based schemes [46–56] representing the current state-of-the-art in defenses. In particular, CFI enforces the control flow of a program by matching the execution against an explicitly defined set of allowed code locations at run time to mitigate code-reuse attacks. Since CFI offers formal security guarantees while incurring only modest performance overheads, it is now in the process of being widely adopted in practice, e.g., through compiler extensions [57, 58], kernel patches [43–45], and hardware support [42, 59–61].

However, all of these defenses assume conventional software attacks in their threat models and software-based defenses implicitly assume the underlying hardware, particularly the processor and the main memory, to be trustworthy. This assumption is now being challenged, as researchers are discovering serious and foundational security vulnerabilities at the hardware level, such as vulnerabilities in memory or processor chips. Indeed, we are witnessing an unexpected shift in the traditional attack paradigms through hardware-based exploits. In this emerging threat the attacker exploits a vulnerability at the hardware level by triggering it from software at run time. Various hardware bugs have been shown to affect a wide range of platforms, manufacturers, and vendors, and a number of real-world exploits have been demonstrated recently [62–67]. So far, state-of-the-art defenses are entirely oblivious to hardware vulnerabilities, and hence, completely bypassed by these upcoming techniques. As a result, there currently exists a protection gap in practice, leaving operating systems vulnerable to remote attacks on hardware.

1.1 GOALS AND SCOPE

The main goals of this dissertation are (1) revisiting state-of-the-art defenses with respect to their assumption that mitigating control-flow hijacking solves the problem of memory-corruption-based exploits, and (2) exploring cross-layer attacks in the form of remote hardware exploits, introducing novel design strategies to defend against these profound attacks in practice.

Due to the advances in effectiveness and deployment of defenses such as CFI, an increasing number of exploits aim at attacking the data of applications or the OS rather than their control flows. While control-flow hijacking was shown to give rise to a general and powerful attack framework granting the attacker Turing-complete computational capabilities [10], no analogous construction has been found for data-only attacks, since data accesses heavily depend on the structure of the target program [68]. As we will discuss in the first part of this dissertation the extensive focus of the related work in this area towards the computational capabilities of attack techniques might be to no avail. In practice, adversaries do not require arbitrary computation as part of individual attacks but rather achieve their goals through a multi-step approach, i.e., by initially compromising an application, and then escalating privileges, or leaking sensitive information such as the administrator password the attacker can successfully gain full control over the platform. None of these steps require Turing-complete computation.

Further, we can already see that adversaries are moving from attacking software vulnerabilities towards exploiting hardware vulnerabilities remotely from software—marking a shift in the traditional attack paradigms and threat models that raises a number of significant challenges for existing defenses. Thus, in the second part of this dissertation we explore emerging attacks and introduce novel design strategies to defend the OS in light of these evolving threats.

1.2 SUMMARY OF CONTRIBUTIONS

The main contributions of this dissertation are as follows:

Bypassing CFI in Microsoft's Edge Browser. We demonstrate the power of data-only attacks, by constructing a Data-Only JIT Attack (DOJITA) [1] for Microsoft's Edge browser despite its deployed code-reuse defenses. In particular, we show that just-in-time (JIT) compilers can be tricked into generating arbitrary malicious payloads dynamically in a pure data-only attack by corrupting the intermediate representation in memory. Our attack works despite existing code-injection and code-reuse defenses being deployed and active in the JIT engine and browser.

Breaking and Fixing CFI in the Kernel. By exploiting a vulnerability in kernel code, the attacker can escalate privileges in a second step, again only resorting to data-only attacks. In our work PT-Rand [2], we craft a page-table based attack that completely bypasses all existing defenses in the kernel. It exploits the fact that the kernel manages memory permissions dynamically at run time using data objects, which are called page tables. The attacker leverages the memory-corruption vulnerability to walk the page tables and maliciously modify memory access permissions, e.g., to make code pages writable in the kernel. Our exploit works despite code-injection and code-reuse defenses being deployed and active.

Automatically Uncovering Memory Corruption in Linux. In our work K-Miner [3], we conclude that software-based memory corruption still poses a significant threat and propose the first data-flow analysis framework for the kernel to identify any causes of memory corruption statically in the code and enable fixing vulnerabilities before they are deployed. K-Miner is able to analyze recent Linux kernels completely automatically and was used to uncover real-world vulnerabilities. Our prototype framework implementation builds on top of the popular open-source LLVM compiler and is available online [69]. K-Miner already sparked significant interest from the community and is actively used in follow-up publications [70].
Attacking DRAM hardware from Software. Today, Dynamic Random Access Memory (DRAM) represents the standard choice for practically all computer systems due to cheap prices and widespread availability. However, several independent studies found that frequently accessing physically co-located memory cells in DRAM chips leads to bit flips in adjacent cells [63, 71], an effect called *Rowhammer*. In our work Hammer Time [4] we demonstrate that these software-based fault-injection attacks pose significant security challenges for security-sensitive applications, and we present an initial blacklisting-based defense [5], which leverages the fact that reproducible bit flips tend to cluster around characteristic areas for a given memory module. We demonstrate that this simple strategy can be implemented in a straightforward and OS-independent way in the boot loader.
Practical Software-only Defenses against Rowhammer. Since DRAM chips represent the basic hardware building blocks for main memory on most platforms, Rowhammer represents a general threat to many computer systems. This was also demonstrated through a number of attacks [72–74] even in the remote setting [75–77] to completely break the security guarantees provided by existing software defenses. In particular, these attacks demonstrate that an adversary can allocate memory locations that are physically adjacent to kernel memory on the DRAM chips and subsequently influence the contents of privileged memory cells by exploiting the Rowhammer effect. Since the security of our initial bootloader-based defense approach relies on the quality of the blacklist, we designed a generic software-only defense, called CATT [5], which does not require any information about the distribution of bit flips and can be implemented through a patch to the physical memory allocator in the OS kernel. The basic idea underlying our defense is to partition physical memory into a privileged and an unprivileged zone. In this way we prevent allocation of adjacent rows that belong to different security domains such as the kernel and userland. We evaluated our prototype on a number of test systems and were able to show that it successfully prevents real-world Rowhammer attacks efficiently.
Sidechannel-resilient Kernel-Space Randomization. Even if software-based memory corruption is excluded from the threat model (e.g., because the OS is formally verified [78, 79]), software-exploitable hardware bugs are becoming an increasingly relevant problem. One example are vulnerabilities in the hardware design, for instance, shared caches in the processor can leak information between privileged and unprivileged software and were previously demonstrated to be able to break randomization-based defenses such as KASLR [80–83]. In our work LAZARUS [6] we design and implement a patch for the low-level address space switching mechanism in the OS to protect randomization-based defenses in the kernel against such

micro-architectural sidechannels. Interestingly, a processor vulnerability dubbed Meltdown [66] was discovered concurrently to our work, which exploits speculative execution to load arbitrary memory from an unprivileged process. Since the responsible disclosure process was ongoing at the time, we did not have prior knowledge of this vulnerability, yet, our design of LAZARUS mitigates the exploit by eliminating the side channel created by the shared address space between userland and OS. A similar strategy has since been adopted by all major operating systems to protect against this widespread hardware vulnerability [84].
Looking ahead on Software-exploitable Hardware Vulnerabilities. Since software-exploitable hardware vulnerabilities represent an emerging threat, we take a deep dive into micro-architectural security from a software perspective [7]. In joint work with experts from the semiconductor industry and academia we systematically review existing security-auditing tools for hardware and found fundamental limitations. In particular, existing methodologies fail to adequately model specific classes of vulnerabilities in hardware-description code and suffer from scalability problems in practice. Since all major processor designs are also highly proprietary this can result in bug classes slipping through even combinations of existing techniques. We demonstrate the feasibility of such vulnerabilities being exploitable from software using the popular open-source RISC-V architecture.

1.3 ORGANIZATION

This dissertation is structured as follows. In Chapter 2 we provide comprehensive background on run-time attacks and defenses with a focus towards memory-corruption vulnerabilities and software-exploitable hardware vulnerabilities, which are an upcoming attack vector. Next, in Part II we present two data-only attacks on CFI. First, we demonstrate a novel attack that completely bypasses CFI in Microsoft's Edge browser in Chapter 3. Second, we show that CFI for the Linux kernel can be broken without modifying code pointers in an attack against the page tables. We also introduce our design and implementation of an efficient mitigation in Chapter 4. Third, we present a novel framework to statically analyze real-world kernel code at scale to automatically uncover memory-corruption vulnerabilities in Chapter 5. In Part III we then turn towards software-exploitable hardware vulnerabilities, which represent an emerging attack paradigm. First, we introduce a novel attack based on the infamous Rowhammer vulnerability in DRAM chips in Chapter 7. Second, we present the design and implementation of CATT, the first software-only defense that successfully stops Rowhammer-based attacks against kernel memory in Chapter 8. Next, we look at side-channel attacks on the kernel and present LAZARUS, our practical side-channel defense for the OS in Chapter 9. Lastly, we show that software-exploitable hardware vulnerabilities represent a growing problem by systematically reviewing state-of-the-art auditing approaches used by semiconductor companies to verify real-world SoCs in Chapter 10. We conclude this dissertation in Part IV.

1.4 PUBLICATIONS

This dissertation is based on the following previous publications:

PART II: MEMORY CORRUPTION AND THE THREAT OF DATA-ONLY ATTACKS

[1] **JITGuard: Hardening Just-in-time Compilers with SGX.**
Tommaso Frassetto, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. In 24th ACM Conference on Computer and Communications Security (CCS), November 2017 [Inproceedings].

[2] **PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables.**
David Gens, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi. In 24th Annual Network and Distributed System Security Symposium (NDSS), February 2017 [Inproceedings].

[3] **K-Miner: Uncovering Memory Corruption in Linux.**
David Gens, Simon Schmitt, Lucas Davi, Ahmad-Reza Sadeghi. In 25th Annual Network and Distributed System Security Symposium (NDSS), February 2018 [Inproceedings].

PART III: REMOTE HARDWARE EXPLOITS AS AN EMERGING ATTACK PARADIGM

[4] **It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs**
Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. In 55th Design Automation Conference (DAC'18), June 2018 [Inproceedings].

[5] **CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory.**
Ferdinand Brasser, David Gens, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi. In 26th USENIX Security Symposium, August 2017 [Inproceedings].

[6] **LAZARUS: Practical Side-channel Resilient Kernel-Space Randomization.**
David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, Ahmad-Reza Sadeghi. In 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), September 2017 [Inproceedings].

[7] **When a Patch is Not Enough — HardFails: Software-Exploitable Hardware Bugs.**
Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Karthik Kanuparthi, Hareesh Khattri, Jason M. Fung, Jeyavijayan Rajendran, Ahmad-Reza Sadeghi. [Technical Report].

2 BACKGROUND

In this chapter we first give a short overview of the traditionally deployed defense mechanisms and operating system security concepts vital to the understanding of this dissertation. We then introduce the notion of memory-corruption vulnerabilities, which underlie most software-based attacks against those traditional defenses and discuss advanced attacks and mitigations. Finally, we cover the high-level architecture of modern hardware platforms, including the most relevant components in the context of upcoming and emerging attack scenarios that exploit hardware vulnerabilities remotely from software.

2.1 BASIC OS-LEVEL DEFENSES

As mentioned in Chapter 1 operating systems support a large number of applications running concurrently or time-shared on a single hardware platform. Since applications may be vulnerable to attacks, or even malicious in some cases, modern operating systems are designed as reference monitors and strictly separate access control and resource management from user processes [8, 20, 21]. Implementing this separation requires hardware support and modern processors provide two different isolation mechanisms in the form of *privilege separation* and *virtual memory*. We give a brief introduction into both mechanisms and explain how the OS leverages them to achieve strict isolation at run time.

2.1.1 *Privilege-Level Separation*

OS kernel software normally runs with higher privileges than application software. Many processors support different *privilege levels* and allow software to switch between them at run time, e.g., via dedicated control-transfer instructions. Typically, there is an unprivileged level for executing applications and a privileged level that is reserved for the operating system. When the processor executes with the highest privilege level, the running software has complete access to any instructions, can directly access platform hardware, interact with debug and control registers of the platform, and consequently has full control over the system. Executing code within the unprivileged level only leaves a limited number of benign instructions available to software, for instance, it is usually only possible to load and store unprivileged memory or registers, calculate arithmetic operations, evaluate conditionals, and branch execution. At run time, the hardware maintains separate execution contexts for the different privilege levels and switching between privilege levels transparently invokes the respective context. For instance, when a user program crashes due to a benign operation, such as when dividing by zero, the processor automatically invokes the code in the highest privilege level (i.e., the operating system). The OS can then handle the error condition caused by the user program and terminate the application. User-level software can also purposefully

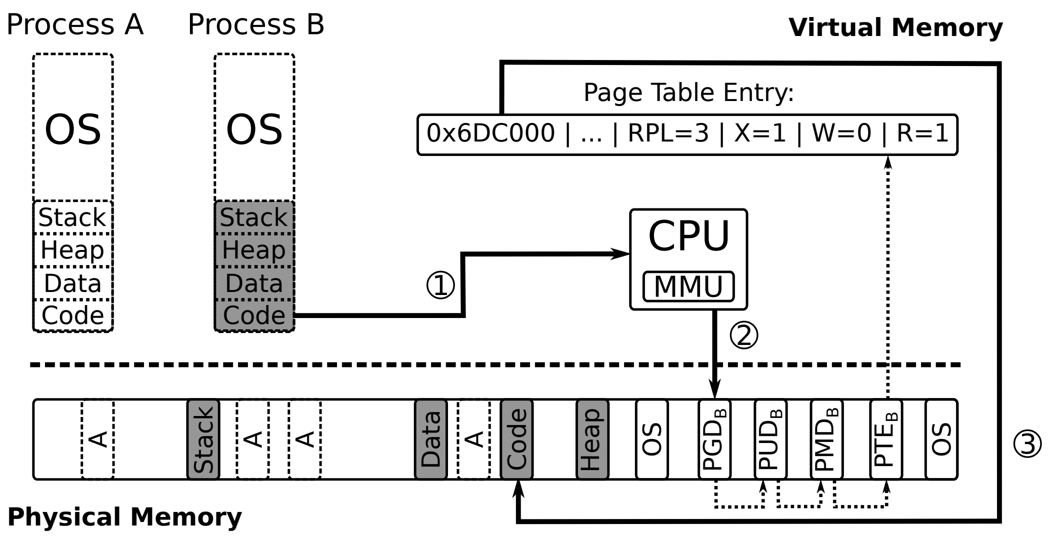


Figure 2.1: Virtual memory represents a page-based memory-protection scheme that can be enforced in hardware on many platforms.

invoke the OS. For example, if an application requires privileged access, such as opening a file stored on the platform's hard disk, it has to issue a request to the OS for accessing the disk and retrieving the file on its behalf. In this way the OS can mediate any accesses and enforce security policies. OS services are standardized in the form of *system calls* [85] and this interface represents one major part of the kernel's attack surface towards user-space software [86].[1]

2.1.2 *Virtual-Memory Protection*

In addition to privilege separation, the OS usually also maintains a strict isolation of memory for application software. On many Central-Processing Units (CPUs), memory isolation is enforced by a dedicated piece of hardware called the *memory-management unit* (MMU), which is part of the processor (as depicted in Figure 2.1). If the MMU is disabled, all memory accesses from software operate directly on physical memory, which means that no memory isolation is enforced.[2] If the MMU is enabled by the OS, all memory accesses from software are interpreted as *virtual addresses* instead and every process is associated with its own *virtual-address space*. This indirection forces all accesses from software to be checked against a set of controls before any actual access to physical memory is allowed. These access control flags are stored in a hierarchical data structure, called the *page table*. In the example from Figure 2.1 Process B accesses a virtual address in Step ① which is translated and checked by the MMU. Obtaining the access controls for the given virtual address requires a look-up in the page table in Step ②, which might involve several layers of indirection in physical memory. Therefore, this *page walk* is considered a costly operation and often optimized by the OS and the hardware in various

1 Alongside the file system, drivers, and the network stack in monolithic kernels.
2 This used to be a common mode of operation on old operating systems, such as Microsoft's Disk Operating System (MS-DOS).

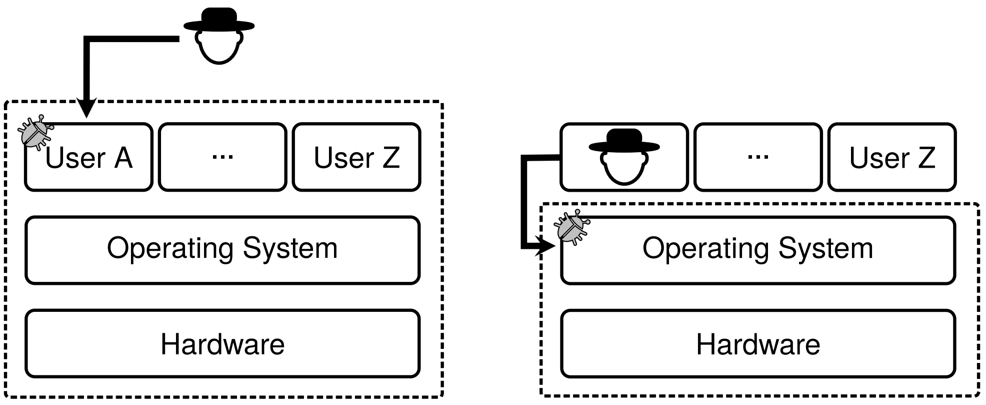


Figure 2.2: Today, many attacks involve multi-step exploits, in which the attacker successively migrates from a *remote* to a *local* adversary model to gain full control over the platform. Since the OS implements most of the security-relevant hardening techniques, attacks against kernel software in this second step are increasingly common in practice.

ways.[3] Once the page table entry for the given address is retrieved, the virtual address can be translated in Step ③ to the respective physical address, which can be interpreted directly by the memory bus of the platform. Alongside the physical address, the page-table entry stores access-control flags associated with the entry, such as Requested-Privilege Level (RPL), readable, writable, or executable. Depending on the nature of the access and the Current-Privilege Level (CPL) of the code that issued the access, the MMU will grant or deny that access. In case of a denied access the OS will be notified. When execution switches to Process A, another set of page tables will be loaded in the MMU. In general, this can result in entirely different access control policies as page tables of different processes can be kept completely independent. In this way, different parts of the main memory can be assigned to different processes exclusively. However, it is also possible to share memory between applications by mapping the same physical page within different address spaces or map it multiple times under different aliases within the same address space.

2.2 THE TRADITIONAL THREAT MODELS

In combination, both mechanisms enable the OS to enforce strict isolation for application software at run time. The adversary models commonly considered in this setting can loosely be categorized into two variants, which are depicted in Figure 2.2.[4] On the left-hand side we can see the *remote adversary* attacking the platform from the outside, e.g., through a network connection. In this case, the Trusted-Computing Base (TCB) of the system includes all user-space programs for which the attacker is able to provide some form of input. Many real-world scenarios can

3 Since this aspect has important security implications we will in fact discuss it in greater detail in Section 2.5.2 and Chapter 9.

4 Both cases usually exclude physical attacks (i.e., any techniques requiring physical presence), since this imposes additional, strong requirements on an adversary.

be modeled in this way. For instance, the attacker could have access to a shared file storage on the local network, or there might be a webserver running on the target system that the attacker can connect to via the internet. If the respective server program contains a vulnerability, an adversary can leverage this to exploit the server and achieve some initial control over the victim machine in the remote setting. In some cases, the attacker might want to extract information, or the goal could be modifying the webserver's configuration. In other cases, an adversary might utilize the compromised process to attack other parts of the system or even compromise other machines. The right-hand side shows the model for a *local adversary*, which assumes that the attacker already controls a user process on the victim machine. This means that the attacker can in principle execute arbitrary unprivileged code. However, most computer systems nowadays run hundreds of processes in parallel, and hence, operating systems are designed to consider user processes as untrusted and potentially malicious a priori. To this end, most major operating systems employ various hardening techniques, such as mandatory-access control [87], sandboxing [88, 89], and address-space layout randomization [90] by default. In practice, we can see that attacks against the kernel as a subsequent step are gaining more and more relevance for several reasons: first, the kernel executes with higher privileges, often allowing the attacker to gain full control over the machine based on a single kernel exploit. Second, the OS implements a major part of the security hardening techniques mentioned above, and hence, obtaining a root process or escaping from sandboxes often requires a kernel compromise. Consequently, many real-world attacks involve a multi-step approach in which the attacker starts out in the remote setting and successively migrates to the local setting by first acquiring code execution in user space and subsequently attacking the OS. Indeed, the OS has become a high-value target in the recent past as kernel exploits are leveraged in all of the latest Android root exploits [37–39, 91] and iOS jailbreaks [92], application sandbox escapes [23], and even large-scale attacks against industrial control systems [24].

2.3 MEMORY-CORRUPTION ATTACKS

In this section, we explain why run-time attacks based on *memory corruption* pose a number of difficult challenges for these basic defenses under the traditional threat models. First, we give a quick introduction into the underlying root causes of software-based memory corruption. Second, we provide an abstract model and definition of what constitutes a memory-corruption exploit. Third, we give a simple real-world example of a typical exploitation workflow.

2.3.1 *Causes of Memory Corruption*

The problem of memory corruption has proven uniquely challenging from a security perspective. In part, this is due to the fact that memory corruption is closely tied to the way in which we specify program behavior using low-level languages like C. Normally, the developer specifies the intended behavior of the program in the code and a compiler then translates this high-level human-readable description into machine code which can be executed directly by the processor. However, these languages explicitly allow for *undefined behavior*, i.e., programs which do not adhere to a strict set of technical specifications defined in the language standard are allowed to exhibit *any* behavior at run time. A popularly exercised example states that a program exhibiting undefined behavior could format the user's hard drive in full accordance with the standard. While in reality, accidentally formatted hard drives do not rank among the most frequently reported programming bugs[5], undefined behavior can have severe consequences for the system that executes the program. In theory, any program that is written in an unsafe language could contain a memory-corruption vulnerability. Unfortunately, empirical evidence from almost 40 years shows that a large part of real-world software is prone to such bugs over time: memory-corruption vulnerabilities are a wide-spread problem and occur frequently in deployed code [13–19]. Even worse, patches that fix a number of older bugs may also introduce new bugs [11, 12]. Indeed, memory-corruption vulnerabilities represent a vast number of security-relevant bugs for modern software written in low-level languages [98, 99], which can be exploited by adversaries, e.g., to obtain remote code execution or elevated privileges at run time. Memory-corruption vulnerabilities are often classified according to their root defect and integer overflows, use-after-free, dangling pointers, double free, buffer overflows, missing pointer checks, uninitialized data usage, type errors, or synchronization errors are commonly listed classes of memory corruption [11–13].

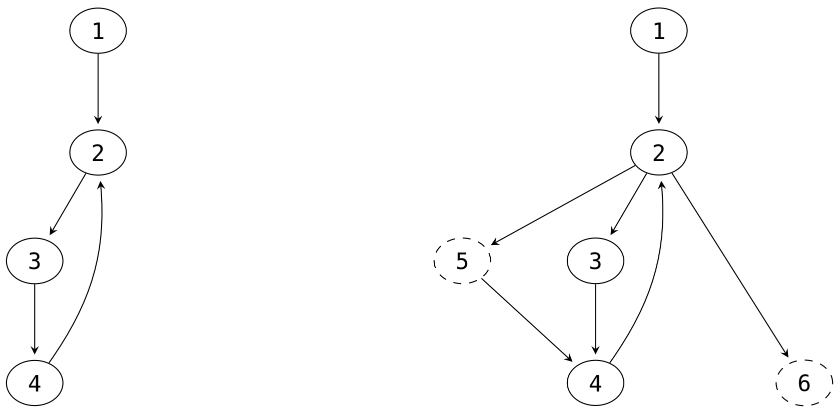
While untrained or inexperienced software developers certainly account for a number of these vulnerabilities, there are also more fundamental, systemic root causes for memory corruption. The ANSI C standard [100] explicitly lists 96 instances which will cause undefined behavior[6] and further remarks that a language implementation "*may generate warnings in many situations, none of which is specified as part of the Standard*" [100]. This means that in practice software developers can easily miss one of the listed conditions in their code, accidentally writing a program that will exhibit undefined behavior at run time. One of the consequences can be *memory corruption*, i.e., a situation in which part of program memory, and hence, the overall program state is no longer well-defined. A simple real-world example of such a case is the buffer overflow. In this situation, the size of some user-controlled input should be written to a memory location but exceeds the capacity of the buffer reserved at that location. This can be exploited by adversaries to maliciously modify the data or even hijack the control flow of the program at run time.

2.3.2 *A Model for Exploitation*

Abstractly, program execution can be viewed as a sequence of transitions within a particular finite-state machine (FSMs) as depicted in Figure 2.3. The left-hand side shows the benign states of the intended state machine as originally specified by the programmer. In this case, there is a dedicated initial state ① of the program from which execution begins. State transitions are specified by the software developer

5 Although, it does happen occasionally [93–97].

6 By comparison, the modernized standard ISO/IEC 9899:1999 [101] (commonly called C99) lists 191 cases and the most recent standard ISO/IEC 9899:2011 [102] (commonly called C11) lists 203 instances of undefined behavior.



(a) Intended states. **(b) Hidden states.**

Figure 2.3: Programs can be represented as finite-state machines **(a)**, with vulnerable programs containing hidden states **(b)**, which only become reachable through undefined behavior to form *weird machines* [103, 104].

in the code, e.g., in this case the developer intended the program to cycle endlessly between the three states ②, ③, and ④ after leaving its initial state. The right-hand side shows that in the case of undefined behavior the same program might in fact exhibit unintended, hidden states or *weird states* [103], which are not reachable during benign execution. If it is possible to trigger the vulnerability explicitly, an adversary might be able to force the program into hidden states that are beneficial from an attacking perspective. Strictly speaking, the code that the developer specified for such a program becomes completely meaningless and the intended finite state machine is replaced by a *weird machine* [103]. This is also evident from the program on the right-hand side, which now contains a hidden terminal state ⑥, contrary to the originally specified intended state machine, which cycles endlessly between the three benign states. In this abstract view of exploitation, the actual exploit represents an input or trigger which programs the weird machine to exhibit unspecified or unintended behavior at run time. It is important to note, that this abstract model is completely supported by the language specification. It was recently formalized and also shown to allow reasoning about non-exploitability of a program [104].

2.3.3 *Typical Attack Workflow*

A classic example for the workflow of such a memory-corruption exploit is given in Figure 2.4: here, the program in **(a)** utilizes the legacy function `gets` to read user input and store it in a pre-defined buffer storage without checking the length of the input. So, while the programmer only reserved memory for 17 characters in the program code a user can provide inputs of arbitrary size at run time. In most benign cases, this will simply cause the vulnerable program to crash. However, an adversary can provide a specifically crafted input similar to **(b)** which will force the program into a weird state **(c)** that spawns a terminal program and provides the attacker with an interactive programming environment. This is achieved by

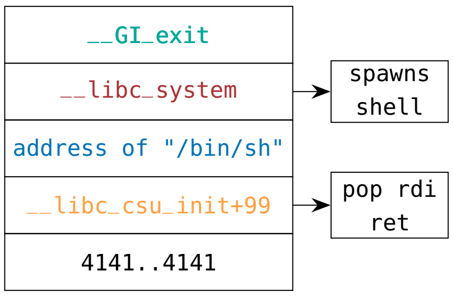
```c
#include <stdio.h>
#include <strings.h>

int main(void) {
  char buffer[17];
  gets(buffer);
  printf(buffer);
  return 0;
}
```

(a) Program.

```
4141 4141 4141 4141
4141 4141 4141 4141
4141 4141 4141 4141
4141 4141 4141 4141
4141 4141 4141 4141
0306 4000 0000 0000
579d b9f7 ff7f 0000
9023 a5f7 ff7f 0000
3070 a4f7 ff7f 0000
```

(b) Exploit.



(c) Corrupted stack.

Figure 2.4: The vulnerable program **(a)** exhibits undefined behavior for inputs exceeding the length of the buffer. An adversary can exploit this by providing malicious input **(b)** to bring the program into a weird state **(c)** that will spawn a terminal—a behavior that was never specified anywhere in the program code.

overwriting the return address of the main function which is stored on the stack. Any instance of memory corruption leaves the program vulnerable to run-time attacks, and typical exploitation techniques range from injecting malicious code, to reusing existing code with a malicious input, to corrupting integral data structures of the running program without hijacking its control flow. An adversary with knowledge of any such vulnerability can potentially exploit this at run time by deliberately triggering the error to achieve unintended, malicious behavior and take control over the system.

2.4 STATE-OF-THE-ART ATTACKS AND DEFENSES

Here, we give a quick overview of the relevant literature and state-of-the-art research in run-time attacks and defenses based on memory corruption, such as Return-Oriented Programming (ROP) [10] and Control-Flow Integrity (CFI) [40]. First, we introduce code-reuse attacks and initial, randomization-based defenses. Second, we explain the basic concept behind enforcement-based code-reuse defenses like CFI and variants thereof. Since a rich body of literature exists on the topic of code-reuse attacks and defenses already, we refer the interested reader to the related work for an in-depth and fully comprehensive discussion [105–111]. Third, we briefly introduce the notion of data-only attacks in that context.

2.4.1 *Return-to-libc and Return-Oriented Programming*

Run-time attacks based on memory-corruption vulnerabilities have been a persistent threat to application and kernel-level software for more than three decades [13–19, 22–26]. Initially, *code-injection attacks* received much attention, since a simple buffer overflow vulnerability enabled adversaries to inject arbitrary code into a running program, e.g., by providing malicious inputs that would spawn a shell (hence the term *shellcode*). Nonetheless, these attacks were quickly prevented by deploying hardware-based defenses such as Data Execution Prevention (DEP) [112, 113], which disable execution of data memory on the platform as a general policy. Hence, attackers are no longer able to execute code that they injected into the memory of the program. However, the underlying attack paradigm was quickly adapted to bypass DEP as a defense by generalizing code-injection attacks to *code-reuse attacks* [10, 114]. In a code-reuse attack an adversary exploits a memory-corruption vulnerability to hijack the control flow of a running program by maliciously modifying a code pointer instead of injecting any code, e.g., the return address which is usually stored on the stack of the program can be modified to point to some arbitrary code location that is already present in the memory. In the simplest case, this enables the attacker to redirect execution to another function, such as a library function that forks another program, and supply a malicious input (e.g., "/bin/sh" to launch a shell). However, the most prominent example is return-oriented programming (ROP) [10, 115], in which execution is redirected to small snippets of individual instructions (called *gadgets*) that end in a return instruction. Since the attacker is able to chain arbitrary sequences of gadgets together, ROP attacks have been shown to be Turing-complete. This means, that the attacker can achieve arbitrary computation in theory, and hence, ROP attacks represent a powerful exploitation tool in practice. It is important to note that code-reuse attacks are possible on a wide range of system, some of which do not even offer a dedicated return instruction. In particular, the possibility of dynamic (or indirect) branching suffices to construct code-reuse attacks [116–118].

2.4.2 *Code-Reuse Defenses and Advanced Attacks*

Consequently, a number of advanced defenses have been proposed in the related work to protect existing systems against such attacks. Most approaches fall into either of two categories: (1) randomization-based, or (2) policy-based defenses.

Randomization-based approaches aim to lower the portability of a vulnerability among identical systems by randomizing parts of the virtual-address space between applications. Thus, these approaches are sometimes referred to as *software diversification* as well [110]. The idea of address space layout randomization (ASLR) dates back to 1997 [119] and was first implemented in 2000 [90] to randomize the virtual memory layout for user processes. In particular, ASLR randomizes the stack, heap and shared libraries prior to program loading. Randomization-based defenses usually come with a security-performance trade-off, where increased randomization granularity amounts to higher entropy while typically also introducing more overhead [50]. Besides brute-force guessing attacks, which are rendered infeasible in practice by requiring more than 20 to 30 bits of entropy, randomization-based defenses are subject to information-disclosure attacks. An information leak discloses information about the memory layout or individual addresses of a process giving an adversary the opportunity to adjust the exploit. There are multiple examples of memory-disclosure vulnerabilities exploited in the real world, such as in Adobe's Flash [120], Adobe's PDF Reader [121], and Microsoft's Internet Explorer [122]. For this reason, more fine-grained randomization techniques were proposed, e.g., by randomizing at the function-level [50], or even at the instruction-level [47, 123]. However, the run-time penalties associated with those schemes is generally considered too high in practice. Moreover, even a single leaked code pointer enables an adversary to completely de-randomize an application at run time by disassembling randomized code pages while following newly discovered code pointers. In this way, the attacker can dynamically assemble a gadget payload via indirect disclosure of code pages, a technique dubbed just-in-time code reuse, or JIT-ROP [17]. While information leaks pose significant challenges to fine-grained randomization schemes, a proposed defense technique is to make code pages non-readable [54, 55]. This was demonstrated to defeat indirect information-disclosure attacks. However, standard hardware does not support non-readable code pages, and hence, implementing this defense requires a hypervisor and extended page table support. Code-Pointer Integrity [46] proposes to partition a program's data memory into a region for code pointers and a region for the remaining data. By randomizing the location of the code-pointer region, all accesses to code pointers can then be assumed to be safe. While this compiler-based approach was shown to introduce a lower overhead than fine-grained randomization defenses against code-reuse attacks, the location of the safe region was subsequently shown to be prone to timing sidechannels [124].

In contrast to these probabilistic randomization-based defenses, Control-Flow Integrity (CFI) [40] emerged as a promising defense mechanism that offers formal security guarantees, while incurring modest overheads. The high-level idea behind CFI is to analyze the program code at compile time to compute *labels* at all possible branch locations and insert checks that restrict the control flow to the correct label set. Since this approach requires additional checks to be executed at run time for indirect jumps, initial implementations induced overheads from 5 to 20 percent, which is still considered impractical for many applications [41]. Moreover, the accuracy of the pre-computed and enforced label sets is essential to achieve a high level of security. Deploying CFI in the context of object-oriented code has proven challenging, as implementations that neglect the detailed semantics of languages like C++ were shown to be subject to attacks [18]. For binary code, imprecise label sets can lead to attacks despite coarse-grained CFI policies being enforced [125]. Hence, secure CFI implementation typically require an in-depth static analysis and instrumentation phase at compile time, costly run-time checks, and a shadow stack implementation [126]. To reduce the performance hit, hardware-assisted CFI implementation have been investigated [42, 60] and hardware extensions have been announced to offer dedicated CFI support for upcoming CPU architectures [61].

2.4.3 *Data-only attacks*

In contrast to code-injection and code-reuse attacks, *data-only attacks* [13] do not hijack the control flow of an application. Instead, the attacker tampers with the input data of functions or directly corrupts data structures in memory. Although data flows are in general heavily application-dependent, data-only attacks were previously shown to severely affect the security of vulnerable software [68]. Additionally, while data-only attacks may not always offer the same flexibility and expressiveness as code-reuse attacks, they can pose a significant threat in practice. For example, the attacker can change the current user id to elevate privileges, or change file names to disclose secrets such as configuration files or cryptographic keys. So far, no efficient and general data-only defense exists as all intended data flows

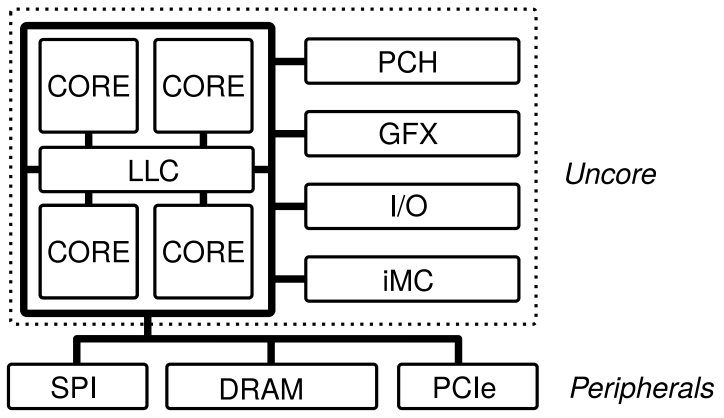


Figure 2.5: Modern computer platforms feature complex SoC architectures that combine multi-core designs with performance-critical elements *uncore* inside the CPU package. Components inside the CPU package share a large last-level cache and are tightly coupled through high-speed interconnect fabric on the die. Peripherals are connected to the processor package via the system bus.

must be protected to construct an effective mitigation, resulting in prohibitively expensive overheads for existing schemes [49, 127, 128]. We demonstrate the real-world threat of data-only attacks in the context of multi-step attack approaches against OS kernels and also present possible mitigations in depth in Part II.

2.5 MODERN HARDWARE PLATFORMS

Since many defenses against memory-corruption-based attacks have been presented and also deployed over the years, adversaries are looking for new ways to attack systems at run time. Naturally, software runs on physical hardware, which has grown quite complex over time. As it turns out, this also opens up novel opportunities for attacks. Computer platforms have evolved significantly during the last 30 years, trading their relatively simple, discrete layouts in the beginning for more complex, but faster system-on-chip (SoC) designs. In this section, we first look into the overall system architecture of modern platforms. We then discuss the processor design principles underlying recent computing platforms in more detail and further outline in which ways this can prove problematic from a security perspective.

2.5.1 *Overall System Architecture*

In particular, earlier architectures connected a large number of these discrete and specialized components through hard-wired or printed circuit lanes on the mainboard. In contrast to that, modern processors integrate many of these traditionally separate chip elements on a single die, minimizing the number of required communication pathways off-chip to enable faster and wider data connections. The basic platform architecture of such an integrated, SoC-based design is depicted in Figure 2.5. At the highest level, multi-core architectures typically have an intricate interconnect fabric between individual cores, the last-level cache (LLC), the uncore elements, and the system bus which connects to the off-chip peripher-

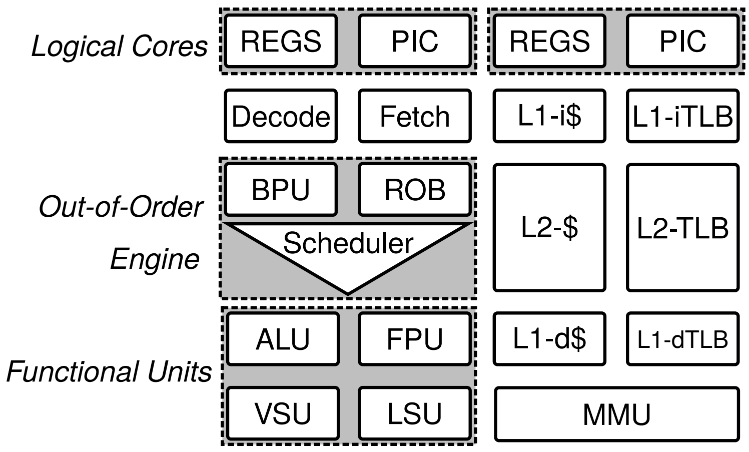


Figure 2.6: Semiconductor manufacturers aim to enhance single-core performance by maximizing *instruction-level parallelism*. For this, chip designers leverage techniques such as super-scalar processing, simultaneous multi-threading, and out-of-order execution, which heavily increase implementation complexity of modern processors.

als. While the main components have remained largely the same, many elements were moved from off-chip to uncore in recent years. Traditionally, the mainboard chipset employed a north- and southbridge architecture, which used to connect peripherals with faster and slower data connections respectively. However, since many peripherals rapidly increased their communication speed and maximal capacity, the Platform Controller Hub (PCH) has replaced this older layout. On recent CPUs, it was moved into the processor package to allow for more direct links, enabling faster communication and more fine-granular control. It now also combines power management and security services, e.g., in the form of the Converged Security and Management Engine (CSME) [129]. Moreover, modern CPUs typically also directly contain dedicated graphics processing capabilities (GFX), as well as complex configurable hardware controllers (I/O). Lastly, they incorporate an integrated memory controller (iMC), which usually resided on the mainboard as well on older platforms. The main peripheral connections that remain on the mainboard on modern platforms represent the flash ROM chip[7] attached via Serial-Peripheral Interface (SPI), main memory in the form of Dynamic Random-Access Memory (DRAM), and high-speed peripherals such as SSDs and GPUs which are connected over Peripheral Component Interconnect Express (PCIe).

2.5.2 *Processor Design Principles*

The bi-annual doubling of transistor density no longer drives performance improvements in integrated circuit designs, as Moore's Law is hitting the physical limits of silicon-based semiconductor technology [130]. Thus, processors are increasingly designed to optimize execution, e.g., by heavily improving the number of instructions executed per unit time (dubbed *instruction-level parallelism*). For instance, practically all modern processors are super-scalar architectures, which

7 Which contains the firmware (i.e., boot ROM, CSME code and data partitions, BIOS, and μ-code).

means they are able to utilize functional units in parallel to execute more than one instruction per cycle. This requires careful management of the instruction pipeline to identify data dependencies between consecutive instructions and puts additional constraints on the design of the decode and fetch stages. Further, Simultaneous Multithreading (SMT) [131] subdivides a single physical core into a number of *logical cores*, introducing the ability to simultaneously process different instruction streams within a single core.

The high-level design of such a processor is depicted in Figure 2.6 and usually simply replicated for each physical core on a multi-core chip. Individual instructions are loaded into the pipeline by the fetch stage of the processor, and decoded into a number of microcode instructions (μ-ops), which are executed by the functional units. High-end processors often feature complex *Out-of-Order Engines* that may schedule and process μ-ops of the same instruction stream ahead of time, leveraging Re-Order Buffers (ROB) to emit the results of re-ordered instructions in the correct sequence before finalizing execution. Some processors additionally employ Branch-Prediction Units (BPUs) to speculate on the target address of branch instructions, e.g., to make loop iterations cheaper. The *functional units*, such as Arithmetic Logic Units (ALU), Floating-Point Units (FPU), Vector-Scalar Units (VSU), and the Load-Store Units (LSU) represent the fundamental building blocks that are contained in some form on every processor. Usually, the LSUs are the only functional units that operate on external memory. DRAM access latencies are many orders of magnitude slower than processor cycles, and hence, CPUs leverage complex, multi-level caching hierarchies to close this gap and enhance performance in practice [132]. The fastest level, L1, is divided into an instruction cache (L1-i$), which contains the machine code of the currently executed thread, and a data cache (L1-d$). Typically, its size is in the order of a few memory pages and access latencies are in the range of a single CPU cycle. The second level, L2, usually unifies code and data, and while its access latency is about an order of magnitude slower than the L1, its size can be a magnitude larger. SMT processors contain multiple Register Sets (REGS) and Programmable-Interrupt Controllers (PICs) to maintain separate threads independently at run time, and hence, threads running on two logical cores share all of the core's physical resources. Since individual instructions can exhibit large differences in execution time, one thread may often stall the core while waiting for a scheduled instruction to finalize operation. The instruction stream from other threads running on the same physical core can then be used to maximize resource utilization by executing its instructions in parallel to the instruction stream from the stalled thread. As mentioned in Section 2.1.2, many processors support virtual-memory protection through a Memory-Management Unit (MMU). While MMUs allow for a fine-granular, page-based policy management and enforcement, this also requires an indirection for every memory access, which strongly affects run-time performance. For this reason, processors which include an MMU typically also feature a series of faster caches for these policy look-ups, called the Translation-Lookaside Buffers (TLBs), which usually mirror the cache hierarchy for code and data memory. Again, these hardware resources are shared among all logical cores of a physical package.

2.5.3 *Remote Hardware Exploits*

Resource sharing maximizes utilization, and hence, optimizes run-time performance. However, the added complexity and extreme transistor densities of today's chips also lead to side effects that can be leveraged by malicious software.

One example of this are *side-channel attacks*. The goal behind a side-channel attack is usually to leak information, such as passwords or cryptographic keys and many practical attacks have been presented in the past. Side-channel attacks are especially problematic when processor resources are shared across different privilege levels, as this may allow an unprivileged adversary to disclose privileged information. For instance, by leaking internal data from the OS through a side channel an adversary can break fine-grained randomization and subsequently launch a code-reuse attack [22, 80]. Additionally, many architectures offer dedicated prefetching and flushing instructions to load or empty out caches to allow manual optimization from software. These operations directly influence and disclose information about the state of the caches and allow co-located attackers to probe and measure the effects of accesses by victim software [82, 83, 133].

While side-channel attacks are limited to information disclosure, *remote-fault injection attacks* enable adversaries to corrupt hardware entirely from software, potentially compromising the security of the entire platform. One recent example is Rowhammer [63], which leverages a hardware vulnerability in DRAM chips to generate bit flips in memory from software at run time and many attacks have been presented that completely bypass existing defense mechanisms [64, 74–77, 134, 135]. Another example is CLKScrew [65], which exploits power management functionality to produce processor glitches at run time from software. This enables attackers to compromise even dedicated hardware security extensions, such as ARM TrustZone, which were envisioned as a general defense against software-based attacks [136].

While both attack classes mentioned above exploit specific vulnerabilities or features of bare-metal hardware, it is important to note that *firmware attacks* may also enable remote attacks against hardware. Firmware represents software that is embedded into the chip by the manufacturer and (contrary to the OS) cannot easily be modified by the owner of the platform. To this end, it serves as a bridging technology by managing low-level control tasks, such as platform bring-up, power-saving stages, and chip-level peripheral control. Although these tasks could in principle be implemented through additional hardware, firmware allows for greater flexibility across different product lines and, in contrast to hard-wired circuitry, can also be updated after deployment. Naturally, as firmware grows more complex it is prone to similar attacks as traditional system software: for instance, firmware attacks have been demonstrated against Intel ME [137] and NVIDIA's Tegra Cores [138] recently, affecting millions of platforms such as many recent Intel's x86 processors [139] and Nintendo's Switch gaming consoles [140].

Overall, we can see all three techniques are increasingly gaining attention, as remote hardware exploits represent an upcoming threat. Recently, successful hardware-based exploits have been demonstrated from malicious software against applications [141], the OS [66], and even against advanced hardware-security architectures like Intel SGX [67] and ARM TrustZone [65]. Since firmware could in principle be protected by adapting software-based defenses, we focus on the first two techniques in this dissertation. We discuss remote hardware-based attacks as well as possible mitigations in depth in Part III.

Part II

MEMORY CORRUPTION: THE THREAT OF DATA-ONLY ATTACKS

based on JIT-Guard [1], PT-Rand [2], and K-Miner [3].

3 BYPASSING CFI IN MODERN BROWSERS.

In this chapter, we demonstrate that state-of-the-art code-reuse defenses like CFI are not sufficient to stop the threat of memory-corruption attacks. In particular, we present a data-only attack, termed DOJITA [1], which completely defeats CFI in the browser context.

3.1 DATA-ONLY ATTACKS ON DYNAMIC CODE GENERATION

Big applications, such as web browsers and document editors support a large number of features, for instance, to support a variety of file formats, and increasingly offer highly customizable functionality to end users. To achieve a high level of flexibility, many of these applications leverage *dynamic code*. These scripting languages, like JavaScript, are interpreted at run time and can easily be updated or modified without changing the static code of the surrounding application. In contrast to static programming languages, like C and C++, this requires a dedicated run-time environment, that additionally takes care of memory management and provides an interface to a rich set of functions. For instance, dynamic scripting languages are typically included in most web browsers, as today many websites require a form of dynamic interaction with the user. As such, high-performance execution of dynamically generated code can be essential. To this end, static code that includes dynamic scripting components usually employs just-in-time (JIT) compilers to translate the interpreted, high-level scripting code into machine code at run time, since interpretation is orders of magnitude slower than native execution.

It is noteworthy to mention that dynamic code is not limited to the application level, but represents a common technique to increase flexibility of large software in general: even some operating systems include a dedicated JIT engine in the form of the Berkley Packet Filter (BPF) [142], which was originally designed as a user-programmable network filter in the kernel, however, it has been significantly extended over time to support a variety of use cases (e.g., sandboxing).

However, adversaries can leverage memory-corruption vulnerabilities in the static component of the application or OS to attack dynamically generated code at run time as a means to circumvent hardening techniques deployed for static code. In particular, adversaries with access to a memory-corruption vulnerability can usually achieve arbitrary read-write capabilities and exploit this access to corrupt code pointers or data and take full control at run time. Code-injection attacks [9] normally represent a legacy exploit method on modern system due to the extensive deployment of non-executable memory [112, 113]. Interestingly, this type of attack has seen a revival in the context of dynamically generated code, since JIT-engines have to be able to write code pages at run time [143, 144]. If an application contains a memory region that is writable and executable at the same time the attacker can inject arbitrary instructions into this region and then hijack the control flow to execute them. This is why browsers soon deployed non-executable memory for JIT engines [145].

Nonetheless, even if code generated by the JIT engine can no longer be modified by the attacker, code-reuse attacks are still possible [146, 147]. Hence, state-of-the-art code-reuse defenses have increasingly seen adoption in the JIT context as well [55, 144, 148], by verifying the destination addresses of dynamically generated branches with a pre-defined security policy.

3.2 ASSUMPTIONS AND THREAT MODEL

We aim to showcase the power of data-only attacks in the context of dynamically generated code, despite state-of-the-art defenses being deployed and active. In particular, we consider attacks against the static component of the application to be prevented by these defenses. Below, we list our detailed, standard assumptions that are also consistent with the prior art in this field [55, 144, 147–149].

- **Code-Injection and Code-Reuse Defenses.** Under our model, code-reuse and code-injection defenses are supported and enforced. Hence, injecting malicious instructions is prevented by non-executable data pages [113]. Further, control-flow integrity [40, 148, 150, 151] is assumed to prevent code-reuse attacks against the static or dynamic parts of the application. The application and the operating system code are considered as trusted.

- **Address-space layout randomization.** We additionally assume a form of Address Space Layout Randomization (ASLR) [90], which hides the location of allocated data regions from an adversary.

- **Memory corruption.** Under our threat model, the application contains a memory-corruption vulnerability which is known to the attacker, which represents a realistic scenario [17, 18, 152]. Consequently, the attacker can exploit this to disclose and manipulate data memory.

- **Scripting Environment.** The dynamic component of the application allows an adversary to execute sandboxed, interpreted code.

The goal of the attacker in this setting to execute arbitrary *native* code in the execution context of the static component (e.g., a browser process or the OS kernel). The attacker can then try and further compromise the system, or leak sensitive information from the web page by launching the attack using some malicious input, such as a injected advertisement code in the browser. [1] For applications, the use of additional defense mechanisms like sandboxing [88, 89] can make this second step harder to achieve in practice. However, such defenses are not available for the OS and also do not generally prevent the first step, hence we consider them as orthogonal to our model.

1 For the scenario of attacking an OS-level JIT engine the malicious JavaScript ad would be replaced by a user-provided network filter script.

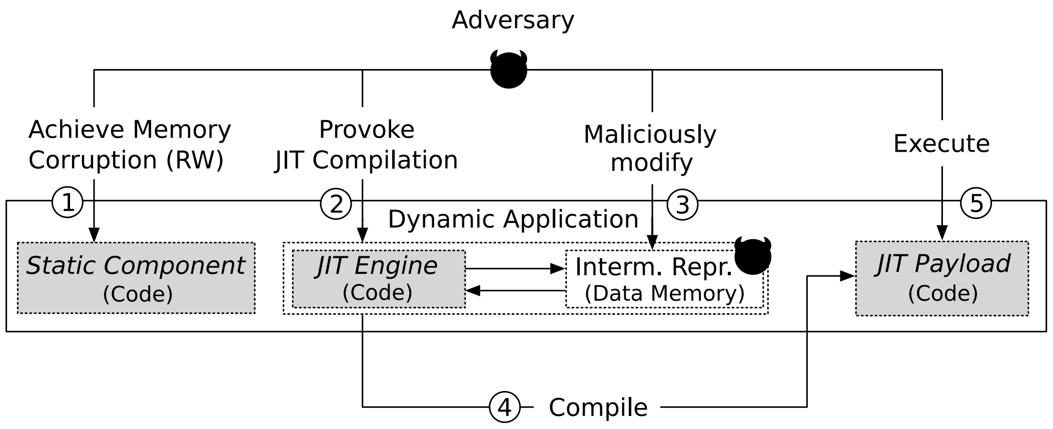


Figure 3.1: Our attack bypasses CFI enforcement by maliciously modifying the intermediate representation (IR), which resides in data memory and is never executed directly. Instead, it is used by the compiler to output machine code for an attack payload which the attacker can then invoke in a subsequent step.

3.3 GENERICALLY BYPASSING CFI BY EXPLOITING JIT-COMPILERS

All previously discussed defenses for the JIT compiler aim to prevent code-injection or code-reuse attacks. To assess those defenses we conducted initial tests that showed that the attacker can use data-only attacks, i.e., memory-corruption attacks that do not corrupt any code pointers to achieve arbitrary remote code execution.

3.3.1 *Attack Overview*

Our attack, dubbed DOJITA (Data-Only JIT Attack), maliciously modifies the intermediate representation (IR) that is used internally by the JIT compiler. This results in the JIT engine generating an attacker-controlled native code. The high-level attack workflow is depicted in Figure 3.1: an adversary starts out by triggering a memory-corruption vulnerability, which grants the attacker read and write privileges within the application memory in the first step ①. Next, the attacker provokes the JIT engine to start an optimization pass for a particular, previously only interpreted dynamic function ②. By default, dynamic code is interpreted and the intermediate representation is only generated if run-time optimization is required for that function. This can usually be triggered automatically, e.g., by executing the same function in a loop to trigger the JIT compiler and compile the function to native code. In the next step ③, the attacker maliciously modifies the data objects that form the intermediate representation to correspond to an arbitrary, malicious payload. Since only code pointers are protected by control-flow integrity enforcement (regardless of the enforced granularity), this is possible despite CFI being deployed and active. The JIT compiler then uses the IR to optimize and eventually compile it ④ to machine code that can be executed natively. This generated native code will even be protected by a CFI policy accordingly, however, since the generated code is itself malicious the attacker only has to trigger its execution to invoke the exploit in the last step ⑤.

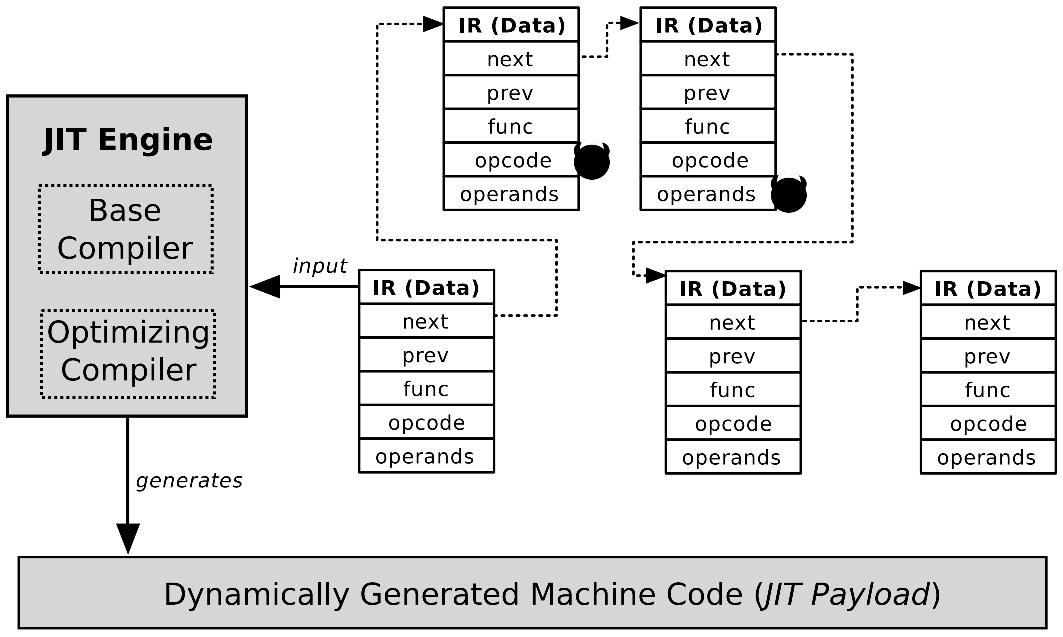


Figure 3.2: Within the JIT engine the currently optimized instruction stream is represented in the form of a doubly linked list of data objects. By inserting crafted or modifying existing objects an adversary can inject arbitrary instructions into the generated machine code. Since DOJITA [1] only involves data accesses it cannot be prevented by means of code-reuse defenses and completely bypasses CFI for the browser.

We note, that our attack emphasizes the threat of data-only attacks, despite defenses against control-flow hijacking being deployed and active.

3.3.2 *Corrupting the Intermediate Representation*

In our initial tests, we utilized the Chakra [153] JavaScript engine, which is used by Microsoft's Edge browser. Starting from a memory-corruption vulnerability, the goal of the attacker is to generate a malicious native payload in a pure data-only attack. This means that an adversary cannot modify any code pointers, since our threat model assumes a state-of-the-art code-reuse and code-injection defense to be deployed for the static and dynamic component of the application.

While reverse-engineering Chakra we focused on the JIT engine and how native code generation is triggered from the interpreted bytecode. Chakra's implementation of the intermediate representation uses a doubly linked list of data objects. Interestingly, this C++ object type already encodes the required data, which is used by the JIT engine to compile IR objects into native instruction sequences. In particular, `opcode` describes the type of operation, and `dst`, `src1`, and `src2` denote the respective operands to be used. A straightforward approach is to craft malicious memory objects using this layout and connecting them to the linked list. Hence, our attack then involves modifying the data pointers of existing list elements to point to the newly created, maliciously crafted C++ objects. The corrupted state of the linked list is also shown in Figure 3.2. Subsequently, the corrupted linked list will be used by the JIT engine to compile the IR to a sequence of machine code instructions which will already encode the attacker payload.

Unfortunately, the operation type field `opcode` is restricted to a set of benign instructions, such as arithmetic, branches, and memory accesses. For this reason, generating system calls is not supported directly and we resort to using unaligned instructions [10] by generating an `add` instruction with an argument that is set to the immediate value 12780815 (i.e., hexadecimal `0xC3050F`). When interpreted as a native opcode, this corresponds to `syscall; ret`. At the end, the attacker emits an indirect call with the target register pointing to the beginning of the immediate value of this `add` instruction, which causes the numerical constant representation of the operand to be executed as an unaligned instruction. This enables our payload to issue arbitrary system calls, and hence, interact fully with the system.

3.3.3 *Attack Framework*

In the prototype for our attack we leveraged the dynamic code generation to implement a framework for encoding arbitrary attack payloads. To this end, the attack payload is automatically converted into the corresponding memory objects, using the required memory layout and alignments for easy testing. We utilized an existing heap-corruption vulnerability in Chakra to obtain arbitrary read and write capabilities within application memory. We then disclose and modify a series of pointers using our framework to automatically compile the corresponding malicious structures and link them to the existing IR objects. Consequently, Chakra's JIT compiler engine will utilize our crafted memory objects while emitting native code.

State-of-the-art defenses [55, 148] cannot prevent our data-only jit attack, since they cannot differentiate between the malicious data objects and the genuine data objects in the JIT engine. During our experiments the attack succeeded with high probability.

3.4 ON THE PREVALENCE OF DATA-ONLY ATTACKS

Interestingly, a related data-only attack [154] was presented concurrently to our DOJITA attack that also targets the data structures used by Microsoft's JIT compiler. However, the presented approach differs from ours in so far as the authors target the temporary output buffer that contains the emitted native code for a certain time frame during compilation, and hence, this buffer is readable and writable during that time. After compilation the JIT engine moves and remaps the emitted code as readable and executable. As a response to this, Microsoft patched their JIT compiler by adding a check-sum to the generated instructions. If the check-sum of the remapped code does not match the check-sum of the buffer, the emitted JIT code is never executed. It is noteworthy to mention that while this defense mechanism stops the attack presented by Theori [154], our attack is completely unaffected and this patch does not prevent DOJITA. The reason is that we modify the IR before a check-sum is generated, and hence, the inserted check will always pass. This is why this latest patch cannot detect DOJITA.

3.5 POSSIBLE MITIGATIONS

To mitigate our attack, the JIT compiler has to be hardened against code-injection, code-reuse and data-only attacks. Achieving this requires to isolate all critical components of the JIT compiler from the main application, potentially containing a number of exploitable vulnerabilities. To isolate the JIT compiler one could use randomized segments protected through segment registers, or a separate process. Using the randomized segments to hide the compiler, its stack, and its heap would be possible, but would require a considerable effort to make sure that no information leak is possible. Using a separate process for the compiler, instead, requires a substantial redesign to support the asynchronous communication used during the resulting inter-process communication. Recent versions of Chakra have been redesigned [155] around an out-of-process compiler, which required 27 000 additional lines of code, compared to 640 000 lines of C/C++ code in the Chakra source. Using separate processes also means the processes have different address spaces and, thus, a higher overhead is required due to additional communication and synchronization. Moreover, a remote procedure call from the browser to the separate compiler process incurs additional latency.

On the other hand, in joint work with our colleagues we recently presented a mitigation that enforces the isolation through hardware by utilizing Intel SGX [1]. Existing browsers can be retrofitted with an SGX-based design, since it preserves the synchronous call semantics of existing code, while at the same time providing a clean separation. Moreover, the SGX enclave launched by the browser process is executed on the same core, so it does not require any action from the system scheduler to run. This also means that the enclave can leverage the data already stored in the CPU caches, resulting in a low overhead of less than 1%. Applying this retrofit to a JIT engine requires manual effort, however, they argue this one-time effort scales due to the similarity in the high-level design of major JIT engines and their limited number. As noted by the authors other mitigations, like CFI [61, 150, 156], require individual effort for each JIT engine as well.

4 BREAKING AND FIXING CFI IN OS KERNELS.

This chapter introduces a new data-only attack that generically bypasses CFI implementations in the OS context by subverting the page table, which is managed by the kernel as a data object. We also present the design and implementation of a novel randomization-based defense mechanism to protect page tables against such attacks. Through rigorous evaluation we demonstrate the effectiveness of our technique, called PT-Rand [2], and further show that it is also highly efficient.

4.1 OS KERNELS AS TARGETS OF DATA-ONLY ATTACKS

The OS represents the fundamental building block in the software stack underlying all user applications. To this end, the kernel abstracts and manages the complexity of real-world hardware and provides well-defined interfaces for user processes. In addition to that, the kernel usually runs with elevated privileges, and hence, constitutes a valuable target for attacks. This is aggravated by the fact that all major operating system kernels are implemented in unsafe programming languages. As explained in Section 2.3 this makes them prone to memory-corruption-based attacks. Since major kernels further offer increased support for new hardware features, diverse configurations, and a large number of devices, they also often incorporate third-party drivers which typically receive less maintenance.

As a response to various kernel attacks, additional architectural hardening techniques have been designed and deployed in practice. For instance, *Supervisor Mode Execution Protection* (SMEP) and *Supervisor Mode Access Protection* (SMAP) [157, 158] ensure that user pages cannot be accessed during kernel execution to protect against ret2usr attacks [159]. To defeat code-injection attacks, modern operating systems enforce the principle of non-writable code page [113]. However, this assumes that the integrity of page tables is assured. While a number of kernel-CFI implementations have been proposed to tackle OS-level code-reuse attacks [43, 45, 160], implementing code-reuse defenses in the kernel is more challenging. One of the reasons is that the data objects that enforce memory isolation (namely, *page tables*) are not protected against malicious accesses in the case of a memory-corruption vulnerability in the kernel.

4.2 ASSUMPTIONS AND THREAT MODEL

Our goal is to demonstrate the threat of data-only attacks against the OS kernel, especially in light of deployed state-of-the-art defenses being enforced. To this end, we assume that code-reuse attacks, including return-to-user [159] and return-to-dir [161] attacks in the kernel context are prevented by the respective architectural and software-based defenses. The adversarial setting for our attack is based on the following standard assumptions, which are consistent with the related work in that area [43, 45, 159–161].

- **Code-Injection and Code-Reuse Defenses.** We assume that the architecture provides a means to make code pages not writable through a globally enforced W⊕X policy (especially also for kernel code pages). We also require dynamically loaded kernel modules to be cryptographically signed. This means that an adversary cannot by default inject malicious code into the kernel.

  Further, we assume a code-reuse defense to be deployed in the kernel [41, 43, 45, 160] and that user pages are not accessible when executing kernel code, which is the case by default on recent architectures [92, 157, 158].

  Consequently, the attacker cannot modify the page tables by means of control-flow hijacking.

- **Kernel-Address Space Layout Randomization.** Most major operating systems additionally deploy a basic form of Kernel-ASLR (KASLR) by default [162–164], to randomize the location of the kernel code section, and we assume KASLR to be deployed and active.

- **Compromised User Process.** Next, we assume that the attacker gained full control over a user-space process, e.g., by exploiting a vulnerable application such as a webbrowser (cf., Chapter 3). Therefore, the attacker is able to execute arbitrary unprivileged code and can fully interact with the system , e.g., by issuing system calls.

- **Memory Corruption.** Finally, under our model the attacker has knowledge of a memory-corruption vulnerability in the kernel code. By triggering the vulnerability an adversary can read and write arbitrary kernel memory to attempt gaining kernel privileges. However, the enforced code-injection and code-reuse defenses prevent the attacker from hijacking the control flow of the kernel.

4.3 OUR PAGE TABLE EXPLOIT

To demonstrate the power of data-only attacks on page tables, we implemented a real world exploit to bypass the popular CFI-based protection for Linux kernel RAP [160]. Modern architectural [157, 158] and software-based [161, 165] hardening solutions cannot prevent data-only attacks, since they either focus on user memory not being accessed with kernel privileges or on mitigating code-reuse attacks. While control-flow hijacking attacks compromise code pointers, data-only attacks only affect non-control data, such as variables, data structures, and flags stored in memory.

4.3.1 *Attack Overview*

The high-level idea behind our attack is depicted in Figure 4.1. A user-level adversary can deactivate the enforced code-injection defense for a kernel code page by maliciously updating the respective page table object which controls the access control flags of that memory page (a mechanism explained in detail Section 2.1.2).

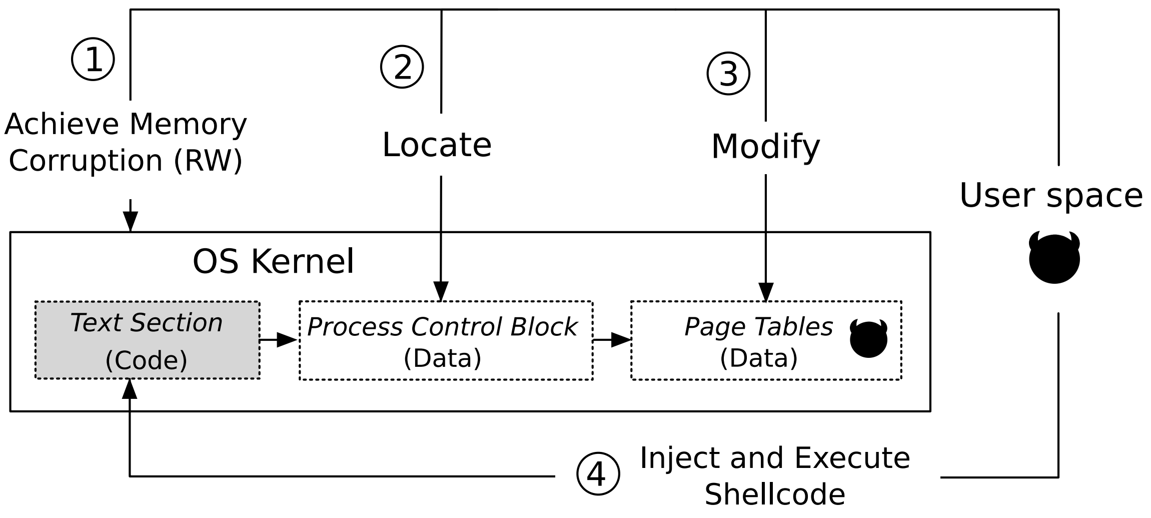

① Achieve Memory Corruption (RW)
② Locate
③ Modify
User space
OS Kernel
Text Section (Code)
Process Control Block (Data)
Page Tables (Data)
④ Inject and Execute Shellcode

Figure 4.1: Our page-table exploit enables code injection attacks in kernel mode despite CFI being deployed and active in the kernel. This motivated us to design an efficient randomization-based defense approach to complement kernel CFI implementations by protecting the page tables against such attacks.

To achieve this, an adversary triggers the memory-corruption vulnerability in the kernel code, which allows the attacker to read and write kernel memory. Next, the attacker has to scan and locate the kernel memory for data pointers to page table objects that control his own address space.

Since the page table is structured hierarchically, this only requires disclosing the address of the root node of the respective page table objects, since afterwards the page table entry for the kernel code page can be located in a straightforward manner by walking the page table in software. Finally, the attacker can update the permission bits stored in that page table entry to make the kernel code page writable and executable. From that point, code-injection attacks are possible again in the kernel context, and an adversary can execute arbitrary malicious code with kernel privileges.

4.3.2 *Exploiting the Kernel by Corrupting the Page Table*

To construct our attack we utilized the popular and freely available operating system Linux (v4.6) hardened with GRSecurity's RAP [160], which represents an open-source CFI implementation for the Linux kernel. This means, in our threat model the kernel is protected against control-flow hijacking and the attacker is not allowed to overwrite any code pointers. However, overwriting data and non-control pointers is still possible.

In that setting, corrupting the page table objects that control the virtual-address space mappings belonging to the attacking process requires several steps: first, the attacker has to disclose the memory location of the top-level page table leveraging the read-write primitive that is granted by the memory-corruption vulnerability in the kernel. This reference can be obtained by locating the process-control block (called `task_struct` in Linux) of the attacker's process. The `task_struct` contains a set of data structures for general book-keeping and management of processes.

Since many components in the kernel make use of these process descriptors these data structures are connected in a linked list, which is globally accessible. Additionally, the Linux kernel stores the process descriptor at a fixed offset below the kernel stack of that process, and hence, the memory location of the `task_struct` can be easily computed from a stack address.

Once the process descriptor has been located the attacker has to locate the top-level page table pointer from it in the second step. To this end, the `task_struct` contains a reference to a memory-management descriptor of the process, called `mm_struct`. This data structure maintains information about the virtual-address space of the respective process. Among other data, this structure contains the pointer to the top-level page table of the managed process, and the data field is called `pgd` (for page global directory).

Now we can walk the page table for the virtual address mapping the kernel code page which we intend to modify. Hence, in this third step the attacker leverages different parts of the virtual address as offsets into the respective page-table level until the page table entry is located. While conceptually straightforward this page walk may involve following up to four indirections in kernel memory (for x86 in 64bit mode).

Finally, once the page table entry for that kernel code page has been located, the attacker can set the bits that control the access permissions to readable, writable, and executable, enabling code injection in the kernel context. In the proof-of-concept implementation of our attack we chose to inject a malicious code into an unused system call function. In this way, we can trigger the malicious function from user space without ever hijacking the control flow of the kernel context.

At no point during our attack does an adversary require to modify a code pointer. Consequently, our data-only attack cannot be prevented by state-of-the-art defenses, such as CFI and KASLR.

4.4 PT-RAND: MITIGATING DATA-ONLY ATTACKS AGAINST THE PAGE TABLES

Given the importance of page tables for exploit mitigation, and the severe consequences incurred by our data-only attack, it is vital to provide efficient and effective protection of the page table against malicious modification. In fact, as our proof-of-concept attack demonstrates, such a defense is required to complement existing defense like CFI in the kernel. However, in our model the attacker can exploit a memory-corruption vulnerability to corrupt existing data structures. With arbitrary read and write capabilities in the kernel the attacker can alter the memory management structures to completely bypass existing and deployed defenses to again launch code-injection and code-reuse attacks.

4.4.1 *Design and Overview*

For this reason, we design a novel defense, called PT-Rand, to prevent data-only attacks against the page tables in the kernel, which is also illustrated in Figure 4.2. The high-level idea behind our defense is two-fold:

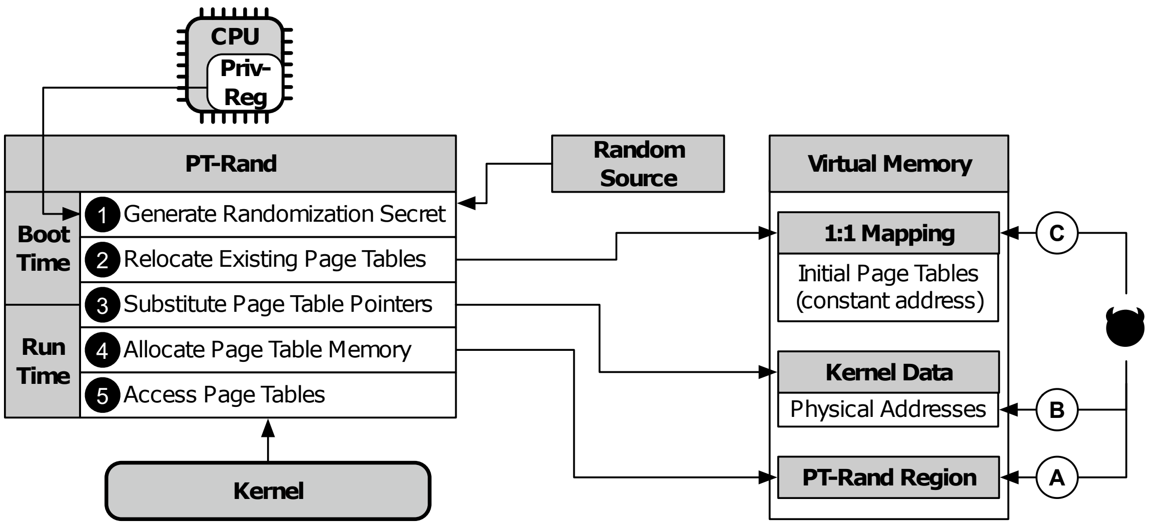

CPU
Priv-Reg
PT-Rand
Boot Time
Run Time
1 Generate Randomization Secret
2 Relocate Existing Page Tables
3 Substitute Page Table Pointers
4 Allocate Page Table Memory
5 Access Page Tables
Kernel
Random Source
Virtual Memory
1:1 Mapping
Initial Page Tables (constant address)
Kernel Data
Physical Addresses
PT-Rand Region
C
B
A

Figure 4.2: Our design of PT-Rand involves several boot-time and run-time components to set up the randomized mapping for page tables in the kernel.

1. We *randomize* the virtual addresses of all page table pages in the kernel using a common, randomized offset.

2. We provide *leakage resilience* by keeping this base offset in a privileged register and replacing all data pointers to page table objects with their physical addresses.

During system boot, PT-Rand first generates a randomization secret ❶, which is stored in a debug register. At that point in time, virtual memory is already enabled, and hence, we have to relocate all existing page tables ❷ to the hidden region. We use a debug register for two reasons: first, they require privileged code access, and second, they are not used during normal operation. Hence, to access the register that stores our random offset an adversary would have to a successful code-injection or code-reuse attack, which is prevented under our model. To make PT-Rand resilient against information-disclosure attacks, we additionally convert all pointers to physical addresses ❸. Whenever the system allocates a new set of page tables at run time, e.g., upon spawning a new process, the kernel invokes a set of wrapper functions that PT-Rand provides instead of the default page table allocation functions to additionally randomize ❹ the virtual address and substitute all pointers to page table references for that new process. If the kernel requires benign access to page table objects, for instance, to update a particular memory mapping, it leverages additional helper functions which will utilize the randomization secret to retrieve the respective page table page ❺. In this way, PT-Rand ensures that benign operation is not impeded and can still run efficiently.

It is noteworthy to recall that physical addresses cannot be used by any software once virtual memory is enabled by the OS during boot time. This means that all page table references in PT-Rand can only be accessed by using the randomization secret. This is a key point to the security of our scheme against data-only attacks: the attacker gains no knowledge from disclosing physical page table pointers. In particular, an adversary cannot use them to read the page tables during a page walk, or modify the required page table entry. Translating the physical page table references to usable, virtual addresses requires the randomization secret, which is stored securely in a register. To this end, we ensure that the register is never spilled.

4.4.2 *Challenges*

To enable PT-Rand we had to tackle a number of challenges as we explain in the following. In Section 4.5, we describe in detail how we address each challenge.
Initial Page Tables. At system start-up, the OS still uses physical memory. In order to enable virtual memory protection, the kernel temporarily allocates boot-time page tables, which are stored at a fixed location. Since, these page tables will still be used under certain circumstances, we have to relocate them in virtual address space at run time.
Page Table Allocation. Page tables are data objects that are dynamically allocated in the kernel. These objects are created by the page allocator, which is a central, low-level service in the kernel that manages physical pages. To randomize the memory pages where page tables are stored, we need to determine and instrument all kernel functions that allocate page tables.
Generating a Randomized Space. While the kernel needs to be able to locate randomized pages for performing benign changes, the attacker must not learn the new mapping. Consequently, we need to provide high entropy to avoid simple brute-force search. Furthermore, the new location of the page tables must not collide with other existing mappings in the virtual address space. This area must also be large enough to hold the page tables of all the processes running on the system.
Page Table References. Memory disclosure vulnerabilities allow the attacker to leak information about code and data pointers. Even fine-grained randomization schemes can be undermined if the attacker can map a single pointer to an instruction [17]. Hence, one of the main challenges in our design is to ensure that all references to page tables and the base address of the PT-Rand region are not leaked to the attacker. For this, we need to locate all page table references and replace them with physical addresses (❸ in Figure 4.2). Furthermore, we need to carefully handle benign page table changes by the kernel. Typically, the kernel processes page table pointers using virtual addresses on the kernel's stack. Since the stack is readable by the attacker, we need to provide a new mechanism to prevent leakage of these pointers.
Translation of Physical Addresses. At run-time, the kernel needs to repeatedly translate physical addresses to virtual addresses, e.g., during a page walk or when creating a page table entry. Normally, this would require a page walk for every access, which is a very costly operation as explained in Section 2.1.2. For this reason, the kernel linearly maps the complete physical memory in a so-called *one-to-one or direct mapping*. Henceforth, the kernel can easily translate physical addresses to virtual addresses by calculating the virtual address of the page from the the base address of this region plus the physical address of the page as an offset into the direct mapping. Since this base address is fixed, the attacker can easily locate and access physical addresses by calculating their corresponding one-to-one virtual address and search this mapping for references to page tables.

Handling of the Direct Mapping. As explained earlier, ❹ in Figure 4.2 removes the page tables from the direct map in the kernel to prevent an adversary from learning the page table location. However, there are some interesting technical challenges with this approach. In particular, removal of page tables is not per-se possible, due to the fact that the one-to-one mapping deploys so-called large pages of 2MB by default. Hence, simply removing the page leads to deletion of adjacent data not related to page tables. In addition, we need to identify all functions that access page tables via the direct mapping, and patch them to perform the translation based on the randomization secret, since without the one-to-one translation the kernel will not be able to locate randomized page table pages from their physical addresses anymore.

4.5 IMPLEMENTATION

Our design as presented in Section 4.4 requires low-level system modifications to the operating system kernel. We decided to prototype PT-Rand for the open-source Linux kernel. However, the concepts underlying our work on PT-Rand can be integrated into other contemporary operating systems. To this end, our kernel patch is comprised of 1382 insertions and 15 deletions across 45 source files.

Listing 4.3 shows how we integrate PT-Rand into the Linux kernel. We create wrapper functions for the page table allocator to randomize the virtual address of pages that contain page table entries. If the wrapper function is called to allocate memory which will be used to store page table entries, it allocates the memory at a random address in the PT-Rand region. The virtual address, pointing to this region, can only be computed by adding the randomization secret, which is stored in the third debug register. Pages for regular memory are still allocated in the direct map and their virtual addresses within this mapping are calculated by adding the base address of the direct map, called *physmap* in Linux, to the physical address of the regular page.

We create wrapper functions for those kernel functions that need to access page table memory. When the kernel starts executing, the PT-Rand initialization function will first generate the randomization secret based on the standard kernel function `get_random_bytes()`. We enable the kernel to use the hardware-based random number generator (HW-RNG) to avoid low entropy during boot time. Note, that since version 3.16 the Linux kernel incorporates the output of HW-RNGs for generating random numbers by default[1].

In the following, we present the implementation details of PT-Rand according to the challenges we outlined in Section 4.4.2.

4.5.1 *Page Table Allocations*

The main task of PT-Rand is to map memory which contains page tables to a random location within the PT-Rand region. Page table allocation involves two steps: (i) randomization of the initial pages, and (ii) randomization of memory allocations which are used to store newly created page tables.

1 https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=be4000

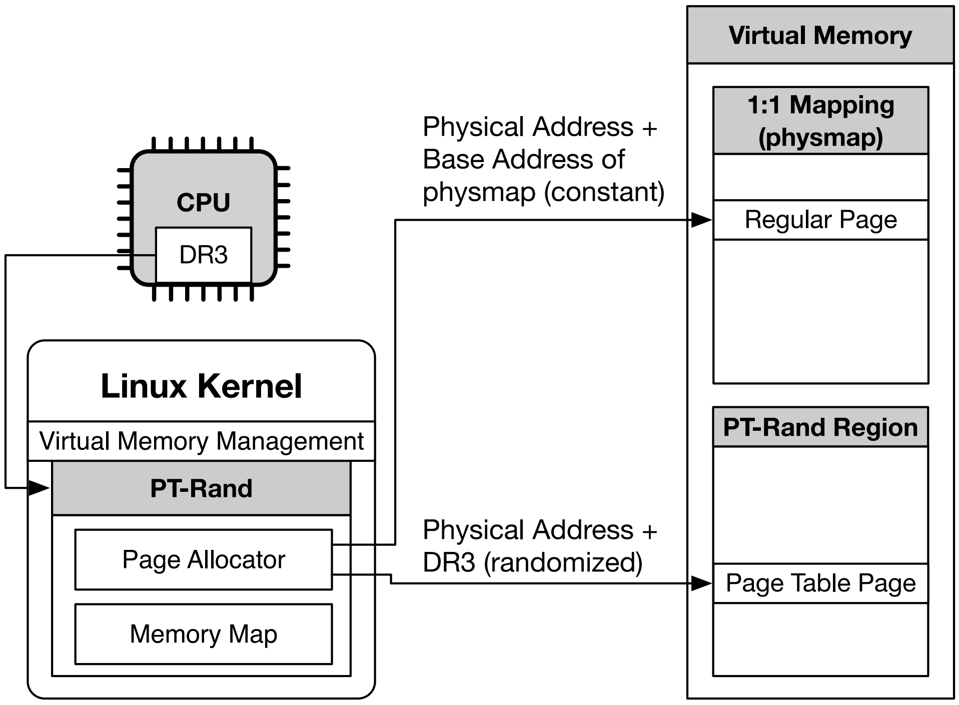


Figure 4.3: We modify the virtual memory location of page table pages by providing a randomization wrapper around the standard page allocator in the kernel. While randomized pages are removed from the one-to-one mapping, regular page allocation requests still fall within this region.

To complete the first step, we need precise knowledge of all existing references to the initial page tables, because after randomization these references need to be updated. The main challenge we faced is identifying all those references. To tackle this challenge, we followed a pragmatic approach: we reverse-engineered the kernel code execution after the location of the initial page tables have been randomized. Since every page table access based on an old reference leads to a kernel crash, we could determine the point of execution and associated kernel function which caused the crash. Thereafter, we inspected the kernel's source files and updated all references to use our new base address. After updating all references, kernel execution continued normally. In our extensive evaluation on different suites of benchmarks and complex software such as the Chrome browser (see Section 4.6.3) we have not experienced any kernel crashes.

To handle the second step, we extend the page table management functions in the kernel. Specifically, we create a wrapper function around the memory allocator for page tables. This allows us to modify their return values, i.e., they return physical addresses as a reference to the allocated memory rather than virtual addresses. Since there is no relation between physical and virtual memory addresses, the attacker cannot infer the location in the virtual memory by leaking the physical address.

We also create wrapper functions for every other kernel function that interacts with page tables to translate page table references (physical addresses) to virtual memory addresses before accessing the page tables.

4.5.2 *Generating a Randomized Area*

In order to provide sufficient protection against guessing attacks we require a high randomization entropy. While 64 bit architectures have a theoretical limit of 16EB of memory, current hardware is limited to support 256TB resulting in 48 bit randomization entropy.

The Linux kernel organizes the available virtual memory into different regions. Since the virtual-address space is really huge the kernel in fact does not use all of the available virtual memory. In particular, there are several memory regions that are unused and provide a reasonable amount (40 Bits) of freely addressable memory.[2] Our proof-of-concept implementation of PT-Rand utilizes one of these holes for the PT-Rand region to store the page table mappings.

4.5.3 *Page Table References*

As described in Section 4.2, the attacker can exploit kernel vulnerabilities to read from and write to kernel memory. However, these vulnerabilities do not allow the attacker to access content stored in registers. Hence, we can securely store the randomization secret into a dedicated register. For our proof-of-concept, we chose the fourth debug register `DR3`. We selected this register since it is only used for debugging purposes. It is noteworthy to mention that application debugging is still supported under PT-Rand. Typically, debuggers can use software and hardware breakpoints. The former are the default breakpoints and not affected by PT-Rand. For the latter, we only use one of the four available hardware breakpoints. Note that exploiting debugging facilities is a widely-accepted strategy when building defenses, e.g., TRESOR [166] or kBouncer [167]. Alternatively, we are currently exploring the feasibility of deploying any of the so-called model-specific registers (MSRs).

However, even though we store the base address in a privileged register, certain events (e.g., function calls) can spill temporary registers for several cycles to memory. As recently shown, this short time window can be exploited to undermine CFI checks [152]. PT-Rand tackles the attack by instructing the compiler to never spill registers which contain a randomized address. This is enabled by a GCC feature, called explicit register variables, which will always keep local variables in registers. However, given the complexity and many optimization techniques deployed by modern compilers, we can only guarantee that the above GCC compiler feature never leaks accordingly flagged variables, but not any intermediate calculation results. As a consequence, we are currently working on a GCC compiler extension that explicitly clears any intermediate results held in other registers.

2 Note, that such large holes will always exist for 64 Bit systems due to the vast amount of available virtual memory.

4.5.4 *Handling of the one-to-one mapping*

The typical page size is 4KB. However, the kernel also supports page sizes of 2MB or 1GB. In particular, for the Linux kernel, the one-to-one mapping is configured to use 2MB pages by default.

In PT-Rand, we rely on unmapping memory that contains page tables from the one-to-one mapping. This becomes challenging when 2MB pages are used because the page might contain other data than page table memory that should not be removed from the direct map. We tackle this challenge by reconfiguring the page size to 4KB pages at run time. However, in order to split a 2MB page into 4KB pages, we need to previously allocate 512 (i.e., 2MB divided by 4KB) new page table entries within the one-to-one mapping. Note that the 4KB split up only affects memory that contains page tables. For other memory parts, the kernel will continue to use large pages. Our performance evaluation in Section 4.6.3 indicates that this change has no impact on the overall performance. Next, we configure each entry to map the corresponding memory of the 2MB page, and adopt the permissions and other metadata. Finally, we update the page table hierarchy to use the 4KB page tables entries instead of the one 2MB entry. After the conversion, we can relocate and delete only those 4KB pages that contained page table entries.

4.5.5 *Translation of Physical Addresses*

Since the page tables are relocated by PT-Rand, they are no longer accessible through the one-to-one mapping. Hence, as described in Section 4.4.2, the kernel has to utilize two different mechanisms when translating physical addresses to virtual addresses, namely one for physical addresses of pages that contain page table entries, and another one to translate physical addresses for non-page table related memory. Fortunately, the kernel already keeps track of the properties of each individual physical page in a dedicated data structure called `memory map`. When we analyzed this structure, we noticed that certain bits of the `flag` field are not used. This allows us to quickly distinguish among the two different types of pages. Specifically, we reserve one bit to mark if a physical page has been removed from the one-to-one mapping by PT-Rand. In other words, if the bit is set, the kernel knows that the requested access is a page table related access which requires handling based on the PT-Rand region.

At run-time, kernel functions that need to translate a physical to a virtual memory address will check the `flag` field of the memory map. If the physical page is not accessible through the one-to-one mapping, the kernel function will use the randomization secret provided by PT-Rand to determine the virtual memory address. Otherwise, the function uses the default translation through the one-to-one mapping. Hence, PT-Rand preserves the high efficiency for the majority of the page requests through the direct mapping. In particular, we modified the `__va` macro to perform the check on the `flag` field. This function is the central point for translating physical to virtual addresses. PT-Rand does not cause any problems for external drivers, since external kernel drivers (e.g., graphic card drivers) are supposed to use these kernel functions to translate addresses.

During the implementation, we encountered that modifying `__va` raises another challenge: in the early boot phase, i.e., before PT-Rand relocates the initial page tables, a few kernel functions already invoke the modified macro. However, at this point of system state, the memory map is not yet initialized. Hence, the macro cannot yet access the `flag` field. We solved this problem by utilizing an unused macro called `__boot_va` which performs the same operation as the uninstrumented version of the `__va` macro. We patched all functions that are executed before the memory map is initialized to use the unmodified `__boot_va` macro.

4.6 EVALUATION

In this section, we present the evaluation results for PT-Rand. We first analyze the security with respect to randomization entropy and leakage resilience. We then we present our in-depth performance evaluation by conducting a series of benchmarks.

4.6.1 *Methodology*

To test our implementation, we assembled three configurational setups for use with a v4.6 kernel tree. The first is for development and early testing using the QEMU emulation software. The second setup is for more elaborate testing and also benchmarking on real hardware. Its configuration includes support for kernel modules, networking, and some hardware drivers, resulting in 658 options selected overall. The third setup is similar to the second, but additionally includes support for a graphical desktop.

Although it is of course possible to boot into a plain single user Linux environment[3], it is usually a lot more convenient to use one of the popular Linux distributions. We tested and evaluated the second setup using a default Debian Jessie installation (version 8.2). We used `fakeroot` and `make-kpkg` to build a `.deb` package for installation of our custom kernel via the `dpkg` package manager.

4.6.2 *Leakage Resilience and Randomization Entropy*

Since PT-Rand mitigates data-only attacks against the page tables by leveraging randomization, we analyze its security according the two main threats in any randomization-based scheme, i.e., *brute-force search* and *information-disclosure attacks*. Lastly, we evaluate the effectiveness of PT-Rand against our novel exploit.

BRUTE-FORCE SEARCH Since the security of randomization-based defenses depends on the amount of entropy, we ensure that PT-Rand achieves a high level of entropy. In particular, we chose the randomized memory region to be 40bits (or 1TB) in size. Since PT-Rand is a page-based randomization scheme, this leaves 28bits of entropy by default (assuming a page size of 4KB). To increase the chances of brute-force attacks a popular technique leverages random *spraying* of memory regions with targeted objects. In such a case, the *effective entropy* of the system

3 For instance, via booting with `init=/bin/sh`.

depends on the number of allocated pages. However, we also limit the number of possible guesses since we configure the kernel to reboot if an invalid memory access to kernel memory occured. This means that successfully manipulating a sprayed page table entry requires the adversary to not hit an unallocated page in the randomized region. We model the probability of such an attack according to an urn experiment without replacement and three cases: (1) hitting an unmapped page, (2) mapped pages, which do not map attacker-controlled memory, and (3) a mapped page that controls attacker virtual memory space. The attacker wins in the third case, may keep guessing in the second case, but loses in the first case. Hence, the probability of an adversary winning dependent on the number of allocated page tables is the sum of all probabilities in which the attacker never hits the first case.

We calculated this probability with up to 2^{16} memory pages and found that the attacker's success probability is about 0.0000003725%. Additionally, this probability grows linearly in the number of pages and will ultimately hit an out-of-memory situation, in which case the attacker loses again. Alternatively, we could thwart such attacks by providing an option to limit the amount of memory that can be allocated in the PT-Rand region. This limit would not necessarily affect the execution of the system under normal circumstances. In our tests even a big, virtualized server system under heavy load did not exceed this number of page table pages.

INFORMATION-DISCLOSURE ATTACKS The second threat to randomization-based defenses is information leakage. Under our model, an adversary is able to leak the address of any non-randomized kernel data structure, such as page table references contained in the process control blocks (`task_struct`). For this reason, we convert all page table references to physical addresses. Since virtual and physical addresses are completely independent, an adversary does not obtain any information from disclosing physical page table pointers, and hence, these references do not represent a source of information leakage any longer. Since we additionally patch all kernel functions dealing with page tables to expect a physical address and utilize the randomization secret to calculate a virtual address at run time, they cannot be leveraged by the attacker.

While temporarily spilled registers were previously demonstrated to compromise randomization-based defenses [152], we prevent access to the debug register that holds the randomization offset by patching all functions in the the Linux kernel that access this register by default. Additionally, we observe that the CPU will never write debug registers to memory during interrupt situations [157]. By further inspecting the functions that access the randomization secret, we also confirm that the secret is never leaked to the stack.

4.6.3 *Performance*

We measured the performance overhead incurred by PT-Rand based on SPEC CPU2006, LMBench, Phoronix, and Chromium benchmarks. All measurements are taken on an Intel Core i7-4790 CPU running at 3.60GHz with 8 GB RAM using Debian 8.2 with a recent Linux kernel, version v4.6.

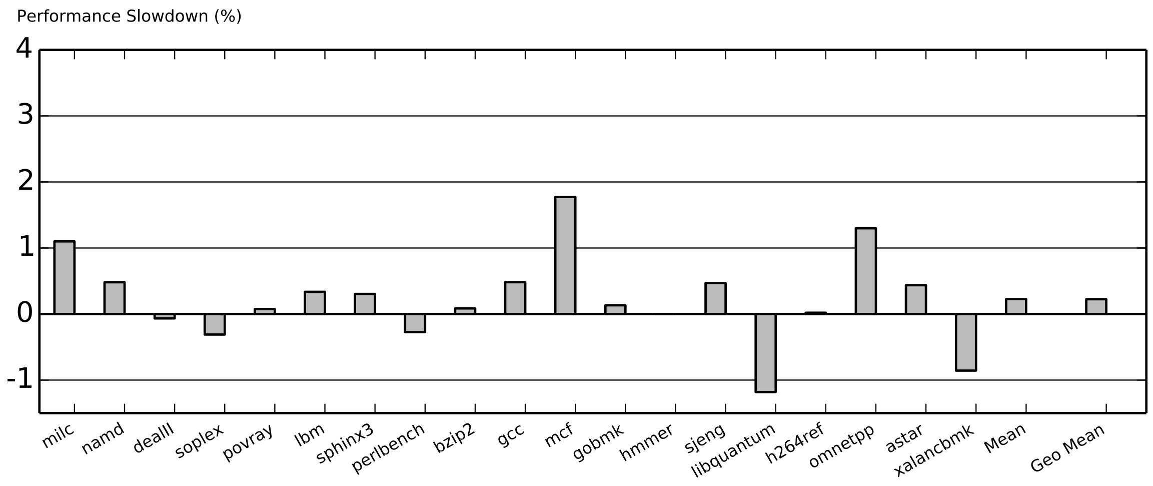


Figure 4.4: Overhead of page table randomization on SPEC CPU2006

SPEC CPU 2006 The SPEC CPU 2006 benchmarks measure the performance impact of PT-Rand on CPU-intensive applications. We executed the entire benchmark suite with the default parameters for reference tests: three iterations with reference input. We did neither encounter any problems during the execution (i.e., crashes) nor strong deviations in the results of the benchmarks. Figure 4.4 summarizes the performance impact of PT-Rand compared to a non-modified kernel. The average performance overhead of PT-Rand is only 0.22% with worst-case overhead of only 1.7%. This confirms the high efficiency and practicality of PT-Rand for contemporary systems.

Note that a few benchmarks perform marginally better when PT-Rand is applied. Such deviations have been also observed in prior work, and can be attributed to negligible measurement variances.

LMBENCH Most of our modifications affect the launch and termination phase of an application's lifecycle. This is due to the fact that PT-Rand needs to randomize the page tables at program start and remove them from the one-to-one mapping. When an application terminates, PT-Rand needs to de-randomize its page tables and make this memory again accessible through the one-to-one mapping. Hence, we additionally tested our approach using the popular LMBench micro benchmark suite [168] to assess the overhead for these critical phases. Specifically, LMBench collects timing information for process launch, fork, and exit. We measured an absolute overhead of less than 0.1 ms on average which is hardly noticeable to the end-user of a PT-Rand-hardened system

LMBench also includes other benchmarks, e.g., performance impact on memory accesses, system calls or floating point operations. We successfully executed all benchmarks and observed no measurable impact on the performance.

PHORONIX Besides SPEC CPU2006 and LMBench we measured the performance impact of PT-Rand with the Phoronix benchmark suite [169] which is widely used

Benchmark Name	Relative Overhead
IOZone	1.0%
PostMark	1.8%
OpenSSL	-2%
PyBench	-0.9%
PHPBench	-0.2%
Apache	0.8%

Table 4.1: Phoronix benchmark results.

to benchmark the performance of operating systems. Table 4.1[4] summarizes the results which are consistent with the results of the SPEC CPU2006 benchmarks.

CHROMIUM Finally, we measured the performance overhead for Google's browser Chromium in two scenarios: 1) we ran the popular browser benchmarking frameworks JetStream, Octane, and Kraken, to measure the run-time overhead for daily usage, and 2) we modified Chromium such that it terminates directly after loading to measure the load-time overhead. We repeated both experiments three times and determined the median to account for small variances.

For the Chromium web browser, we report a run-time overhead of -0.294% and a load-time overhead of 9.1%. The run-time overhead represents the arithmetic mean of 0.76% for JetStream, 1.183% for Kraken, and 2.825% for Octane.

The browser frameworks measure browser engine latency and load, with a focus on JavaScript execution. While these tests do not accurately measure the direct performance overhead of PT-Rand, they provide a first estimation of the performance impact on the popular end-user applications such as a web browser. Given only -0.294% overhead, we confirm that PT-Rand does not negatively impact performance of user applications.

To measure the load-time overhead, we simply added a `return` instruction in the main function of Chromium. This ensures that Chromium immediately terminates after it is completely loaded. We measured the elapsed time based on the Unix tool `time`. With less than 1 ms load-time overhead we assert that PT-Rand does not impair the user experience. We find these results to be in line with our LMBench test results for process creation and termination.

4.6.4 *Compatibility and Stability*

During the development and our tests of PT-Rand there were no incompatibilities between our modifications to the Linux kernel and GRSecurity's RAP [160]. In addition to executing the previously described benchmarks and tests, we assessed the overall stability and compatibility of our prototype implementation of PT-Rand by running various widely used applications. Throughout our tests there

4 Note that we excluded some of the benchmarks because we got errors when executing them on a vanilla system.

were no crashes or deviations from the default behavior without our patch applied to the kernel. Finally, we also ran the relevant test cases from the Linux Test Project (LTP) [170], which combines various scenarios that are designed to stress the system. Again, we did not find notable differences from the default system configuration.

5 AUTOMATICALLY UNCOVERING MEMORY CORRUPTION IN KERNEL CODE.

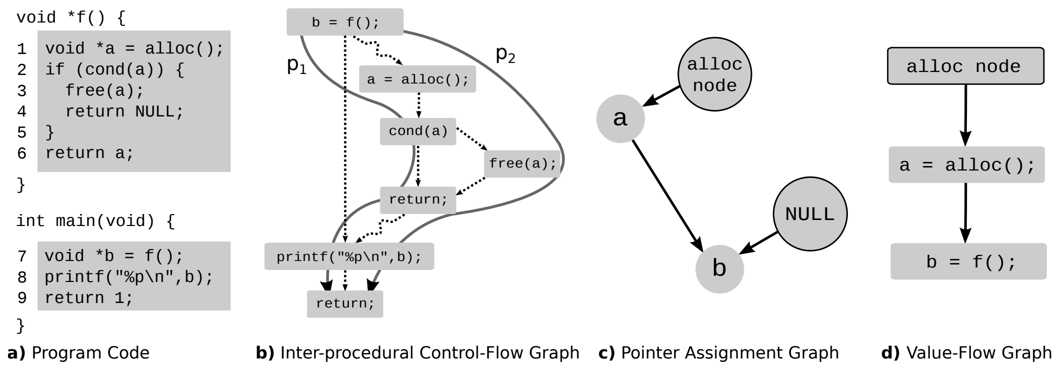
As demonstrated in the last chapter, data-only attacks resulting from memory-corruption vulnerabilities in the kernel represent a powerful exploitation tool against operating systems. While mitigating the adverse effects of these vulnerabilities by preventing exploitation at run time is often possible through the design of appropriate defenses, the root cause will still be present in the kernel even if those defenses are deployed. Moreover, attackers often find new, surprising ways of exploiting vulnerabilities through unanticipated exploits. In this chapter, we turn towards automatically identifying the root causes behind memory corruption, i.e., bugs in kernel code. We present K-Miner [3], the first data-flow analysis framework for operating system kernels. As we are able to demonstrate our compiler-based framework reliably uncovers different classes of memory-corruption vulnerabilities through state-of-the-art static analysis passes in complex real-world kernel code.

5.1 RUN-TIME DEFENSES VS. COMPILE-TIME VERIFICATION

Operating system kernels form the foundation of practically all modern software platforms. The kernel features many important services and provides the interfaces towards user applications. As explained in Chapter 2, the OS is usually isolated from applications through hardware mechanisms such as memory protection and different privilege levels in the processor. However, as demonstrated in the last chapter memory-corruption vulnerabilities in kernel code open up the possibility for unprivileged users to subvert control flow or data structures and take control over the entire system [14–16, 19]. For this reason, many defenses have been proposed [2, 43, 45, 156, 160, 161, 171–173] specifically for run-time protection of operating system kernels. Their goal is to provide countermeasures and secure the kernel against attacks exploiting memory corruption and most of these approaches can be loosely categorized as run-time monitors [43, 45, 156, 160, 172–174].

Typically, adversaries are modeled according to their capabilities, and reference monitors are then designed to defend against a specific class of attacks. For instance, CFI [40] is tailored towards control-flow hijacking attacks. However, CFI is not designed to protect against data-only adversaries resulting in a protection gap that allows for crucial attacks despite the presence of run-time monitors, such as CFI, in the kernel [2, 13–15]. Thus, a combination of many different defenses is required to protect the kernel against multiple classes of adversaries. Consequently, commodity operating systems will remain vulnerable to new types of software attacks as long as memory-corruption vulnerabilities are present in the code [12].

An alternative approach to employing run-time monitors is to ensure the absence of memory-corruption vulnerabilities by analyzing the system before deployment. This was shown to be feasible for small microkernels with less than



```
void *f() {
  void *a = alloc();
  if (cond(a)) {
    free(a);
    return NULL;
  }
  return a;
}
int main(void) {
  void *b = f();
  printf("%p\n",b);
  return 1;
}
```

Figure 5.1: Data-flow analyses utilize graphs to reason about program behavior at compile time.

10,000 lines of code [78, 79, 175], by building a formal model of the entire kernel and (manually) proving the correctness of the implementation with respect to this model. The invariants that hold for the formal model then also hold for the implementation. However, the formal correctness approach is impractical for commodity monolithic kernels due to their size and extensive use of machine-level code [176], which provides no safety guarantees. Even for small microkernels formulating such a model and proving its correctness requires more than 10 person years of work [78, 79]. While dynamic analyses are used to detect vulnerabilities in OS kernels rather successfully [177–179], static approaches have a major advantage: sound static analysis safely over-approximates program execution, allowing for strong statements in the case of negative analysis results. In particular, if no report is generated for a certain code path by a sound analysis, one can assert that no memory-corruption vulnerability is present. Hence, static analysis is also a practical and pragmatic alternative to formal verification, as it is able to offer similar assurances for real-world software by means of automated compile-time checks [180].

5.2 DATA-FLOW ANALYSIS

The general idea of static analysis is to take a program and a list of pre-compiled properties as input, and find all the paths for which a given property is true. Examples of such properties are liveness analysis, dead-code analysis, typestate analysis, or nullness analysis [181]. For instance, a nullness analysis for the program **a)** in Figure 5.1 could be answered by looking at its pointer-assignment graph (PAG) depicted in **c)**: since there is a path in which variable `b` is assigned a `NULL` value (`b` points to `NULL` in the PAG) a report will be issued. Another commonly used data structure is the inter-procedural control-flow graph (ICFG) in **b)** — limited to the procedures `main` and `f` for brevity — which propagates control flow globally. This can be used to conduct path-sensitive analysis. Finally, taint and source-sink analysis may track individual memory objects through their associated value-flow graph (VFG) in **d)**.

Static analysis for tracking individual values in a program is called *data-flow analysis*. Most data-flow analysis approaches follow a general concept, or framework, to analyze programs systematically. The naive approach is to enumerate all possible program paths and test each graph for a given property. This is commonly referred to as the *Meet Over all Paths* (MOP). In Figure 5.1, the MOP would be calculated by testing a property against the two alternative program paths p_1 and p_2. Unfortunately, in the general case the MOP solution was shown to be undecidable by reduction to the post correspondence problem [182].

However, the MOP can be approximated through a so-called *Monotone Framework*, which is a set of mathematically defined objects and rules to analyze program behavior. At the heart of the monotone framework is a *lattice*, i.e., a partial order with a unique least upper bound that must be defined over the domain of all possible values during program execution. Further, the analysis framework must specify monotone flow functions that define how program statements effect lattice elements (the monotony requirement ensures termination of the analysis). Finally, sets of domain elements (i.e., *values*) must be combined using a *merge operator*. A commonly used definition for points-to analysis is the domain of points-to sets for all pointers in a program. The flow functions then select all program statements, which potentially modify any pointer relations and specify their target transitions in the lattice. The merge operator defines how to combine the resulting points-to sets for such a transition. The notion of the monotone framework is significant for static program analysis: for any monotone framework, there exists a Maximum Fixed Point (MFP) solution, which safely approximates the MOP solution [182]. If the flow functions are distributive under the merge operator that is defined by the lattice, the MFP solution is identical to the MOP solution. The montone framework is then called a distributive framework, and data-flow analysis problems can be solved efficiently by solving a corresponding graph reachability problem [183].

As explained in Chapter 2, memory-corruption vulnerabilities represent a vast number of security relevant bugs for operating system software (e.g., [98, 99]). Run-time attacks exploit such bugs to inject malicious code, reuse existing code with a malicious input, or corrupt integral data structures to obtain higher privileges. Recall, that memory-corruption vulnerabilities are often classified according to their root defect: integer overflows (IO), use-after-free (UAF), dangling pointers (DP), double free (DF), buffer overflow (BO), missing pointer checks (MPC), uninitialized data (UD), type errors (TE), or synchronization errors (SE) are commonly listed classes of memory corruption [12, 13]. Any instance of memory corruption leaves the program vulnerable to run-time attacks. Each class represents a violation of well-defined program behavior as specified by the programming-language standard or the compiler, and hence, the violating program can exhibit arbitrary behavior at run time. For this reason an adversary with knowledge about any such vulnerability can exploit the program by deliberately triggering the error to achieve unintended, malicious behavior.

Also recall, that the main interface which exposes kernel code to a user space adversary are system calls [86]. In our approach we aim to combine different data-flow analysis passes for the classes listed above to report potential bugs in kernel code, which are accessible to a user space program through the system call interface. Since memory-corruption vulnerabilities account for many real-world exploits [12], we focus on reporting instances of dangling pointers (DP), user-after-free (UAF), and double free (DF) in our proof-of-concept implementation. For instance, dangling-pointer vulnerabilities occur when a memory address is assigned to a pointer variable, and the memory belonging to that address subsequently becomes unavailable, or invalid. For heap allocations this can happen, e.g., when a memory location is freed but the pointer is still accessible. For stack-based allocations this happens when the stack frame containing the allocated object becomes invalid, e.g., due to a nested return statement in or below the scope of the allocation. Our framework is extensible such that new analyses passes can be integrated to search for additional vulnerability classes (cf., Section 5.7).

5.3 PROBLEM DESCRIPTION

However, static analysis faces severe scalability challenges, and hence, all analysis frameworks for kernel code are limited to *intra-procedural* analysis, i.e., local checks per function. In particular, there are five popular analysis frameworks targeting Linux: Coccinelle [184], Smatch [185], TypeChef [186], APISAN [187], and EBA [188]. None of these support *inter-procedural* data-flow analyses, which are required to conservatively approximate program behavior, and reliably uncover memory corruption caused by global pointer relationships. For instance, TypeChef focuses heavily on the construction variability-aware program representations and does not conduct any data-flow analysis on top of the resulting representation. EBA is restricted to checking individual source files—similar to Coccinelle and Smatch—and hence, cannot analyze the entire kernel image. [1] One of the main reasons why data-flow analysis for operating system kernels is beyond the capabilities of all existing approaches is the huge size and complexity of the code base in monolithic kernels. Currently, Linux comprises over 24 million lines of code [189]. Just compiling a common distribution kernel takes several hours, even on top-of-the-line hardware. While some of the existing tools allow for limited, text-based analysis of kernel code, these are conceptually restricted to local intra-procedural (i.e., per-function) or simpler file-based analysis. These limitations are due to the fact that the number of possible data flows to be analyzed generally grows exponentially with the size of the program code, and hence, static analysis approaches face severe scalability challenges [190–192]. At the same time, analysis methods have to take all paths and states into account to remain sound, and hence, pruning or skipping certain parts of the code would lead to unreliable results. This is why the resource requirements for conducting such analyses in the Linux kernel quickly outgrows any realistic thresholds. As a result, global and inter-procedural analysis of kernel code, which is required to uncover memory corruption reliably, remains largely an unsolved problem for all existing approaches.

5.4 DESIGN OF K-MINER

In this section, we explain the goals and assumptions in our model, introduce the high-level design, and elaborate on challenges to enable precise, inter-procedural static analysis of complex, real-world kernels.

1 Moreover, EBA specializes in much simpler flow-insensitive analyses.

5.4.1 *Goals and assumptions*

With K-Miner we aim to identify and report potential memory-corruption bugs in the kernel's user-space interface, so that developers can fix them before shipping code that includes such vulnerabilities. Regarding potential malicious processes at run time we make the following standard assumptions:

- The attacker has control over a user-space process and can issue all system calls to attack the kernel through the subverted process.
- The operating system is isolated from user processes, e.g., through virtual memory and different privilege levels. Common platforms like x86 and ARM meet this requirement.
- An adversary cannot insert malicious code into the kernel through modules, because modern operating systems require kernel modules to be cryptographically signed [193–195].
- K-Miner should reliably report memory-corruption vulnerabilities that can be triggered by a malicious process.

Our assumptions force the attacker to exploit a memory-corruption vulnerability in the kernel code to gain kernel privileges through a purely software-based attack. The goal of K-Miner is to systematically scan the system call interface for these vulnerabilities.

Since real-world adversaries are not limited to software vulnerabilities, it is important to note that even with a completely verified kernel (e.g., seL4) hardware attacks such as rowhammer [63, 72] still pose a serious threat to the integrity of the kernel. However, for our work we consider hardware implementation defects to be an orthogonal problem [5].

5.4.2 *Overview*

K-Miner is a static analysis framework for commodity operating system kernels. We provide a high-level overview in Figure 5.2.

Our framework builds on top of the existing compiler suite LLVM. The compiler (cf., step ①) receives two inputs. First, a configuration file, which contains a list of selected kernel features. This configuration file enables the user to select or deselect individual kernel features. When a feature is disabled, its code is not included in the implementation. Hence, an analysis result is only valid for a specific pair of kernel code and configuration file. Second, the compiler suite parses the kernel code according to the configuration. It syntactically checks the code and builds an abstract syntax tree (AST). The compiler then internally transforms the AST into a so-called intermediate representation (IR), which essentially represents an abstract, hypothetical machine model. The IR is also used for analyzing and optimizing kernel code through a series of transformation passes.

In step ②, the compiler suite passes the IR of the kernel as an input to K-Miner, which starts to statically check the code by going through the list of all system calls. For every system call, K-Miner generates a call graph (CG), a value-flow

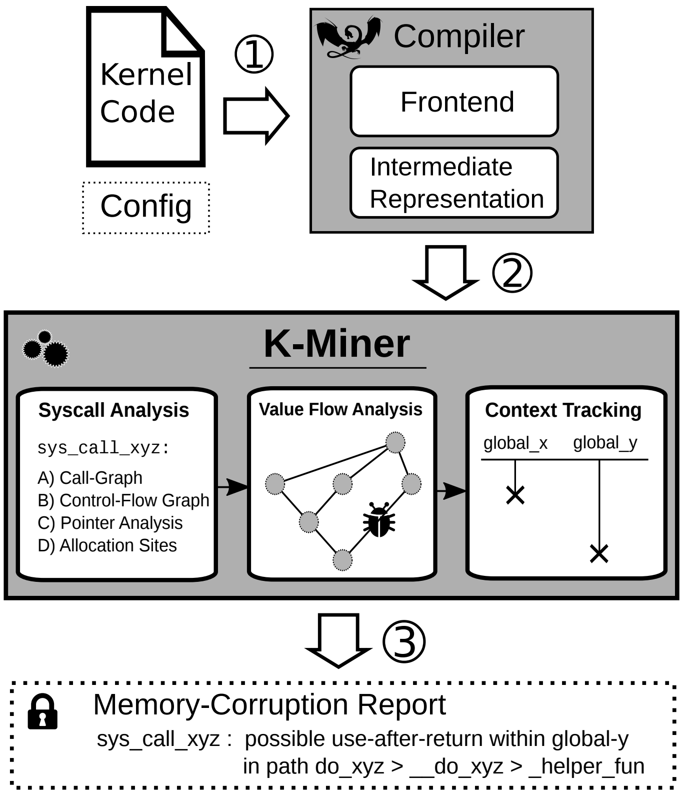


Figure 5.2: Overview of the different components of K-Miner.

graph (VFG), a pointer-analysis graph (PAG), and several other internal data structures by taking the entry point of the system call function as a starting point. Additionally, we compute a list of all globally allocated kernel objects, which are reachable by any single system call. Once these data structures are generated, K-Miner can start the actual static analysis passes. There are individual passes for different types of vulnerabilities, e.g., dangling-pointer, use-after-free, double-free, and double-lock errors. All of these passes analyze the control flow of a specific system call at a time, utilizing the previously generated data structures. The passes are implemented as context-sensitive value-flow analyses: they track inter-procedural context information by taking the control flow of the given system call into account and descend in the call graph.

If a potential memory-corruption bug has been detected, K-Miner generates a report, containing all relevant information (the affected kernel version, configuration file, system call, program path, and object) in step ③.

5.4.3 *Uncovering Memory Corruption*

The goal of K-Miner is to systematically scan the kernel's interface for different classes of memory-corruption vulnerabilities using multiple analysis passes, each tailored to find a specific class of vulnerability. The individual analysis pass utilizes data structures related to the targeted vulnerability class to check if certain conditions hold true. Reasoning about memory and pointers is essential for analyzing the behavior of the kernel with respect to memory-corruption vulnerabilities, hence, the data base for all memory objects (called *global context*) and the pointer-analysis graph represent the foundation for many analysis passes. Individual memory objects are instantiated at allocation sites throughout the entire kernel and the variables potentially pointing to them are tracked per system call using the PAG. Forward analysis then reasons about the past behaviour of an individual memory location, whereas a backward analysis determines future behaviour (since a forward analysis processes past code constructs before processing future code and vice versa).

We can also combine such analysis passes, for instance, to find double-free vulnerabilities: first, we determine sources and sinks for memory objects, i.e., allocation sites and the corresponding free functions respectively. We then process the VFG in the forward direction for every allocation site to determine reachable sinks. Second, we reconstruct the resulting paths for source-sink pairs in the execution by following sinks in the backward direction. Finally, we analyze the forward paths again to check for additional sinks. Since any path containing more than one sink will report a duplicate de-allocation this approach suffers from a high number of false positives. For this reason, we determine if the first de-allocation invocation *dominates* (i.e., is executed in every path leading to) the second de-allocation invocation in the validation phase.

In similar vein we provide passes that are checking for conditions indicating dangling pointers, use-after-free, and double-lock errors. We provide more detailed examples for the implementation of such passes in Section 5.5.

5.4.4 *Challenges*

Creating a static analysis framework for real-world operating systems comes with a series of difficult challenges, which we briefly describe in this section. In Section 5.5 we explain how to tackle each challenge in detail.

Global state.
Most classes of memory-corruption vulnerabilities deal with pointers, and the state or type of the objects in memory that they point to. Conducting inter-procedural pointer analyses poses a difficult challenge regarding efficiency. Because inter-procedural analysis allows for global state, local pointer accesses may have non-local effects due to aliasing. Since our analyses are also flow-sensitive, these aliasing relationships are not always static, but can also be updated while traversing the control-flow graph. To enable complex global analyses, we make use of sparse program representations: we only take value flows into account that relate to the currently analyzed call graph and context information.

Huge codebase.
The current Linux kernel comprises more than 24 million lines of code [189], supporting dozens of different architectures, and hundreds of drivers for external hardware. Since K-Miner leverages complex data-flow analysis, creating data structures and dependence graphs for such large amounts of program code ultimately results in an explosion of resource requirements. We therefore need to provide techniques to reduce the amount of code for individual analysis passes without omitting any code, and allowing reuse of intermediate results. By partitioning the kernel accord-

ing to the system call interface, we are able to achieve significant reduction of the number of analyzed paths, while taking all the code into account, and allowing reuse of important data structures (such as the kernel context).

False positives.
False positives represent a common problem of static analysis, caused by too coarse-grained over approximation of possible program behavior. Such over approximation results in a high number of reports that cannot be handled by developers. K-Miner has to minimize the number of false positives to an absolute minimum. As the number of false positives depends greatly on the implementation of the individual analysis passes we carefully design our analyses to leverage as much information as possible to eliminate reports that require impossible cases at run time, or make too coarse-grained approximations. Moreover, we sanitize, deduplicate, and filter generated reports before displaying them for developers in a collaborative, web-based user interface.

Multiple analyses.
A comprehensive framework needs to be able to eliminate all possible causes of memory corruption. This is why K-Miner must be able to combine the results of many different analyses. Additionally, individual analyses may depend on intermediate results of each other. Hence, our framework has to be able to synchronize these with respect to the currently inspected code parts. To this end we leverage the modern pass infrastructure of LLVM to export intermediary results and partially re-import them at a later point in time.

5.5 IMPLEMENTATION

In this section we describe our implementation of K-Miner, and how we tackle the challenges mentioned in Section 5.4.4. Our framework builds on the compiler suite LLVM [196] and the analysis framework SVF [197]. The former provides the basic underlying data structures, simple pointer analysis, a pass-infrastructure, and a bitcode file format which associates the source language with the LLVM intermediate representation (IR). The latter comprises various additional pointer analyses and a sparse representation of a value-flow dependence graph.

Since it is possible to compile the Linux kernel with LLVM [198], we generate the required bitcode files by modifying the build process of the kernel, and link them together to generate a bitcode version of the kernel image. This image file can then be used as input for K-Miner. Figure 5.3 depicts the structure of our framework implementation. In particular, it consists of four analysis stages: in step ①, the LLVM-IR is passed to K-Miner as a `vmlinux` bitcode image to start a pre-analysis, which will initialize and populate the global kernel context. In step ②, this context information is used to analyze individual system calls. It is possible to run multiple analysis passes successively, i.e., our dangling pointer, use-after-free, and double-free checkers, or run each of them independently. In step ③, bug reports are sanitized through various validation techniques to reduce the number of false positives. In step ④, the sorted reports are rendered using our vulnerability reporting engine. In the following, we describe each of the steps in more detail and explain how each of them tackles the challenges identified in the previous section.

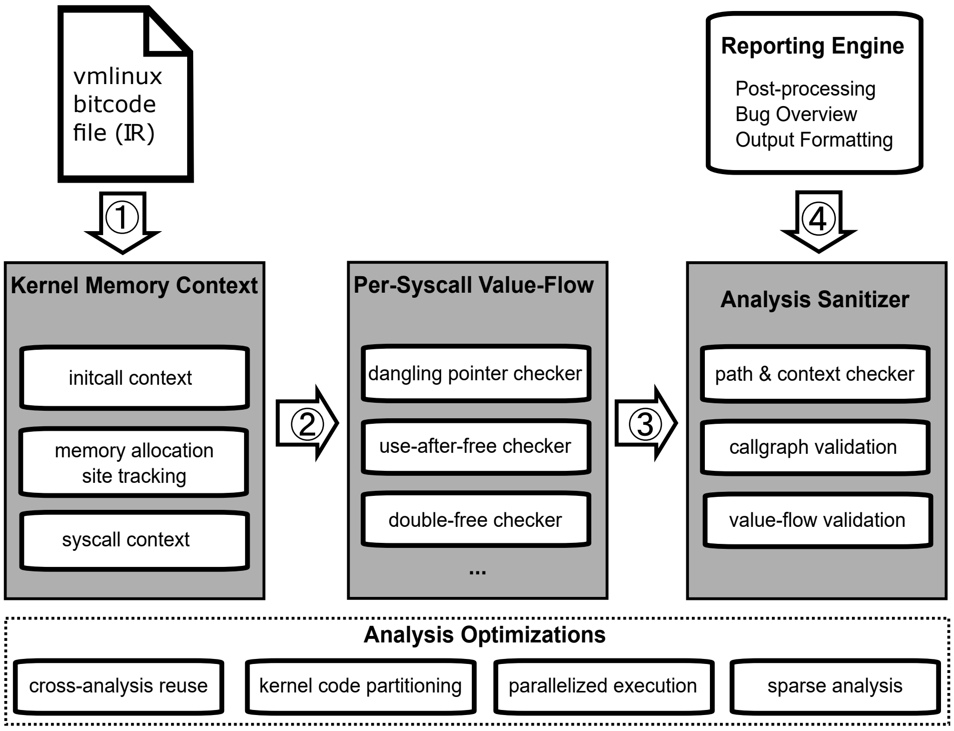


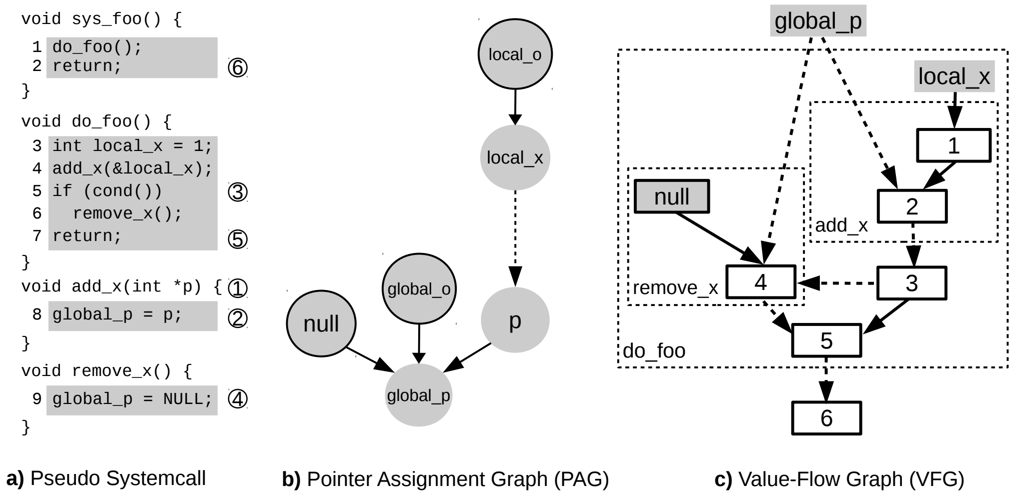
Figure 5.3: Overview of the K-Miner implementation: we conduct complex data-flow analysis of the Linux kernel in stages, re-using intermediate results.

5.5.1 *Global Analysis Context*

The global context stored by K-Miner essentially represents a data base for all the memory objects that are modeled based on the source code. Managing global context information efficiently is a prerequisite to enable analysis of highly complex code bases such as the Linux kernel. Additionally, we have to ensure that the context is sufficiently accurate to support precise reporting in our subsequent analysis. This is why the pre-analysis steps of our framework resemble the execution model of the kernel to establish and track global kernel context information.

Initializing the Kernel Context: The kernel usually initializes its memory context at run time by populating global data structures, such as the list of tasks or virtual memory regions during early boot phase. This is done by calling a series of specific functions, called *Initcalls*. These are one-time functions which are annotated with a macro in the source files of the kernel. The macro signals the compiler to place these functions in a dedicated code segment. Functions in this segment will only be executed during boot or if a driver is loaded. Hence, most of the memory occupied by this segment can be freed once the machine finished booting [199]. To initialize the global kernel context, we populate global kernel variables by simulating the execution of these initcalls prior to launching the analyses for each system call. The resulting context information is in the order of several hundred megabytes, therefore, we export it to a file on disk and re-import it at a later stage when running individual data-flow analysis passes.

Tracking Heap Allocations: Usually, user space programs use some variant of `malloc` for allocating memory dynamically at run time. There are many different



```
void sys_foo() {
  do_foo();
  return;                ⑥
}
void do_foo() {
  int local_x = 1;
  add_x(&local_x);
  if (cond())            ③
    remove_x();
  return;                ⑤
}
void add_x(int *p) {     ①
  global_p = p;          ②
}
void remove_x() {
  global_p = NULL;       ④
}
```

Figure 5.4: Example of a Dangling Pointer vulnerability in a (simplified) system call definition.

methods for allocating memory dynamically in the kernel, e.g., a slab allocator, a low-level page-based allocator, or various object caches. To enable tracking of dynamic memory objects, we have to compile a list of allocation functions which should be treated as heap allocations. Using this list K-Miner transforms the analyzed bitcode by marking all call sites of these functions as sources of heap memory. In this way kernel memory allocations can be tracked within subsequent data-flow analysis passes.

Establishing a Syscall Context: Because subsequent analysis passes will be running per system call, we establish a dedicated memory context for each of them. We do this by collecting the uses of any global variables and functions in each of the system call graphs. By cross-checking this context information against the global context, we can establish an accurate description of the memory context statically.

5.5.2 *Analyzing Kernel Code Per System Call*

Although analyzing individual system calls already reduces the amount of relevant code significantly, the resource requirements were still unpractical and we could not collect any data-flow analysis results in our preliminary experiments. For instance, conducting a simple pointer analysis based on this approach already caused our server system to quickly run out of memory (i.e., using more than 32G of RAM). Through careful analysis we found that one of the main causes for the blow-up are function pointers: in particular, the naive approach considers all global variables and functions to be reachable by any system call. While this approximation is certainly safe, it is also inefficient. We use several techniques to improve over this naive approach, which we describe in the following.

Improving Call Graph Accuracy: We start with a simple call-graph analysis, which over-approximates the potential list of target functions. By analyzing the IR of all functions in the call graph we determine if a function pointer is reachable (e.g., by being accessed by a local variable). This allows us to collect possible target functions to improve the precision of the initial call graph. Based on this list, we perform a two-staged pointer analysis in the next step.

Flow-sensitive Pointer-Analysis: To generate the improved call graph we first perform a simple inclusion-based pointer analysis to resolve the constraints of the function pointers collected earlier. To further improve the precision, we conduct a second pointer analysis while also taking the control flow into account. This again minimizes the number of relevant symbols and yields a very accurate context for individual system calls. We store these findings as intermediate results per system call which can be used by subsequent data-flow analysis passes.

Intersecting Global Kernel State: Finally, we combine the previously identified context information for a system call with the global kernel context. We do this by determining the global variables of a system call that contain missing references and intersecting these with the list of variables of the global kernel context populated earlier. While possibly increasing the context information our precision improvents prevent an infeasible blow-up in this last step.

5.5.3 *Minimizing False Positives*

False positives are a common problem in static analysis and frequently occur when over-approximating program behavior: for instance, an analysis may assume an alias relationship between pointers that do not co-exist at run time, if the control flow is not taken into account. In the following, we explain how we designed our analysis to be precise and reduce the number of false positives, using dangling pointers as an example. We also provide details on how K-Miner sanitizes the resulting bug reports to further limit the number of false positives.

Precise Data-Flow Analysis: Figure 5.4 a) shows the code of a pseudo system call with a dangling pointer bug. In step ①, the address of the local variable in `do_foo` is copied into the parameter `p` of `add_x` and subsequently stored in the global pointer `global_p` in step ②. In step ③, we can see that `remove_x` will only be called conditionally. Hence, there is a path for which `global_p` still points to the address of a local variable after execution of `do_foo` has returned. Looking at the PAG in Figure 5.4b) reveals that `local_o` and `global_o` represent the abstract memory objects behind these possible pointer values. The (simplified) VFG in Figure 5.4c) shows the corresponding value flows. Our algorithm to find these kinds of bugs consists of two phases: first, we traverse the VFG in forward order starting from local nodes. A reference to a local node leaves its valid scope, if the number of function exits is greater than the number of function entries after traversing the entire path. For the node `local_x` we can see, that there is one entry to `add_x`, an exit from `add_x`, and an exit from `do_foo` at the end of the path. Consequently, there is a path for which `local_x` leaves its valid scope, i.e., `local_x` → ① → ② → ③ → ⑤ → ⑥.

In the second phase we traverse the VFG in backward direction to find (global or local) references to this node, since any such reference represents a dangling pointer. In this case the second phase yields the path ⑥ → ⑤ → ③ → ② → `global_p`. By querying the PAG dynamically during backward traversal we avoid visiting edges that do not belong to the currently tracked memory location such as ⑤ → ④. This allows us to minimize inaccuracies resulting from over-approximation.

				Magnitude of Analysis			Report Results		
Vers.	LOC	Size	Time	#Funs	#Vars	#Refs	DP	UAF	DF
3.19	15.5M	280M	796.69s	99K	433K	>5M	7 (40)	3 (131)	1 (13)
4.2	16.3M	298M	1435.62s	104K	466K	>6M	11 (46)	2 (106)	0 (19)
4.6	17.1M	298M	1502.54s	105K	468K	>6M	3 (50)	2 (104)	0 (31)
4.10	22.1M	353M	1312.41s	121K	535K	>7M	1 (30)	2 (105)	0 (22)
4.12	24.1M	364M	2164.96s	126K	558K	>7.4M	1 (24)	0 (27)	1 (24)

Table 5.1: Overview of the specifications, resource requirements, and results for the different kernel versions and data-flow passes we used in our evaluation of K-Miner.

We store the respective path information along with the list of nodes and contexts they were visited in as memory-corruption candidate for sanitizing and future reporting.

Sanitizing Potential Reports: Upon completion of the data-flow analysis, we cross-check the resulting candidates for impossible conditions or restrictions which would prevent a path from being taken during run-time execution. Examples for such conditions include impossible call graphs (e.g., call to function g preceding return from function f), or invalid combinations of context and path information. Additionally, we eliminate multiple reports that result in the same nodes for different contexts by combining them into a single report.

5.5.4 *Efficiently Combining Multiple Analyses*

To enable the efficient execution of multiple data-flow analyses, our framework makes heavy use of various optimizations and highly efficient analysis techniques as we describe below.

Using Sparse Analysis: An important data structure in our data-flow analysis is the value-flow graph, which is a directed inter-procedural graph tracking any operations related to pointer variables. The VFG captures the def-use chains of the pointers inside the kernel code to build a *sparse* representation for tracking these accesses. The graph is created in four steps: first, a pointer analysis determines the points-to information of each variable. Second, the indirect definitions and uses of the address-taken variables are determined for certain instructions (e.g. store, load, callsite). These instructions are then annotated with a set of variables that will be either defined or used by this instruction. Third, the functions are transformed into Static Single Assignment form using a standard SSA conversion algorithm [200]. Finally, the VFG is created by connecting the def-use for each SSA variable and made partially context-sensitive by labeling the edges of the callsites. Using this sparse VFG representation in a partially context-sensitive way enables us to conduct precise analysis while reducing the amount of code.

Revisiting Functions: Using different analysis passes, functions are potentially visited multiple times with different values as an input. However, one function might call dozens of other functions and forwarding all the resulting nodes multiple times in the same way would be very inefficient. Our analysis reduces the amount of nodes that have to be forwarded by processing a function only once for all of its possible contexts and storing the intermediate results. If a function entry node is requested by an analysis with a given context, the analysis checks if this node was already visited and re-uses the pre-computed results.

Parallelizing Execution: Because certain analysis steps can actually run independently from each other, we implemented another optimization by parallelizing the forwarding and backwarding processes using OpenMP [201]. OpenMP provides additional compiler directives that allow the definition of parallel regions in the code. In this way, we process some of the heavy container objects used during the analysis in parallel.

5.6 EVALUATION

In this section, we evaluate and test our static analysis framework for real-world operating system kernels. We run our memory-corruption checkers against five different versions of the Linux kernel, using the default configuration (`defconfig`). Our test system running K-Miner features an Intel Xeon E5-4650 with 8 cores clocked at 2.4GHz and 32G of RAM. Table 5.1 shows an overview of the analyzed Linux kernel specifications and results: on average, our framework needs around 25 minutes to check a single system call (cf., *Avg. Time* in Table 5.1). This means that a check of the entire system call interface on this server with all three analyses takes between 70 and 200 hours for a single kernel version. [2] In our experiments, K-Miner found 29 possible vulnerabilities, generating 539 alerts in total, most of which were classified as false positives (total alerts are given in parenthesis in Table 5.1). [3] Next, we will evaluate the coverage and impact of those reports and afterwards also discuss the performance of our framework in more detail.

5.6.1 *Security*

Since K-Miner aims to uncover memory-corruption vulnerabilities in the context of system calls, we investigate its security guarantees by inspecting the coverage of the underlying graph structures. To demonstrate practicality, we also present some of the publicly known vulnerabilities we were able to find statically using our framework.

Coverage: Our goal is to uncover all possible sources of memory corruption that are accessible via the system call interface that the kernel exposes to user processes. Hence, we have to ensure that the analysis passes for a certain class of vulnerabilities have access to all relevant information required to safely approximate run-time behavior of the kernel. At the time of writing, our framework contains passes for DP, DF, and UAF, hence, other sources of memory corruption are not covered in this evaluation. However, K-Miner is designed to be extensible and we are work-

2 Largely depending on the respective kernel version as seen in the average time per system call in Table 5.1.

3 Additionally, we are still investigating 158 memory-corruption alerts for the most recent version of Linux.

ing on implementing further analysis passes to cover all remaining vulnerability classes.

The most important factors for the coverage of our three analysis passes are their underlying analysis structures, i.e., PAG, VFG, and context information. Because the inter-procedural value-flow graph and the context information are derived from the pointer-analysis graph, their accuracy directly depends on the construction of the PAG. Our pointer analysis makes two assumptions: 1) partitioning the kernel code along its system call graph is sound, and 2) deriving kernel context information from init calls is complete. We would like to note that verifying both assumptions requires a formal proof, which is beyond the scope of this work. However, we sketch why these assumptions are reasonable in the following.

The first assumption is sensible, because system calls are triggered by individual processes to provide services in a synchronous manner, meaning that the calling process is suspended until execution of the system call finishes. While interrupts and inter-process communication may enable other processes to query the kernel asynchronously, this is orthogonal to the partitioning of kernel code, because these operate under a different context. In particular, our framework allows analyses to take multiple memory contexts into account, e.g., to uncover synchronization errors. Individual analysis passes then have to ensure that the associated contexts are handled accordingly.

Our second assumption is derived from the design of the analyzed kernel code. The init call infrastructure for Linux is quite elaborate, using a hierarchy of different levels that may also specify dependencies on other init calls. Additionally, init calls are used in many different scenarios, e.g., to populate management structures for virtual memory and processes during early boot, or to instantiate drivers and modules at run time. By including all init call levels following their hierarchical ordering in the pre-analysis phase, we ensure that the relevant context information is recorded and initialized accordingly.

Real-world Impact: We cross-checked the reported memory corruptions against publicly available bug reports and found two interesting matches. In particular, our dangling pointer analysis automatically found a bug in Linux kernel version 3.19, which was previously discovered through manual inspection and classified as a security-relevant vulnerability in Linux in 2014 (i.e., CVE-2014-3153). In fact, this vulnerability gained some popularity due to being used as a tool to allow users privilegede access (aka "root") on their locked-down Android devices, such as the Samsung S5 [91]. The bug was later discovered to be exploited by the HackingTeam company to spy on freedom fighters and dissidents through a malicious kernel extension [202].

Further, our double-free analysis found a driver bug (i.e., CVE-2015-8962) in a disk protocol driver in version 3.19. The vulnerability allows a local user to escalate privileges and corrupt kernel memory affecting a large range of kernel versions including Android devices such as Google's PIXEL [203]. Both vulnerabilities were classified as critical issues with a high severity and could have been easily found through K-Miner's automated analysis. Moreover, we reported two of our use-after-return alerts to the kernel developers.

5.6.2 *Performance*

We now analyze the performance, in particular, the run time, memory consumption, and number of false positives.

Analysis Run Time: As already mentioned, the average analysis time per system call is around 25 minutes. In addition, we analyzed those system calls for which our analyses took longer than 30 minutes. Most system call analyses are dominated by the context handling. However there are some exceptions, notably `sys_execve`, `sys_madvise`, and `sys_keyctl`. The context handling is time consuming, because it represents the first phase of any subsequent data-flow analysis pass. This means, that it conducts multiple inter-procedural pointer analysis, cross-references the global kernel context with the syscall context, and populates the underlying graph data structures for the current system call. This also involves importing and copying information stored on disk, which is slower than accessing RAM. In theory, it should be possible to pre-compute and export the results of the context handling phase for each system call to disk as well. Any data-flow analysis pass could then just re-import the respective file for the current system call, potentially saving some of this overhead (especially in combination with fast SSD hardware). However, we did not implement this optimization feature in the current version of K-Miner.

The UAF checker is notably quicker than the remaining passes, which is due to its reuse of underlying analysis structures from the first pass. In contrast to the use-after-free pass, the double-free analysis has to reconstruct the value-flow graph, which accounts for the majority of its run time. Taken separately, the individual analysis phases require between 5 and 35 minutes run time, which is expected for graph-based analysis, given the magnitude of the input.

Memory Utilization: Initially, our main concern regarded the memory requirements, because of the size of the intermediate representation of the kernel as bitcode image. However, our approach to partition the kernel per system call proved to be effective: on average the analyses utilized between 8.7G and 13.2G of RAM, i.e., around a third of our server's memory, with a maximum of 26G (cf., version 4.10 in Table 5.2). Granted that these numbers also depend to a large extent on the respective kernel version and used configuration, our overall results demonstrate that complex data-flow analysis for OS kernels are feasible and practical. In particular, the memory requirements of K-Miner show that an analysis of future kernel releases is realistic, even with the tendency of newer versions to grow in size.

While the default configuration for the kernel offers a good tradeoff between feature coverage and size, real-world distribution kernels usually have larger configurations, because they enable a majority of features for compatibility reasons. Our current memory utilization is within range of analyzing kernels with such feature models as well. Although we expect to see increased memory requirements (i.e., 128G or more), this does not meet the limit of modern hardware, and K-Miner is able to conduct such analyses without requiring any changes.

Version	Avg. Used	Max Used
3.19	8,765.08M	18,073.60M
4.2	13,232.28M	24,466.78M
4.6	11,769.13M	22,929.92M
4.10	12,868.30M	25,187.82M
4.12	13,437.01M	26,404.82M

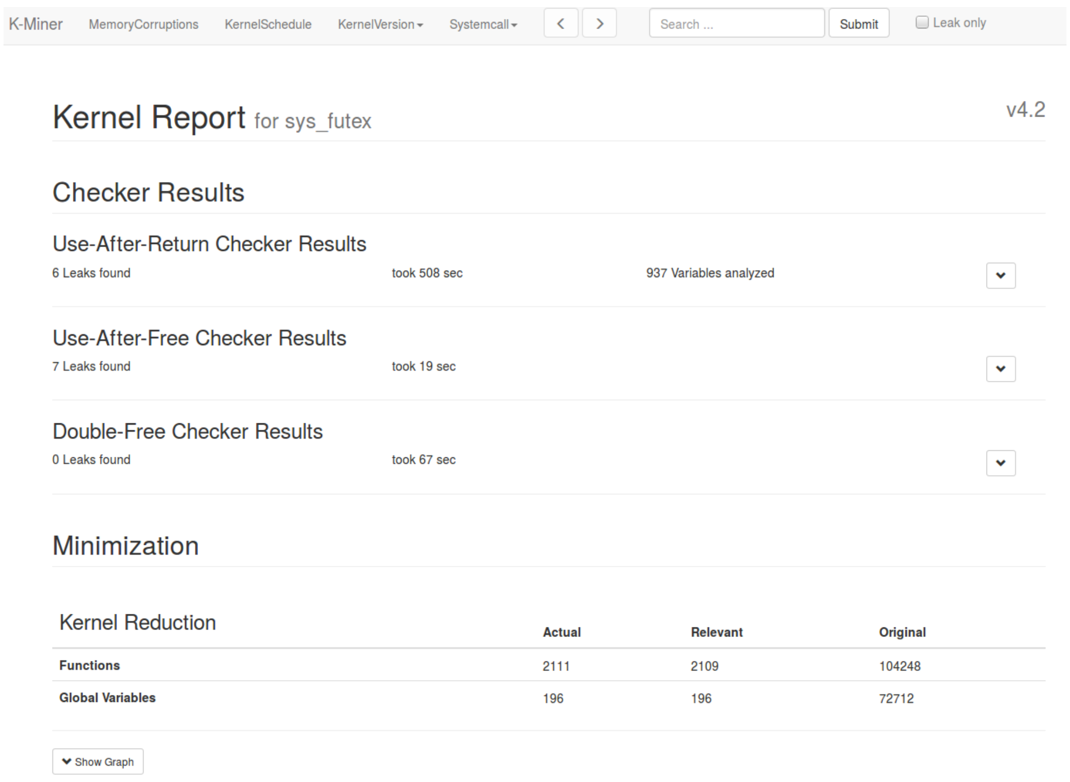
Table 5.2: Average and maximum memory usage of K-Miner

5.6.3 *Usability*

Our framework can be integrated into the standard build process for the Linux kernel with some changes to the main build files, which will then generate the required intermediate representation of the kernel image. Using this bitcode image as main input, K-Miner can be configured through a number of command line arguments, such as number of threads, which checkers to use, and output directory for intermediate results. Results are written to a logfile, which can be inspected manually or subsequently rendered using our web interface to get an overview and check reports for false positives.

Reporting Engine: The web interface for our framework is written in Python. It parses the resulting logfile to construct a JSON-based data model for quick graphing and tabular presentation, and a screenshot is depicted in Figure 5.5 and Figure 5.6. to give an impression of an exemplified workflow. While relatively simple, we found this web-based rendering to be extremely helpful in analyzing individual reports. Developers can already classify and comment upon alerts and reports, and we plan to incorporate the possibility to schedule and manage the launch and configuration of analyses from the web interface in future versions.

False Positives: Similar to other static analysis approaches like the Effect-Based Analyzer (EBA) [188], Coccinelle [184], Smatch [185], or APISAN [187], K-Miner naturally exhibits a number of false positives due to the necessary over-approximations. For instance, the use-after-free analysis still shows a high number of false alarms, and leaves room for improvement. In particular, our investigation showed that there are many cases in the kernel code where a conditional branch based on a nullness check is reported as potential use-after-free. Including these cases in our sanitizer component should be straightforward to further reduce this number. However, there will always be a certain number of false positives for any static analysis tool and developers have to cross-check these alerts, similar to how they have to check for compiler-warnings. Overall K-Miner demonstrates that this scenario is practical through some post-processing and intelligent filtering in our web-interface.



Kernel Report for sys_futex

v4.2

Checker Results

Use-After-Return Checker Results

6 Leaks found | took 508 sec | 937 Variables analyzed

Use-After-Free Checker Results

7 Leaks found | took 19 sec

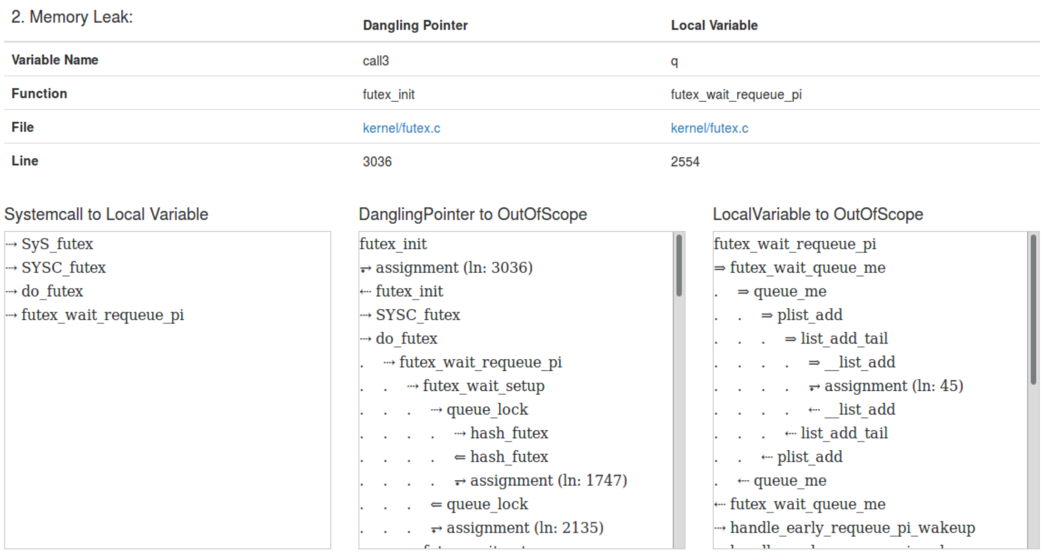
Double-Free Checker Results

0 Leaks found | took 67 sec

Minimization

Kernel Reduction	Actual	Relevant	Original
Functions	2111	2109	104248
Global Variables	196	196	72712

Figure 5.5: Overview of results in the web user interface of K-Miner.



2. Memory Leak:	Dangling Pointer	Local Variable
Variable Name	call3	q
Function	futex_init	futex_wait_requeue_pi
File	kernel/futex.c	kernel/futex.c
Line	3036	2554

Systemcall to Local Variable

```
⇢ SyS_futex
⇢ SYSC_futex
⇢ do_futex
⇢ futex_wait_requeue_pi
```

DanglingPointer to OutOfScope

```
futex_init
⥱ assignment (ln: 3036)
⇠ futex_init
⇢ SYSC_futex
⇢ do_futex
.   ⇢ futex_wait_requeue_pi
.   .   ⇢ futex_wait_setup
.   .   .   ⇢ queue_lock
.   .   .   .   ⇢ hash_futex
.   .   .   .   ⇐ hash_futex
.   .   .   .   ⥱ assignment (ln: 1747)
.   .   .   ⇐ queue_lock
.   .   .   ⥱ assignment (ln: 2135)
```

LocalVariable to OutOfScope

```
futex_wait_requeue_pi
⇒ futex_wait_queue_me
.   ⇒ queue_me
.   .   ⇒ plist_add
.   .   .   ⇒ list_add_tail
.   .   .   .   ⇒ __list_add
.   .   .   .   ⥱ assignment (ln: 45)
.   .   .   .   ⇠ __list_add
.   .   .   ⇠ list_add_tail
.   .   ⇠ plist_add
.   ⇠ queue_me
⇠ futex_wait_queue_me
⇢ handle_early_requeue_pi_wakeup
```

Figure 5.6: Individual bug report in the web user interface of K-Miner.

5.7 POSSIBLE EXTENSIONS

While K-Miner currently does not offer a proof of soundness, we sketched an informal reasoning of why the kernel-code partitioning along the system call API is a sensible strategy in Section 5.6. There are additional challenges for a formal result: first, in some cases the kernel uses non-standard code constructs and custom compiler extensions, which may not be covered by LLVM. However, for these constructs the LLVM Linux project maintains a list of patches, which have to be applied to the kernel to make it compatible to the LLVM compiler suite. Second, some pointer variables are still handled via `unsigned long` instead of the correct type. These low-level "hacks" are difficult to handle statically, because they exploit knowledge of the address space organization or underlying architecture specifics. Nonetheless, such cases can be handled in principle by embedding the required information in LLVM or by annotating these special cases in the source. Finally, our memory tracking component currently relies on a list of allocation functions. For cases like file descriptors or sockets the respective kernel objects are pre-allocated from globally managed lists and individual objects are retrieved and identified by referring to their ID (usually an integer number). This can be resolved by considering all objects from the same list to be modeled as objects of the same type, and marking functions for retrieval as allocations.

As K-Miner is designed to be a customizable and extensible framework, implementing additional checkers is straightforward. To this end, we already implemented additional double-lock and memory-leak checkers, thereby covering additional bug classes. Up to this point we only verified that these additional pass implementations are able to detect intra-procedural bugs.[4] However, as our other analysis passes in K-Miner, the double-lock implementation covers inter-procedural double-lock errors in principle, including bugs spanning multiple source files. Due to lack of time we did not conduct a complete analysis of all five kernel versions, which is why these results are not yet included in Table 5.1. Similarly, implementing analyses to find buffer overflows, integer overflows, or uninitialized data usage remains as part of our future work to cover all potential sources of memory corruption as mentioned in Section 5.2.

Moreover, we note that it is possible to add analyses passes, e.g., to check for buffer overflows, integer overflows, or uninitialized data usage, and we are actively pursuing these for the further development of K-Miner. Implementing additional analyses for the remaining bug classes as mentioned in Section 5.2 is also part of our future work to cover all potential sources of memory corruption.

While primarily analyzing the system call API, we found that analyzing the module API in a similar way should be possible and provide interesting results, since many bugs result from (especially out-of-tree) driver and module code. Although this interface is not as strict as the highly standardized system call API, the main top-level functions of many drivers are exported as symbols to the entire kernel image, while internal and helper functions are marked as `static`. Hence, we should be able to automatically detect the main entry points for most major driver modules by looking at its exported symbols and building a call graph that starts

4 In particular, the lock errors introduced in commits `09dc3cf` [204], `e50fb58` [205], `0adb237` [206], and `16da4b1` [207] of Linus' tree.

with the exported functions. We can then analyze this automatically constructed control flow of the drivers by applying the data-flow analysis passes to the resulting code partitions. In addition to our current approach, this would allow for an extension of our adversary model to include malicious devices and network protocols. We included a prototypical draft of this functionality to analyze module code using K-Miner in the future.

6 RELATED WORK

In this chapter, we give an overview of the related work regarding data-only attacks and defenses, and how they relate to our data-only attack on dynamically generated code [1]. We then briefly compare existing data-only defenses for the OS context to our randomization-based defense PT-Rand [2]. Lastly, we discuss the role of our novel data-flow analysis framework K-Miner [3] with respect to the existing tools and approaches for static analysis of kernel and user-level code.

6.1 DATA-ONLY ATTACKS AND DEFENSES FOR STATIC CODE

One of the earliest instances of a data-only attack was presented in the context of server applications [13], where authentication checks could be bypassed resulting in a privilege escalation without requiring any modification of the control flow of the application. With FLOWSTITCH [208] data-only attacks where demonstrated to be automatable in a similar fashion to *auto-ROPers* [209, 210]. However, in contrast to finding and compiling gadget chains for a code-reuse attack this is more challenging for data-only exploits, since it requires careful analysis of the data flows leading from the starting point to a given vulnerability, and from that vulnerability to potentially sensitive information or relevant data structures, which are are highly application-dependent. Under certain assumptions this can even enable Turing-complete malicious computation, called Data-Oriented Programming (DOP) [68], for a given application. A prominent example is changing the access permissions to a memory page by manipulating the input parameters of the function that dynamically loads libraries (i.e., `dlopen()`).

A number of randomization-based schemes have been proposed to tackle the challenges raised by data-only attacks. For instance, by partitioning all data accesses of a program based on extensive static analysis, data objects of an application can be isolated into different memory compartments [211, 212]. Through instrumentation of all data accesses in the program, this isolation can be enforced at run time, e.g., through use of an `xor` operation in combination with a per-compartment key, preventing an adversary from leveraging a memory-corruption vulnerability in one compartment to access or modify data in another compartment. Unfortunately, this incurs overheads of around 30% on average.

Another approach is to constantly re-randomize data objects in time intervals [171]. However, this again leaves a time window during which attacks are possible. While this can potentially be avoided for applications that handle data with a strict lifecycle, e.g., based on events or data being handled over the network [213], this does not represent a general approach.

Many other schemes aim towards leakage resilience against data-only attacks by keeping the randomization secret outside of memory at all times during operation, e.g., by storing it in a register. In this way TRESOR [166] demonstrated memoryless AES encryption on x86. Additionally, some randomization approaches keep the random base offset into a sensitive memory region hidden by storing it in a register. In combination with a code-reuse defense, this enables strong leakage resilience guarantees against information-disclosure attacks based on memory corruption [46, 53, 214].

6.2 DATA-ONLY ATTACKS AND DEFENSES FOR JIT CODE

In contrast to static code, which is normally mapped as readable and executable pages, attacks on dynamically generated (i.e., just-in-time compiled) code usually leverage the circumstance that memory pages for JIT code have to be mapped writable. For this reason, many JIT-defenses proposed various techniques to implement a W⊕X mapping for just-in-time compiled code [55, 145, 215, 216], to prevent code injection. Yet, even if a non-writable mapping is enforced during JIT code execution, the compiler still has to write newly generated code to memory at some point. Hence, a (possibly small) time window remains. From a security standpoint this time window is generally problematic, since an adversary could be able to inject malicious code even during a relatively short period, and hence, code-injection attacks are still possible in principle. In this way, JIT exploits are also often able to construct sophisticated exploits without requiring any code-reuse techniques, as demonstrated in a Chrome browser exploit at pwn2own in 2016 [143].

This is why Song et al. [144] proposed to de-couple JIT code generation and JIT code execution into separate processes and sharing memory pages for JIT code. The compiler process then maps JIT code pages as writable, whereas the JIT process can map the same code pages as readable and executable, but not writable. This design avoids the drawback of a time window completely, successfully preventing JIT-based code-injection attacks. Although the required communication and synchronization overhead between the two processes can severely impact the run-time performance (i.e., up to 50%), a similar design was subsequently implemented by Microsoft for the JavaScript engine of the Edge browser [217]. However, Microsoft recently announced that future Edge versions will be using Google's V8 instead of Chakra [218].

In contrast to code injection, code-reuse attacks do not write malicious code. Instead, these attacks string existing code snippets together to achieve arbitrary behavior [10]. As such, both static and dynamic code are prone to code-reuse attacks despite W⊕X being deployed and active. While JIT code generation is usually limited to a small subset of (benign) machine instructions, several techniques have been proposed to trick the JIT compiler into generating arbitrary instructions. For instance, Blazakis [147] demonstrated that numeric constants can be inserted into the generated, benign instructions as parameters. By utilizing unaligned code fetches, these parameters can subsequently be interpreted as code, leading to the possibility of injecting and executing a small number of attacker-chosen bytes into the JIT code.

For this reason, JIT compilers employ *constant blinding*, a technique through which numerical constants are obfuscated using a secret value that is generated at compile time. This requires that the masked constant is de-obfuscated at run time, incurring additional run-time overheads. For this reason constant blinding is only applied to large constants in practice, leading to potential attacks [146].

Further, the underlying technique of hiding unaligned code in instruction parameters was recently generalized, e.g., demonstrating that the offset parameters of relative branch instructions can be exploited to trick the JIT compiler into emitting potentially malicious, unaligned instructions [149].

Another line of research proposes to mitigate JIT-based code reuse by applying fine-grained randomization [219]. However, all fine-grained randomization approaches are prone to attacks based on information disclosure [17]. This is why randomization approaches require leakage resilience, for example, by providing an execute-only mapping of the generated code [54, 55]. However, this is not supported by current architectures, and hence, it is currently unclear how this should be implemented efficiently for real-world JIT engines. Other code-reuse defenses such as *destructive code reads* [220, 221] disable information disclosure by intercepting memory reads from the code section and overwriting the results with random data. While this disables code-reuse attacks there are two problems with this approach: first, it assumes that benign software never reads its own code section. Second, in the context of dynamically generated code the possibility of blind code-reuse attacks remains and was successfully demonstrated [222] by forcing required gadgets to be generated multiple times.

Moreover, Software-Fault Isolation was proposed for JIT code as well, by preventing the dynamically generated code from modifying non-JIT memory through a sandbox [223]. However, like all other SFI-based techniques this incurs a large overhead.

While CFI was traditionally designed for static code, it has been ported to dynamically generated code with an average overhead of around 15% [148]. However, as we demonstrate with DOJITA [1] this can be generically bypassed by resorting to data-only attacks and modifying the intermediate representation of the JIT compiler component to generate arbitrary malicious instructions despite CFI being deployed and active.

6.3 KERNEL AND PAGE TABLE PROTECTION

In addition to our data-only attack against the page tables, several page-table based attacks have been presented, e.g., by industry researchers [224]. As a consequence, an adversary with access to a memory-corruption vulnerabilities in the kernel is able to modify the page tables and deactivate any memory protection [225]. These attacks are enabled by the fact that the location of the page tables in kernel memory is writable and can easily by determined.

Several defenses have been proposed by the related work to protect the page table against malicious modification [45, 173, 226, 228, 229], and we provide a quick overview of these schemes in Table 6.1.

All proposed defenses so far are designed as a *run-time monitor*, i.e., they include and check every page-table related operation against a pre-defined set of security policies. In contrast to this, with PT-Rand [2] we are the first to propose a randomization-based approach to protect the page tables in the OS.

Both SecVisor [228] and HyperSafe [229] are based on a Virtual-Machine Monitor (VMM) and require a hardware trust anchor in the form of virtualization extensions. SecVisor deploys a thin hypervisor to ensure integrity of kernel code.

	SPROBES [226]	TZ-RKP [227]	SKEE [173]	KCoFI [45]	SecVisor [228]	HyperSafe [229]
OS support	✓	✓	✓	✓	✓	✗
VMM-based	✗	✗	✗	✓	✓	✓
TEE-based	✓	✓	✗	✗	✗	✗
RT-checks	✓	✓	✓	✗	✓	✓
Overhead	*unspecified*	7.56%	15%	> 100%	14.58%	5%

Table 6.1: Comparison of defense approaches ensuring the integrity of page tables. In contrast to the related work, PT-Rand [2] does not require hardware trust anchors such as Virtual-Machine Monitors (VMMs) or Trusted-Execution Environments (TEEs). Since our scheme additionally does not require costly policy checks at run time, but is based on leakage-resilient randomization instead it also incurs a low overhead of only 0.22% on average.

This is achieved by utilizing hardware-aided memory virtualization and IOMMU to enforce memory protection independently from the kernel. While this prevents an adversary with access to kernel memory from tampering with the hypervisor page tables, it requires policy-based checks to prevent confused deputy attacks against the VMM. Similarly, HyperSafe [229] prevents control-flow hijacking attacks against the hypervisor by means of control-flow integrity enforcement, however, it is not designed to protect OS memory. Moreover, an OS-based attacker could still compromise the hypervisor page tables from a virtualized guest system in that scenario [230]. Despite leveraging hardware-supported virtualization extensions, these defenses still visibly affect the run-time performance of the overall system with 14.58% for SecVisor and 5% for HyperSafe.

Another hypervisor-based defense is KCoFI [45], which enforces control-flow integrity for the entire operating system kernel. It leverages the Secure-Virtual Architecture (SVA) built on top of LLVM's intermediate representation to protect the page tables and the CFI policies in a memory region that is in-accessible to the OS kernel. Since this virtualization approach is enforced in software by the SVA, KCoFI imposes a substantial overhead of more than 100%.

Instead of deploying a hypervisor, SPROBES and TZ-RKP both utilize trusted execution environments [226, 227]. To this end, they deploy run-time checks for enforcing a policy on the memory management functionality provided by the OS, which run inside the TEE. Since isolation between the OS and the TEE is enforced by hardware, this prevents an adversary from compromising secure memory using a kernel vulnerability. While the authors of SPROBES [226] unfortunately do not provide any benchmark numbers or other run time overhead measurements, the performance impact of TZ-RKP [227] is reported as 7.56%.

In contrast to those TEE-based schemes, SKEE [173] implements a similar policy enforcement based on the fact that the ARM architecture provides multiple virtual address spaces at run time. By utilizing this micro-architectural feature, SKEE isolates the run-time checks from the OS. Since the platform now has to maintain two separate address spaces, and an additional run-time environment, it incurs an overhead of 15%.

It is noteworthy to mention that all policy-based approaches mark the page tables read-only to protect them against unauthorized access. This means that the respective pages have to be remapped as writable for a short period during execution, enabling timing attacks through parallel accesses by an adversary in principle.

While Microsoft released a patch to enable kernel-space randomization in the context of page tables for Windows 10 [231], they did not implement any leakage resilience features, and hence, the location of the randomized page table can still be leaked by means of information-disclosure attacks. With our work PT-Rand [2] we show that randomization-based defenses in the kernel require leakage resilience to protect against such attacks and demonstrate how this can be implemented efficiently through register-based information hiding techniques.

6.4 STATIC ANALYSIS FRAMEWORKS

In this section we give a brief overview of the related work and compare K-Miner to existing frameworks and tools. In contrast to dynamic run-time approaches, such as KASAN [232], TypeSan [233], Credal [234], UBSAN [235], and various random testing techniques [177–179], our approach aims at static analysis of kernel code, i.e., operating solely during compile time. As there already exists a large body of literature around static program analysis [181, 236], we focus on static analysis tools targeting operating system kernels, and data-flow analysis frameworks for user space that influenced the design of K-Miner.

It is important to note that applying static analysis frameworks designed for user space programs is not possible a priori in the kernel setting: data-flow analysis passes expect a top-level function, and an initial program state from which analysis passes can start to propagate value flows. These requirements are naturally satisfied by user space programs by providing a `main` function, and a complete set of defined global variables. However, operating systems are driven by events, such as timer interrupts, exceptions, faults, and traps. Additonally, user space programs can influence kernel execution, e.g., by issuing system calls. Hence, there is no single entry point for data-flow analysis for an operating system. With K-Miner we present the first data-flow analysis framework that is specifically tailored towards this kernel setting.

6.4.1 *Kernel Static Analysis Frameworks*

The Effect-Based Analyzer (EBA) [188] uses a model-checking related, inter-procedural analysis technique to find a pre-compiled list of bug patterns. In particular, it provides a specification language for formulating and finding such patterns. EBA provides lightweight, flow-insensitive analyses, with a focus towards double-lock bugs. Additionally, EBA restricts analysis to individual source files. K-Miner provides an expressive pass infrastucture for implementing many different checkers, and is specifically tailored towards the execution model of the kernel allowing complex, context and flow-sensitive data-flow analyses, potentially spanning the entirety of the kernel image.

Coccinelle [184] is an established static analysis tool that is used on a regular basis to analyze and transform series of patches for the kernel. While originally not intended for security analysis, it can be used to conduct text-based pattern matching without the requirement for semantic knowledge or abstract interpretation of the code, resulting in highly efficient and scalable analyses. In comparison to our framework, Coccinelle is not able to conduct any data-flow, or inter-procedural analysis.

The Source-code Matcher (Smatch) [185] is a tool based on Sparse [237], a parser framework for the C language developed exclusively for the Linux kernel. Smatch enriches the Sparse syntax tree with selected semantic information about underlying types and control structures, enabling (limited) data-flow analyses. Like Coccinelle, Smatch is fast, but constrained to intra-procedural checks per source file.

APISAN [187] analyzes function usage patterns in kernel code based on symbolic execution. In contrast to other static analysis approaches, APISAN aims at finding semantic bugs, i.e., program errors resulting from incorrect usage of existing APIs. Because specifying the correct usage patterns manually is not feasible for large code bases, rules are inferred probabilistically, based on the existing usage patterns present in the code (the idea being that correct usage patterns should occur more frequently than incorrect usage patterns). In comparison to K-Miner, APISAN builds on LLVM as well, but only considers the call graph of the kernel and is not able to conduct any inter-procedural data-flow analyses.

TypeChef [186] is an analysis framework targeting large C programs, such as the Linux kernel. In contrast to our work, TypeChef focuses on variability-induced issues and analyzing all possible feature configurations in combination. For this, it provides a variability-aware pre-processor, which extracts the resulting feature model for the kernel, e.g., by treating macros like regular C functions. TypeChef does not conduct any data-flow analysis on their resulting variability-aware syntax tree.

6.4.2 *User Space Static Analysis*

The Clang Static Analyzer [196] consists of a series of checkers that are implemented within the C frontend Clang of the LLVM compiler suite. These checkers are invoked via command-line arguments during program compilation and can easily be extended. As part of the Linux LLVM project [198] there was an effort to implement kernel-specific checkers. However, to the best of our knowledge, this effort has since been abandoned.

The Static Value-Flow (SVF) [197] analysis famework enhances the built-in analysis capabilities of LLVM with an extended pointer analysis and a sparse value-flow graph representation. K-Miner builds on top of LLVM and leverages the pointer analyses provided by SVF to systematically analyze kernel APIs, such as the system call interface.

Mélange [238] is a recent data-flow analysis framework for user space, that is able to conduct complex analyses to find security-sensitive vulnerabilities, such as unitialized reads. Mélange is able to analyze large C and C++ user space code bases such as Chromium, Firefox, and MySQL.

Astrée [180] is a proprietary framework for formal verification of C user programs for embedded systems through elaborate static analysis techniques. It operates on *synchronous programs*, i.e., analyzed code is not allowed to dynamically allocate memory, contain backward branches, union types, or other conflicting side effects. Astrée is able to provably verify the absence of any run-time errors in a program obeying these restrictions and was used to formally verify the primary flight control software of commercial passenger aircraft.

Soot [239] is a popular and widely used static analysis framework capable of conducting extensible and complex data-flow analyses. However, Soot is targeted towards Java programs, and hence, cannot analyze programs written in C or C++.

Part III

REMOTE HARDWARE EXPLOITS: AN EMERGING ATTACK PARADIGM

based on Hammer Time [4], CAn't Touch This [5], LAZARUS [6], and HardFails [7].

7 HAMMER TIME: REMOTE ATTACKS ON DRAM AND INITIAL DEFENSES.

We now turn towards remote hardware exploits, which represent an emerging attack vector besides the traditional memory-corruption-based approaches. While hardware-based exploits typically rely on platform-specific implementation details they are also more powerful and do not require vulnerable software. In this chapter, we first introduce Rowhammer which represents a recent class of hardware-based attacks that are possible from software due to vulnerable DRAM modules, which are used by the majority of computer systems deployed today. We first present our remote Denial-of-Service attack [4] and briefly discuss several attacks that were presented in the literature. We then discuss possible defense approaches such as blacklisting [5], before introducing our general software-only defense in the next Chapter.

7.1 DRAM AND THE ROWHAMMER BUG

Dynamic-Random Access Memory (DRAM) modules, as shown in Figure 7.1, are structured hierarchically. These individual hardware modules usually come in the form of Dual Inline Memory Modules (DIMM), which are attached physically to the mainboard of the platform and directly connected to the memory controller of the platform through a *channel*. Modern desktop systems usually feature two channels facilitating parallel accesses to memory. The DIMM can be divided into one or two ranks corresponding to its front- and backside. Each rank contains multiple chips which are comprised of one or multiple banks that contain the memory cells. Each bank is organized in columns and rows, as shown in Figure 7.2. DRAM rows contain neighboring memory cells, whereas columns hold adjacent cells.

An individual memory cell consists of a capacitor and a transistor. To store a bit in a memory cell, the capacitor is electrically charged. By reading a bit from a memory cell, the cell is discharged, i.e., read operations are destructive. To prevent information loss, read operations also trigger a process that writes the bit back to the cell. A read operation always reads out the bits from a whole row, and the result is first saved in the row buffer before it is then transferred to the memory controller. This *row buffer* is also used to write back the content into the row of memory cells to restore their content.

While DRAM hardware is highly standardized, cheap, widely used, and commercially available as off-the-shelf components, it also exhibits a number of reliability problems: adverse conditions such as higher temperature or electromagnetic radiation can lead to bit errors in the information stored digitally on the hardware [240–242]. Moreover, researchers demonstrated that DRAM hardware is in fact subject to bit errors that are reproducable purely from software *under completely benign operating conditions*. In particular, several independent studies found

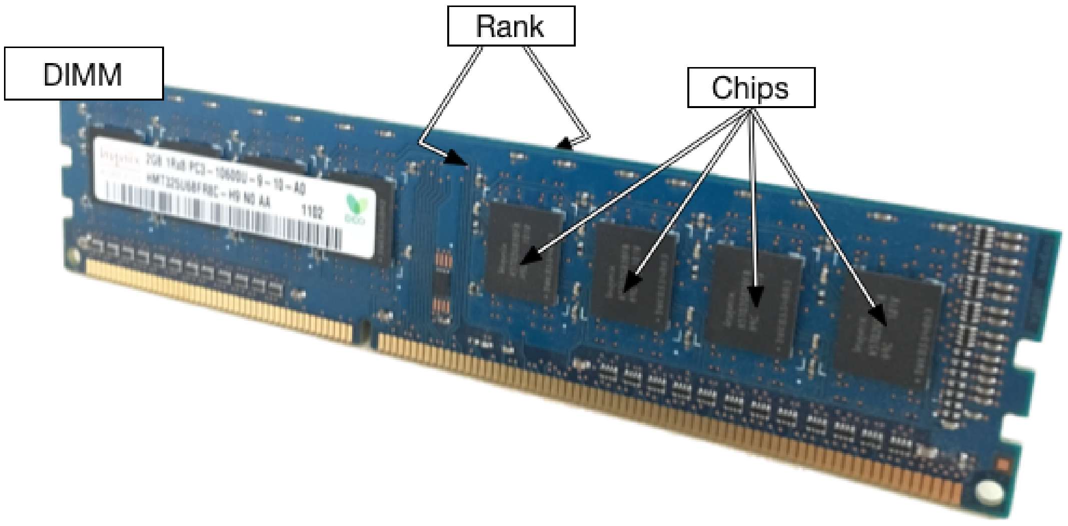


Figure 7.1: Organization of a DRAM module.

that frequently accessing physically co-located memory cells leads to bit flips in adjacent memory cells [63, 71]. This effect, called *Rowhammer*, was subsequently shown to be exploitable by remote adversaries to undermine deployed security mechanisms on computing platforms using DRAM hardware [72–74].

The Rowhammer bug occurs during the write-back operation when accessing co-located rows of the same bank: frequently triggering (or *hammering*) an aggressor row (**A**) that is physically located on top of the victim row (**V**) affects the cells of the victim row due to electromagnetic coupling—although these cells are never accessed directly. Amplified by the shrinking feature size of the manifactured DRAM chips, several studies found Rowhammer to be a wide-spread reliability issue, and conclude that Rowhammer seems to be an inherent design problem of DRAM chips as all vendors and even ECC DRAM modules are affected [63, 71].

It is noteworthy to mention that there exists a mapping between physical memory addresses and the respective DRAM structures (i.e., rank, bank, and row index) on the hardware module that is usually defined by the manufacturer and implemented within the memory controller of the platform. Consequently, two consecutive physical memory addresses can be mapped to memory cells that are located on different ranks, banks, or rows. For example, on Intel Ivy Bridge CPUs the 20th bit of the physical address determines the rank. As such, the consecutive physical addresses 0x2FFFFF and 0x300000 can be located on front and back side of the DIMM for this architecture. The knowledge of the physical memory location on the DIMM is important for both Rowhammer attacks and defenses, since bit flips can only occur within the same bank. For Intel processors, the physical-to-DRAM mapping is not documented, but has been reverse engineered [76, 243].

7.2 ROWHAMMER IN THE CONTEXT OF SECURITY-SENSITIVE APPLICATIONS

Since DRAM is cheap and widely deployed its use has been considered in the context of security-sensitive applications. One of those applications are Physically Unclonable Functions (PUFs), which represent a research technology that promises

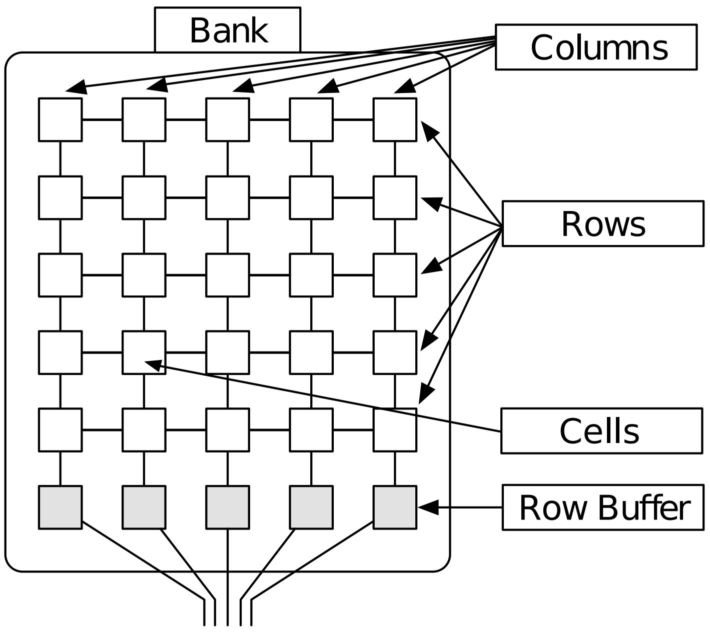


Figure 7.2: DRAM chips are structured hierarchically, with *banks* forming one of the intermediate components that can be replicated for individual chips. This diagram shows the organization inside a bank, which is only accessed via its *row buffer* from the outside.

to establish trust anchors in embedded systems with minimal hardware requirements. They allow for utilizing inherent manufacturing process variations to extract unique identifiers or secrets in a reproducable way. Since the generated bits directly depend on the physical properties of the underlying technology (i.e., DRAM) it is assumed to be unique. However, the uniqueness property is not sufficient to ensure security: the response to a particular challenge must also be unpredictable, i.e., it is required that the output looks like a random sequence to a software attacker. PUFs have been proposed as tamper-evident building blocks for various use cases such as for authentication, key-derivation, and device identification [244–246].

So far many different implementations in hardware have been proposed and evaluated over the recent years [246–248] and memory-based PUFs leverage physical properties of memory cells in SRAM or DRAM technology, which is widely deployed in practice. The decreasing prices for DRAM hardware aggrevate those research efforts [249–252].

We demonstrate a Rowhammer-based attack that effectively denies meaningful operation of a DRAM PUF [4] under the same software adversary model as in Part II. However, in contrast to the previously presented data-only attacks, we *do not* require a memory-corruption vulnerability in the any part of the system software. Instead, we exploit the Rowhammer hardware bug that is found in many DRAM modules that are sold on the market today. We stress that our attack works entirely remotely and does not require physical presence of the attacker. Under this model, we first revisit the security of security-sensitive applications, such as PUFs. We implemented and extensively evaluated our attack in a real-world setup for different configurations. Our results confirm that DRAM hardware is not a suitable building block for security-sensitive applications.

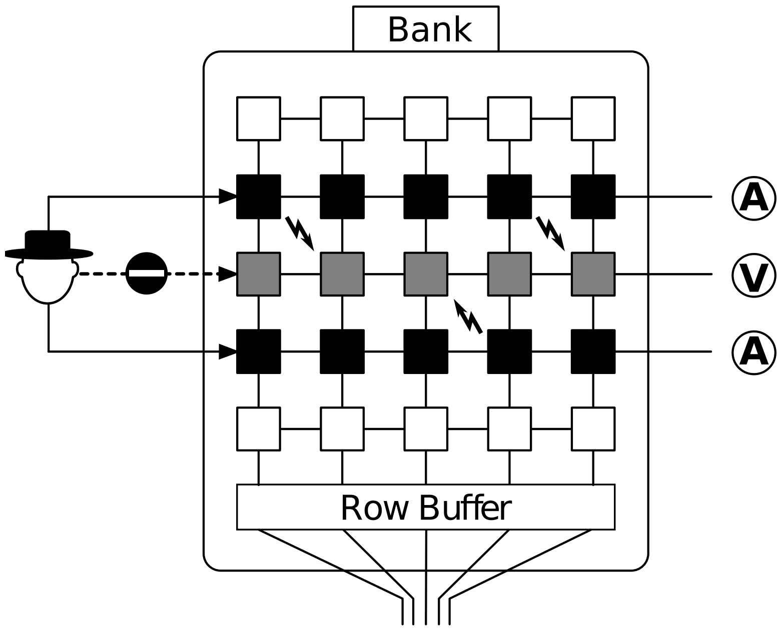


Figure 7.3: What makes Rowhammer-based attacks [4, 72–76, 134, 135] so severe is the fact that only *adjacent* memory rows (i.e., above and below the attacked memory region) are accessed by the attacker. Due to the physical-to-DRAM mapping, these physically neighbouring rows can reside in completely separate parts of the address space. However, through electro-magnetic coupling effects an adversary can influence these otherwise inaccessible memory cells which can even result in flipped bits in the attacked row.

Usually, PUF responses would be used to derive id tokens or secret key material in a reproducable way. However, as depicted in Figure 7.3 Rowhammer-based attacks aim at maliciously modifying the sensitive memory region (**V**). Crucially, Rowhammer attacks such as our DoS attack [4] operate without ever accessing the sensitive memory region itself, i.e., only using *adjacent* memory locations (**A**) which the attacker is allowed to access. This means that the attacker never violates any existing access-control mechanisms deployed to protect the sensitive memory region against malicious software accesses. Due to the physical-to-DRAM mapping an adversary can exploit physically neighbouring memory cells, and hence, Rowhammer represents a hardware vulnerability with serious security consequences for real-world systems. This has also been demonstrated previously to be exploitable in practice [72–74]: the physical-to-DRAM mapping between physical addresses on the system and the physical location this corresponds to on the DRAM module, the attacker can trick the memory allocator into co-locating its memory with the sensitive memory region in question. In the context of PUFs this results in a faulty id or key, since those are directly based on the hammered memory cells. Consequently, it cannot be used and benign operation is blocked.

7.3 ROWHAMMER AS AN EXPLOIT PRIMITIVE

Recently, various Rowhammer-based attacks have been presented [72–76, 134, 135]. Specifically, Rowhammer was utilized to undermine isolation of operating system and hypervisor code, and escape from application sandboxes leveraged in web browsers. In the following, we describe the challenges and workflow of Rowhammer attacks. A more elaborated discussion on real-world, Rowhammer-based exploits will be provided in Chapter 11.

The Rowhammer fault allows an attacker to influence the electrical charge of individual memory cells by activating neighboring memory cells. Kim et al. [63] demonstrate that repeatedly activating two rows separated by only one row, called *aggressor rows*, lead to a bit flip in the enclosed row, called *victim row*. To do so, the attacker has to overcome the following challenges: (i) undermine memory caches to directly perform repetitive reads on physical DRAM memory, and (ii) gain access to memory co-located to data critical to memory isolation.

Overcoming challenge (i) is complicated because modern CPUs feature different levels of memory caches which mediate read and write access to physical memory. Caches are important as processors are orders of magnitude faster than current DRAM hardware, turning memory accesses into a bottleneck for applications [132]. Usually, caches are transparent to software, but many systems feature special instructions, e.g., `clflush` or `movnti` for x86 [72, 73], to undermine the cache. Further, caches can be undermined by using certain read-access patterns that force the cache to reload data from physical memory [253]. Such patterns exist, because CPU caches are much smaller than physical memory, and system engineers have to adopt an eviction strategy to effectively utilize caches. Through alternating accesses to addresses which reside in the same cache line, the attacker can force the memory contents to be fetched from physical memory.

The attacker's second challenge (ii) is to achieve the physical memory constellation that results in DRAM co-location. For this, the attacker needs access to the aggressor rows in order to activate (*hammer*) them. In addition, the victim row must contain data which is vulnerable to a bit flip. Both conditions cannot usually be enforced by the attacker a priori. However, this memory constellation can be achieved with high probability using the following approaches: first, the attacker allocates memory hoping that the aggressor rows are contained in the allocated memory. If the operating system maps the attacker's allocated memory to the physical memory containing the aggressor rows, the attacker has satisfied the first condition. Since the attacker has no influence on the mapping between virtual memory and physical memory, she cannot directly influence this step, but she can increase her probability by repeatedly allocating large amounts of memory. Once control over the aggressor rows is achieved, the attacker releases all allocated memory except the parts which contain the aggressor rows. Next, victim data that should be manipulated has to be placed on the victim row. Again, the attacker cannot influence which data is stored in the physical memory and needs to resort to a probabilistic approach. The attacker induces the creation of many copies of the victim data with the goal that one copy of the victim data will be placed in the victim row. The attacker cannot directly verify whether the second step was successful, but can simply execute the Rowhammer attack to validate whether the attack was successful. If not, the second step is repeated until the Rowhammer successfully executes. Seaborn et al. [72] successfully implemented this approach to compromise the kernel from an unprivileged user process. They gain control over the aggressor rows and then let the OS create huge amounts of page table entries with the goal of placing one page table entry in the victim row. By flipping a bit in a page table entry, they gained control over a subtree of the page tables allowing them to manipulate memory access control policies.

7.4 ON THE DISTRIBUTION OF BIT FLIPS

While working on our attack we discovered that bit flips do not always stabilize over time and that relatively stable bit-flip locations do not necessarily adhere to a random distribution. More specifically, certain bit flips cannot be used by an attacker since they only appear as *transient* bit flips that vanish after a short period of time. We conducted our tests using the Rowhammer testing tool developed and published by Google [254]. However, there are two factors that impact the comprehensiveness of identifying vulnerable bits: spatial coverage and temporal coverage.

The memory coverage of the Rowhammer testing tool is limited to the physical memory it can access. Since the Rowhammer testing tool executes as an ordinary process on the system, it has no influence on the physical memory assigned when allocating memory. In particular, there are parts of physical memory which will never be assigned to the Rowhammer test tool, as some memory is reserved by other systems entities, e.g., memory reserved by the kernel. Hence, the Rowhammer testing tool can never test these regions for vulnerable bits.

Additionally, the Rowhammer testing tool has to share the system's memory with other processes. For instance, the Rowhammer testing tool cannot allocate all memory since other processes (and other dynamic components like loadable kernel modules – LKM) also require memory. However, since these memory allocations are dynamic they can easily change among reboots and runs. This means that the temporal coverage of the tool for each individual run is limited. However, the *cumulative* coverage achieves coverage of the entire process' allocatable memory. Lastly, even if the Rowhammer testing tool cannot detect all vulnerable memory bits, it still can detect all bits that are exploitable from the attacker's point of view, i.e., the attacker leverages the same techniques as the Rowhammer testing tool to detect vulnerable bits.

Additionally, we found that stable bit flips tended to cluster around particular regions for a given DRAM module. For instance, in our tests successive bit flips were highly correlated with respect to their physical location on the DRAM module. For this reason, bit flips would be much more likely around areas that experienced bit flips previously. By collecting and observing a number of bit-flip locations for various machines and DRAM modules, we were able to formulate a fault-model of the affected DRAM modules (e.g., to break the unclonability property of the DRAM PUF [4]) and even predict possible locations of future bit flips with a high accuracy.

Although not representative or statistically significant (we only evaluated five different systems [4, 5]) our findings motivated us to design an initial defense that operates by blacklisting vulnerable memory pages that contain stable bit-flip locations. We will briefly describe our initial defense in the next section.

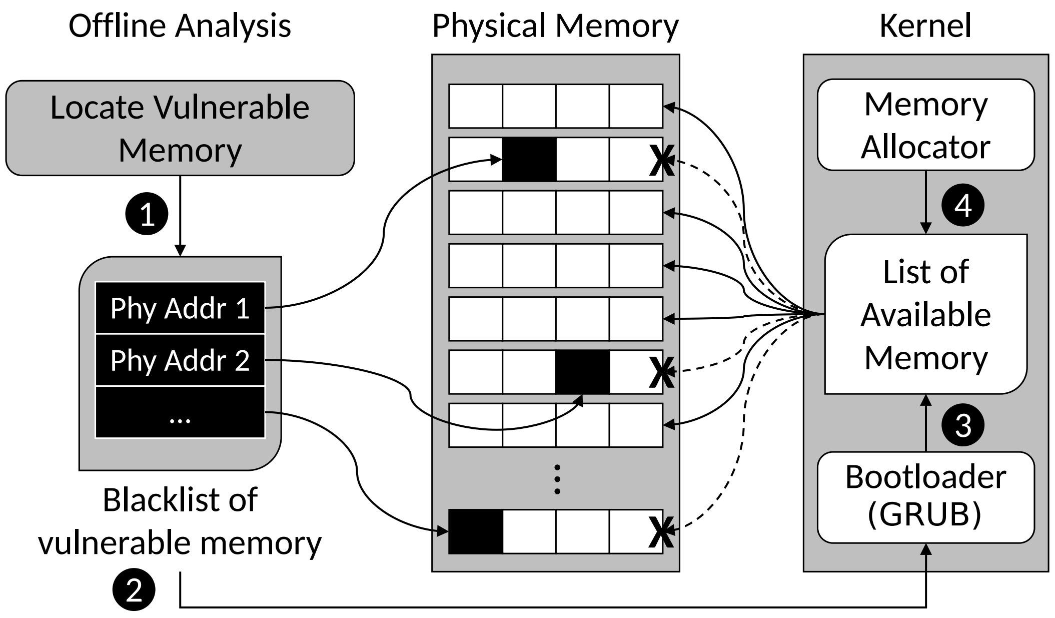


Figure 7.4: Workflow of ouf bootloader extensions

7.5 INITIAL ROWHAMMER DEFENSE

Prior to launching a Rowhammer attack, the attacker needs to probe the memory to locate vulnerable memory. Hence, a straightforward software-only defense is to identify the vulnerable memory parts and mark them as unavailable. To validate this, we developed a bootloader extension that locates and disables vulnerable memory parts. While we investigated different implementation strategies, we opted for a bootloader extension in order to protect a wide range of operating systems. That is, our bootloader-based approach does not require any changes to the operating system and is fully compatible with legacy systems.

In general, all bootloaders commit a list of available memory to the operating system. This list is dynamically obtained during boot time using the system's firmware (e.g., BIOS/UEFI). It is common that certain physical memory addresses cannot be used by the operating system, e.g., the memory reserved for hardware devices such as graphics or network cards or memory that is already occupied by the firmware. Our bootloader extension hooks into this workflow. Specifically, it adds the physical addresses of vulnerable memory to the list of unavailable memory. To locate vulnerable addresses, we re-use techniques from existing Rowhammer exploitation tools [75, 254] and slightly adjust them for the purpose of our approach. This analysis is performed offline prior to activating the blacklisting approach. From the operating system's perspective, the vulnerable physical memory is simply never used.

We exploit the fact that this blacklisting of certain physical addresses is already supported by the operating system. Specifically, x86 compatible operating systems detect usable memory through the so-called *e820 map* [255]. This map contains a list of usable physical memory. It can be obtained from the BIOS through the interrupt number `0xe820`. On modern systems, this map is not requested at run time, but forwarded directly at boot-time instead. To prevent Rowhammer attacks, we instrument the GRUB [256] bootloader to add physical addresses of vulnerable memory to the e820 map. Although the original e820 specification only supports 128 entries, the EFI Platform Initialization specification – which is also supported by GRUB – does not limit the number of entries [257]. In fact, GRUB first allocates a buffer to store both the original e820 map and additional entries. Once it has stored the original e820 entries into this buffer, it passes a pointer to this buffer to the operating system. Our patch increases the size of the allocated buffer to additionally fit the entries for the blacklisted pages.

Figure 7.4 depicts the detailed workflow of our bootloader-based defense approach. After installing our patch the vulnerable memory rows are identified with the help of publicly available tools during an offline analysis [72, 75, 254]. These programs scan the memory of the machine for vulnerable pages by allocating large amounts of memory and trying to produce bit flips. Next, these pages are included as an environment variable in the boot configuration file of GRUB. When the BIOS starts the bootloader will request the environment variable and extend the e820 map using the specified entries. The OS will not use the vulnerable memory thereby preventing the attacker from leveraging Rowhammer-based attacks using the identified pages.

8 MITIGATING ROWHAMMER ATTACKS AGAINST OS KERNELS.

In the last chapter, we have seen that the Rowhammer hardware bug can be exploited in a remote adversary setting to launch successful attacks completely from software. We also discussed that various Rowhammer-based attacks were presented by the related work in the recent past, such as exploits against the operating system kernel, and briefly presented an initial defense approach that blacklists vulnerable memory locations. However, since the security of blacklisting-based isolation approaches relies on the quality of the blacklist, a more general defense approach would be desirable. Hence, in this chapter we present CATT [5], the first software-only defense against Rowhammer attacks targeting kernel memory. The main idea behind our defense strategy is to partition DRAM allocations in such a way that unprivileged memory can never be physically co-located with privileged memory.

8.1 ON THE NECESSITY OF SOFTWARE DEFENSES AGAINST ROWHAMMER

CPU-enforced memory protection is fundamental to modern computer security: for each memory access request, the CPU verifies whether this request meets the memory access policy. However, the infamous Rowhammer attack [63] undermines this access control model by exploiting a hardware fault (triggered through software) to flip targeted bits in memory. The cause for this hardware fault is due to the tremendous density increase of memory cells in modern DRAM chips, allowing electrical charge (or the change thereof) of one memory cell to affect that of an adjacent memory cell. Unfortunately, increased refresh rates of DRAM modules – as suggested by some hardware manufacturers – cannot eliminate this effect [253]. In fact, the fault appeared as a surprise to hardware manufacturers, simply because it does not appear during normal system operation, due to caches. Rowhammer attacks repetitively read (*hammer*) from the same physical memory address in very short time intervals which eventually leads to a bit flip in a physically co-located memory cell.

Although it might seem that single bit flips are not per-se dangerous, recent attacks demonstrate that Rowhammer can be used to undermine access control policies and manipulate data in various ways. In particular, it allows for tampering with the isolation between user and kernel mode [72]. For this, a malicious user-mode application locates vulnerable memory cells and forces the operating system to fill the physical memory with page-table entries (PTEs), i.e., entries that define access policies to memory pages. Manipulating one PTE by means of a bit flip allows the malicious application to alter memory access policies, building a custom page table hierarchy, and finally assigning kernel permissions to a user-mode memory page. Rowhammer attacks have made use of specific CPU instructions to force DRAM access and avoid cache effects. However, prohibiting applications from executing these instructions, as suggested in [72], is ineffective because recent Rowhammer attacks do no longer depend on special instructions [253]. As such, Rowhammer has become a versatile attack technique allowing compromise of co-located virtual machines [76, 134], and enabling sophisticated control-flow hijacking attacks [17, 18, 152] without requiring memory corruption bugs [72, 75, 135]. Lastly, Rowhammer is not limited to x86-based systems but also applies to mobile devices running ARM processors as well [74].

As mentioned before, memory access control is an essential building block of modern computer security, e.g., to achieve process isolation, isolation of kernel code, and manage read-write-execute permission on memory pages. Modern systems feature a variety of mechanisms to isolate memory, e.g., paging [157], virtualization [258, 259], IOMMU [260], and special execution modes like SGX [157] and SMM [157]. However, these mechanisms enforce their isolation through hardware that mediates the physical memory accesses (in most cases the CPU). Hence, memory assigned to isolated entities can potentially be co-located in physical memory on the same bank. Since a Rowhammer attack induces bit flips in co-located memory cells, it provides a subtle way to launch a remote attack to undermine memory isolation.

The common belief is that the Rowhammer fault cannot be fixed by means of any software update, but requires production and deployment of redesigned DRAM modules. Hence, existing legacy systems will remain vulnerable for many years, if not forever. An initial defense approach performed through a BIOS update to increase the DRAM refresh rate was unsuccessful as it only slightly increased the difficulty to conduct the attack [72]. The only other software-based mitigation of rowhammer, we are aware of, is a heuristic-based approach that relies on hardware performance counters [253]. However, it induces a worst-case overhead of 8% and suffers from false positives which impedes its deployment in practice.

We present the first practical software-based defense against Rowhammer attacks that can instantly protect existing vulnerable legacy systems without suffering from any performance overhead and false positives. From all the presented Rowhammer attacks [72–76, 134, 135], only those which compromise kernel memory have been publicly reproduced in successful end-to-end attacks. Unfortunately, attacks that target the OS are also challenging to mitigate without replacing or fixing vulnerable hardware. Other attacks can be either mitigated by disabling certain system features, or are impractical for real-world attacks: Rowhammer attacks on virtual machines [76, 134] heavily depend on *memory deduplication* which is disabled in most production environments by default. Further, attacks launched from a webbrowser require more than half an hour in practice [135]. Hence, for the prototype implementation of our partitioning policy we focus on the more practical Rowhammer attacks that target kernel memory from malicious user-space processes.

8.2 ASSUMPTIONS AND THREAT MODEL

We present the design and implementation of a practical mitigation scheme, called CATT, that does not aim to prevent bit flips but rather remove the dangerous effects (i.e., exploitation) of bit flips. This is achieved by limiting bit flips to memory pages

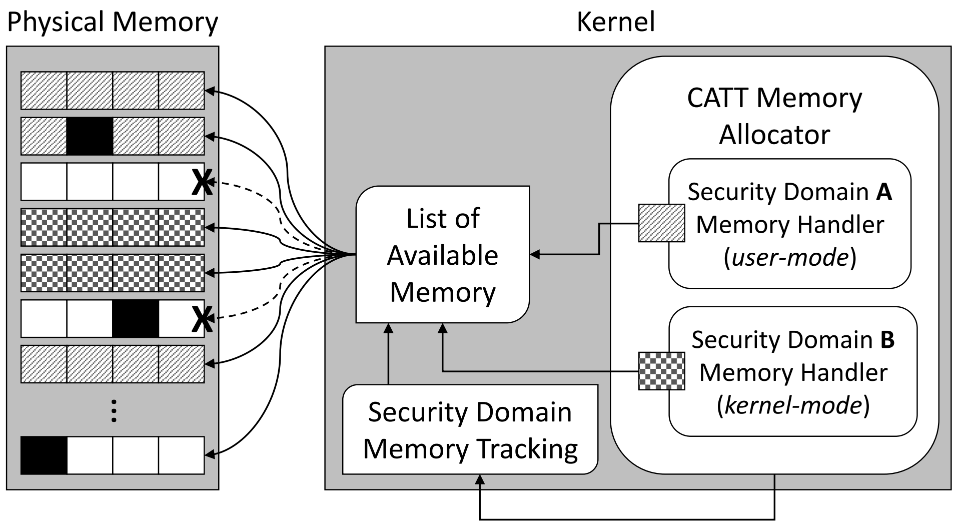

Physical Memory
Kernel
CATT Memory Allocator
Security Domain **A** Memory Handler (*user-mode*)
Security Domain **B** Memory Handler (*kernel-mode*)
List of Available Memory
Security Domain Memory Tracking

Figure 8.1: CATT constrains bit flips to the process' security domain.

that are already in the address space of the malicious application, i.e., memory pages that are per-se untrusted. For this, we extend the operating system kernel to enforce a strong physical isolation of user and kernel space.

Our threat model is in line with related work [72, 73, 75, 76, 134, 135]. The detailed assumptions for our Rowhammer-defense for the OS, called CATT, are as follows:

- We assume that the operating system kernel is not vulnerable to software attacks. While this is hard to implement in practice it is a common assumption in the context of rowhammer attacks.
- The attacker controls an unprivileged process, and hence, can execute arbitrary code in userland but has only limited access to other system resources which are protected by the kernel through mandatory and discretionary access control.
- We assume that the system's RAM is vulnerable to rowhammer attacks. Many commonly used systems (see Table 8.1) include vulnerable RAM.

8.3 DESIGN OF CATT

In this section, we present the high-level idea and design of our practical software-based defense against rowhammer attacks. Our defense, called *CAn't Touch This* (CATT), tackles the malicious *effect* of rowhammer-induced bit flips by instrumenting the operating system's memory allocator to constrain bit flips to the boundary where the attacker's malicious code executes. CATT is completely transparent to applications, and does not require any hardware changes.

8.3.1 *Overview*

Note that our initial defense strategy (presented in the last Chapter) takes a snapshot of vulnerable memory addresses. Since there is no security guarantee that vulnerable memory changes over time or other memory addresses get vulnerable under different test conditions, we developed a novel, long-term protection strategy against Rowhammer attacks, which does not rely on any knowledge about vulnerable memory locations. Instead, our second defense, called CATT, follows a different and more generic defense strategy: it tolerates rowhammer-induced bit flips, but prevents bit flips from affecting memory belonging to higher-privileged *security domains*, e.g., the operating system kernel or co-located virtual machines. The general idea of CATT is to tolerate bit flips by confining the attacker to memory that is already under her control. This is fundamentally different from all previously proposed defense approaches that aimed to prevent bit flips. In particular, CATT prevents bit flips from affecting memory belonging to higher-privileged *security domains*, e.g., the operating system kernel or co-located virtual machines. As a general requirement, an adversary has to bypass the CPU cache to conduct a Rowhammer attack. Further, the attacker must arrange the physical memory layout such that the targeted data is stored in a row that is physically adjacent to rows that are under the control of the attacker. Hence, CATT ensures that memory between these two entities is physically separated by at least one row. [1]

To do so, CATT extends the physical memory allocator to partition the physical memory into security domains.

Figure 8.1 illustrates the concept. Without CATT the attacker is able to craft a memory layout, where two aggressor rows enclose a victim row of a higher-privileged domain. With CATT in place, the rows which are controlled by the attacker are grouped into the security domain A, whereas memory belonging to higher-privileged entities resides with their own security domain (e.g., the security domain B). Both domains are physically separated by at least one row which will not be assigned to any security domain.

8.3.2 *Security Domains*

Privilege escalation attacks are popular and pose a severe threat to modern systems. In particular, the isolation of kernel and user-mode is critical and the most appealing attack target. If a user-space application gains kernel privileges, the adversary can typically compromise the entire system. We define and maintain two security domains: a security domain for kernel memory allocations, and one security domain for user-mode memory allocations (see also Figure 8.1). While our prototype focuses on this kernel setting, the underlying principle of spatial DRAM isolation supports many different scenarios. For instance, in virtualized environments, virtual machines as well as the hypervisor can be treated as individual security domains. To prevent Rowhammer attacks crossing the isolation boundaries of processes in user-mode, CATT would need to assign security domains at the granularity of processes. However, many security domains associated with many memory allocations and deallocations could potentially lead to fragmentation. To

1 Kim et al. [63] mention that the rowhammer fault could potentially affect memory cells of directly adjacent rows, but also memory cells of rows that are next to the adjacent row. Although we did not encounter such cases in our experiments, CATT supports multiple row separation between adversary and victim data memory.

resolve fragmentation issues it would be possible to group untrusted system entities (such as non-critical processes) into the same domain.

8.3.3 *Challenges*

The physical isolation of data raises the challenge of how to effectively isolate the memory of different system entities. To tackle this challenge, we first require knowledge of the mapping between physical addresses and memory banks. Since an attacker can only corrupt data within one bank, but not across banks, CATT only has to ensure that security domains of different system entities are isolated within each bank. However, as mentioned in Section 7.1, hardware vendors do not specify the exact mapping between physical address and banks. Fortunately, Pessl et al. [243] and Xiao et al. [76] provide a methodology to reverse engineer the mapping. For CATT, we use this methodology to discover the physical addresses of rows.

We need to ensure that the physical memory management component is aware of the isolation policy. This is vital as the memory management components have to ensure that newly allocated memory is adjacent only to memory belonging to the same security domain. To tackle this challenge, we instrumented the memory allocator to keep track of the domain association of physical memory and serve memory requests by selecting free memory from different pools depending on the security domain of the requested memory.

8.4 IMPLEMENTATION

Our software-based defense is based on modifications to low-level system software components, i.e., the physical memory allocator of the operating system kernel. In our proof-of-concept implementation of CATT, we focus on hardening Linux against rowhammer-based attacks since its memory allocator is open-source. We successfully applied the mentioned changes to the x86-kernel version 4.6 and the Android kernel for Nexus devices in version 4.4. We chose Linux as our target OS for our proof-of-concept implementations for two reasons: (1) its source code is freely available, and (2) it is widely used on workstations and mobile devices. In the following we will explain the implementation of CATT's policy enforcement mechanism in the Linux kernel which allows for the partitioning of physical memory into isolated security domains. We note that CATT targets both x86 and ARM-based systems. Until today, rowhammer attacks have only been demonstrated for these two prominent architectures.

The basic idea underlying our software-based rowhammer defense is to physically separate rows that belong to different security domains. Operating systems are not per-se aware of the notions of cells and rows, but rather build memory management based on paging. Commodity operating systems use paging to map virtual addresses to physical addresses. The size of a page varies among architectures. On x86 and ARM, the page size is typically 4096 bytes (4K). As we described in Section 7.1, DRAM hardware consists of much smaller units of memory, i.e., individual memory cells storing single bits. Eight consecutive memory cells represent a byte, 4096 consecutive bytes a page frame, two to four page frames a row.

Hence, our implementation of CATT changes low-level components of the kernel to make the operating system aware of the concept of memory rows.

In the following, we describe how we map individual memory pages to domains, keep track of different domains, modify the physical memory allocator, and define partitioning policies for the system's DRAM hardware.

8.4.1 *Mapping Page Frames to Domains*

To be able to decide whether two pages belong to the same security domain we need to keep track of the security domain for each page frame. Fortunately, the kernel already maintains meta data about each individual page frame. More specifically, each individual page frame is associated with exactly one meta data object (`struct page`). The kernel keeps a large array of these objects in memory. Although these objects describe physical pages, this array is referred to as *virtual memory map*, or `vmemmap`. The Page Frame Number (PFN) of a physical page is used as an offset into this array to determine the corresponding `struct page` object. To be able to associate a page frame with a security domain, we extend the definition of `struct page` to include a field that encodes the security domain. Since our prototype implementation targets rowhammer attacks that aim at violating the separation of kernel and user-space, we encode security domain 0 for kernel-space, and 1 for user-space.

8.4.2 *Tracking Security Domains*

The extension of the page frame meta data objects enables us to assign pages to security domains. However, this assignment is dynamic and changes over time. In particular, a page frame may be requested, allocated, and used by one domain, after it has been freed by another domain. Note that this does not violate our security guarantees, but is necessary for the system to manage physical memory dynamically. Yet, we need to ensure that page frames being reallocated continue to obey our security policy. Therefore, we reset the security domain upon freeing a page.

Upon memory allocation, CATT needs to correctly set the security domain of the new page. To do so, we require information about the requesting domain. For our case, where we aim at separating kernel and user-space domains, CATT utilizes the call site information, which is propagated to the memory allocator by default. Specifically, each allocation request passes a range of flags to the page allocator. These flags encode whether an allocation is intended for the kernel or the user-space. We leverage this information and separate the two domains by setting the domain field of the respective page frame.

When processes request memory, the kernel initially only creates a virtual mapping without providing actual physical page frames for the process. Instead, it only assigns physical memory on demand, i.e., when the requesting process accesses the virtual mapping a page fault is triggered. Thereafter, the kernel invokes the physical page allocator to search for usable pages and installs them under the virtual address the process attempted to access. We modified the page fault handler, which initiates the allocation of a new page, to pass information about the security domain to the page allocator. Next, the page is allocated according to our policy and sets the domain field of the page frame's meta data object to the security domain of the interrupted process.

8.4.3 *Modifying the Physical Page Allocator*

The Linux kernel uses different levels of abstraction for different memory allocation tasks. The physical page allocator, or *zoned buddy allocator* is the main low-level facility handling physical page allocations. It exports its interfaces through functions such as `alloc_pages`, which can be used by other kernel components to request physical pages. In contrast to higher-level allocators, the buddy allocator only allows for allocating sets of memory pages with a cardinality which can be expressed as a power of two (this is referred to as the *order* of the allocation). Hence, the buddy allocator's smallest level of granularity is a single memory page. We modify the implementation of the physical page allocator in the kernel to include a mechanism for separating and isolating allocated pages according to the security domain of the origin of the allocation request. In particular, the page allocator already performs maintenance checks on free pages. We extend these maintenance checks to add our partitioning policy before the allocator returns a physical page. If this check fails, the page allocator is not allowed to return the page in question, but has to continue its search for another free page.

8.4.4 *Defining DRAM Partitioning Policies*

Separating and isolating different security domains is essential to our proposed mitigation. For this reason, we incorporate detailed knowledge about the platform and its DRAM hardware configuration into our policy implementation. While our policy implementation for a target system largely depends on its architecture and memory configuration, this does not represent a fundamental limitation. Indeed, independent research [76, 243] has provided the architectural details for the most prevalent architectures, i.e., it shows that the physical address to DRAM mapping can be reverse engineered automatically for undocumented architectures. Hence, it is possible to develop similar policy implementations for architectures and memory configurations beyond x86 and ARM. We build on this prior research and leverage the physical address to DRAM mapping information to enforce strict physical isolation. In the following, we describe our implementation of the partitioning strategy for isolating kernel and user-space.

Kernel-User Isolation. To achieve physical separation of user and kernel space we adopt the following strategy: we divide each bank into a top and a bottom part, with a separating row in-between. Page frames for one domain are exclusively allocated from the part that was assigned to that domain. The part belonging to the kernel domain is determined by the physical location of the kernel image.[2] As a result, user and kernel space allocations may be co-located within one bank, but never within adjacent rows. Different partitioning policies would be possible in

2 This is usually at 1MB, although Kernel Address Space Layout Randomization (KASLR) may slightly modify this address according to a limited offset.

System	Operating System	System Model
S1	Ubuntu 14.04.4 LTS	Dell OptiPlex 7010
S2	Debian 8.2	Dell OptiPlex 990
S3	Kali Linux 2.0	Lenovo ThinkPad x220

Table 8.1: Model numbers of the vulnerable systems used for our evaluation.

theory: for instance, we could confine the kernel to a certain DRAM bank to avoid co-location of user domains within a single bank. However, this would likely result in a severe increase of memory latency, since reads and writes to a specific memory bank are served by the bank's row buffer. The benefit of our partitioning policy stems from the fact that we distribute memory belonging to the kernel security domain over multiple banks thereby not negatively impacting performance. For our solution towards kernel isolation, we only need to calculate the row index of a page frame. More specifically, we calculate this index from the physical address (PA) in the following way:

$$Row(PA) := \frac{PA}{PageSize \cdot PagesPerDIMM \cdot DIMMs}$$

$$\text{where } PagesPerDIMM := PagesPerRow \cdot BanksPerRank \cdot RanksPerDIMM$$

Since all possible row indices are present once per bank, this equation determines the row index of the given physical address.[3] We note that this computation is in line with the available rowhammer exploits [72] and the reported physical to DRAM mapping recently reverse engineered [76, 243]. Since the row size is the same for all Intel architectures prior to Skylake [75], our implementation for this policy is applicable to a wide range of system setups, and can be adjusted without introducing major changes to fit other configurations as well.

8.5 SECURITY EVALUATION

The main goal of our software-based defense is to protect legacy systems from rowhammer attacks. We tested the effectiveness of CATT on diverse hardware configurations. Among these, we identified three hardware configurations, where we observed many reproducible bit flips. Table 8.1 and Table 8.2 list the exact configurations of the three platforms we used for our evaluation. We test the effectiveness of CATT with respect to two different attack scenarios. For the first scenario we systematically search for reproducible bit flips based on a tool published by Gruss

3 The default values for DDR3 on x86 are 4K for the page size, 2 pages per row, 8 banks per rank, 2 ranks per DIMM and between 1 one 4 DIMMs per machine. For DDR4 the number of banks per rank was doubled. DDR4 is supported on x86 starting with Intel's Skylake and AMD's Zen architecture.

et al.[4] Our second attack scenario leverages a real-world rowhammer exploit[5] published by Google's Project Zero. We compared the outcome of both attacks on our vulnerable systems before and after applying CATT. As shown in Table 8.3, both tests only succeed when our protection is not in place. Next, we elaborate on the two attack scenarios and their mitigation in more detail.

8.5.1 *Rowhammer Testing Tool*

We use a slightly modified version of the double-sided rowhammering tool, which is based on the original test by Google's Project Zero [72]. Specifically, we extended the tool to also report the aggressor physical addresses, and adjusted the default size of the fraction of physical memory that is allocated for the test. The tool scans the allocated memory for memory cells that are vulnerable to the rowhammer attack. To provide comprehensive results, the tool needs to scan the entire memory of the system. However, investigating the entire memory is hard to achieve in practice since some parts of memory are always allocated by other system components. These parts are therefore not available to the testing tool, i.e., memory reserved by operating system. To achieve maximum coverage, the tool allocates a huge fraction of the available memory areas. However, due to the lazy allocation of Linux the allocated memory is initially not mapped to physical memory. Hence, each mapped virtual page is accessed at least once, to ensure that the kernel assigns physical pages. Because user space only has access to the virtual addresses of these mappings, the tool exploits the `/proc/pagemap` kernel interface to retrieve the physical addresses. As a result, most of the systems physical memory is allocated to the rowhammering tool.

Afterwards, the tool analyzes the memory in order to identify potential victim and aggressor pages in the physical memory. As the test uses the double-sided rowhammering approach two aggressor pages must be identified for every potential victim page. Next, all potential victim pages are challenged for vulnerable bit flips. For this, the potential victim page is initialized with a fixed bit pattern and "hammered" by accessing and flushing the two associated aggressor pages. This ensures that all of the accesses activate a row in the respective DRAM module. This process is repeated 10^6 times.[6] Lastly, the potential victim address can be checked for bit flips by comparing its memory content with the fixed pattern bit. The test outputs a list of addresses for which bit flips have been observed, i.e., a list of victim addresses.

PRELIMINARY TESTS FOR VULNERABLE SYSTEMS. Using the rowhammering testing tool we evaluated our target systems. In particular, we were interested in systems that yield reproducible bit flips, as only those are relevant for practical rowhammer attacks. This is because an attacker cannot force the system to allocate page tables at a certain physical position in RAM. In contrast, the attacker sprays the memory with page tables to increase her chance of hitting the desired memory location.

4 https://github.com/IAIK/rowhammerjs/tree/master/native
5 https://bugs.chromium.org/p/project-zero/issues/detail?id=283
6 This value is the hardcoded default value. Prior research [63, 71] reported similar numbers.

	CPU			RAM		
System	Version	Cores	Speed	Size	Speed	Manufacturer
S1	i5-3570	4	3.40GHz	2x2GB	1333 MHz	Hynix Hyundai
				1x4GB	1333 MHz	Corsair
S2	i7-2600	4	3.4GHz	2x4GB	1333 MHz	Samsung
S3	i5-2520M	4	2.5GHz	2x4GB	1333 MHz	Samsung

Table 8.2: Technical specifications of the vulnerable systems used for our evaluation.

	Rowhammer Exploit: Success (avg. # of tries)	
	Vanilla System	CATT
S1	✓(11)	✗(3821)
S2	✓(42)	✗(3096)
S3	✓(53)	✗(3768)

Table 8.3: Results of our security evaluation. We executed the exploit continuously on each system for more than 48 hours and found that CATT mitigates real-world Rowhammer exploits against the kernel.

Hence, we configured the rowhammering tool to only report memory addresses where bit flips can be triggered repeatedly. We successively confirmed that this list indeed yields reliable bit flips by individually triggering the reported addresses and checking for bit flips within an interval of 10 seconds. Additionally, we tested the bit flips across reboots through random sampling.

The three systems mentioned in Table 8.1 and Table 8.2 are highly susceptible to reproducible bit flips. Executing the rowhammer test on these three times and rebooting the system after each test run, we found 133 pages with exploitable bit flips for S1, 31 pages for S2, and 23 pages for S3.

To install CATT, we patched the Linux kernel of each system to use our modified memory allocator. Recall that CATT does not aim to prevent bit flips but rather constrain them to a security domain. Hence, executing the rowhammer test on CATT-hardened systems still locates vulnerable pages. However, in the following, we demonstrate based on a real-world exploit that the vulnerable pages are not exploitable.

8.5.2 *Real-world Rowhammer Exploit*

To further demonstrate the effectiveness of our mitigation, we tested CATT against a real-world rowhammer exploit. The goal of the exploit is to escalate the privileges of the attacker to kernel privileges (i.e., gain root access). To do so, the exploit leverages rowhammer to manipulate the page tables. Specifically, it aims to manipulate the access permission bits for kernel memory, i.e., reconfigure its access permission policy.[7]

To launch the exploit, two conditions need to be satisfied: (1) a page table entry must be present in a vulnerable row, and (2) the enclosing aggressor pages must be allocated in attacker-controlled memory.

Since both conditions are not directly controllable by the attacker, the attack proceeds as follows: the attacker allocates large memory areas. As a result, the operating system needs to create large page tables to maintain the newly allocated memory. This in turn increases the probability to satisfy the aforementioned conditions, i.e., a page table entry will eventually be allocated to a victim page. Due to vast allocation of memory, the attacker also increases her chances that aggressor pages are co-located to the victim page.

Once the preconditions are satisfied, the attacker launches the rowhammer attack to induce a bit flip in victim page. Specifically, the bit flip modifies the page table entry such that a subtree of the paging hierarchy is under the attacker's control. Lastly, the attacker modifies the kernel structure that holds the attacker-controlled user process privileges to elevate her privileges to the superuser root. Since the exploit is probabilistic, it only succeeds in five out of hundred runs (5%). Nevertheless, a single successful run allows the attacker to compromise of the entire system.

EFFECTIVENESS OF CATT. Our defense mechanism does not prevent the occurrence of bit flips on a system. Hence, we have to verify that bit flips cannot affect data of another security domain. Rowhammer exploits rely on the fact that such a cross domain bit flip is possible, i.e., in the case of our exploit it aims to induce a bit flip in the kernel's page table entries.

However, since the exploit by itself is probabilistic, an unsuccessful attempt does not imply the effectiveness of CATT. As described above, the success rate of the attack is about 5%. After deploying CATT on our test systems we repeatedly executed the exploit to minimize the probability of the exploit failing due to the random memory layout rather than due to our protection mechanism. We automated the process of continuously executing the exploit and ran this test for 48 h, on all three test systems. In this time frame the exploit made on average 3500 attempts of which on average 175 should have succeeded. However, with CATT, none of the attempts was successful. Hence, as expected, CATT effectively prevents rowhammer-based exploits.

As we have demonstrated, CATT successfully prevents the original attack developed on x86 by physically isolating pages belonging to the kernel from the user-space domain. In addition to that, the authors of the Drammer exploit [74] confirm that CATT prevents their exploit on ARM. The reason is, that they follow the same strategy as in the original kernel exploit developed by Project Zero, i.e., corrupting page table entries in the kernel from neighboring pages in user space. Hence, CATT effectively prevents rowhammer exploits on ARM-based mobile platforms as well.

7 A second option is to manipulate page table entries in such a way that they point to attacker controlled memory thereby allowing the attacker to install new arbitrary memory mappings. The details of this attack option are described by Seaborn et al. [72].

8.6 PERFORMANCE EVALUATION

One of our main goals is practicability, i.e., inducing negligible performance overhead. To demonstrate practicability of our defense, we thoroughly evaluated the performance and stability impact of CATT on different benchmark and testing suites. In particular, we used the SPEC CPU2006 benchmark suite [261] to measure the impact on CPU-intensive applications, LMBench3 [168] for measuring the overhead of system operations, and the Phoronix test suite [169] to measure the overhead for common applications. We use the Linux Test Project, which aims at stress testing the Linux kernel, to evaluate the stability of our test system after deploying CATT. We performed all of our performance evaluation on system S2 (cf. Table 8.2).

8.6.1 *Run-time Overhead*

We briefly summarize the results of our performance benchmarks. In general, the SPEC CPU2006 benchmarks measure the impact of system modifications on CPU intensive applications. Since our mitigation mainly affects the physical memory management, we did not expect a major impact on these benchmarks. However, since these benchmarks are widely used and well established we included them in our evaluation. In fact, we observe a minimal performance improvement for CATT by 0.49% which we attribute to measuring inaccuracy. Such results have been reported before when executing a set of benchmarks for the same system with the exact same configuration and settings. Hence, we conclude that CATT does not incur any performance penalty.

LMBench3 is comprised of a number of micro benchmarks which target very specific performance parameters, e.g., memory latency. For our evaluation, we focused on micro benchmarks that are related to memory performance and excluded networking benchmarks. Similar to the previous benchmarks, the results fluctuate on average between -0.4% and 0.11%. Hence, we conclude that our mitigation has no measurable impact on specific memory operations.

Finally, we tested the impact of our modifications on the Phoronix benchmarks. In particular, we selected a subset of benchmarks[8] that, on one hand, aim to measure memory performance (IOZone and Stream), and, on the other hand, test the performance of common server applications which usually rely on good memory performance.

To summarize, our rigorous performance evaluation with the help of different benchmarking suites did not yield any measurable overhead. This makes CATT a highly practical mitigation against rowhammer attacks.

8 The Phoronix benchmarking suite features a large number of tests which cover different aspects of a system. By selecting a subset of the available tests we do not intend to improve our performance evaluation. On the contrary, we choose a subset of tests that is likely to yield measurable performance overhead, and excluded tests which are unrelated to our modification, e.g., GPU or machine learning benchmarks.

Linux Test Project	Vanilla	CATT
clone	✓	✓
ftruncate	✓	✓
prctl	✓	✓
ptrace	✓	✓
rename	✓	✓
sched_prio_max	✓	✓
sched_prio_min	✓	✓
mmstress	✓	✓
shmt	✗	✗
vhangup	✗	✗
ioctl	✗	✗

Table 8.4: Result for individual stress tests from the Linux Test Project.

8.6.2 *Memory Overhead*

CATT prevents the operating system from allocating certain physical memory pages. The memory overhead of CATT is constant and depends solely on number of memory rows per bank. Per bank, CATT omits one row to provide isolation between the security domains. Hence, the memory overhead is $1/\#rows$ ($\#rows$ being rows per bank). While the number of rows per bank is dependent on the system architecture, it is commonly in the order of 2^{15} rows per bank [9], i.e., the overhead is $2^{-15} \hat{=} 0,003\%$.

8.6.3 *Robustness*

Our mitigation restricts the operating system's access to the physical memory. To ensure that this has no effect on the overall stability, we performed numerous stress tests with the help of the Linux Test Project (LTP) [170]. These tests are designed to stress the operating system to identify problems. We first run these tests on a vanilla Debian 8.2 installation to receive a baseline for the evaluation of CATT. We summarize our results in Table 8.4, and report no deviations for our mitigation compared to the baseline. Further, we also did not encounter any problems during the execution of the other benchmarks. Thus, we conclude that CATT does not affect the stability of the protected system.

9 https://lackingrhoticity.blogspot.de/2015/05/how-physical-addresses-map-to-rows-and-banks.html

8.7 DISCUSSION

Our prototype implementation targets Linux-based systems. Linux is open-source allowing us to implement our defense. Further, all publicly available rowhammer attacks target this operating system. CATT can be easily ported to memory allocators deployed in other operating systems. In this section, we discuss in detail the generality of our software-based defense against rowhammer.

8.7.1 *Applying CATT to Mobile Systems*

The rowhammer attack is not limited to x86-based systems, but has been recently shown to also affect the ARM platform [74]. The ARM architecture is predominant in mobile systems, and used in many smartphones and tablets. As CATT is not dependent on any x86 specific properties, it can be easily adapted for ARM based systems. We demonstrate this by applying our extended physical memory allocator to the Android kernel for Nexus devices in version 4.4. Since there are no major deviations in the implementation of the physical page allocator of the kernel between Android and stock Linux kernel, we did not encounter any obstacles during the port.

8.7.2 *Single-sided Rowhammer Attacks*

From our detailed description in Section 8.3 one can easily follow that our proposed solution can defeat all known rowhammer-based privilege escalation attacks in general, and single-sided rowhammer attacks [74] in particular. In contrast to double-sided rowhammer attacks, single-sided rowhammer attacks relax the adversary's capabilities by requiring that the attacker has control over only one row adjacent to the victim memory row. As described in more detail in Section 8.3, CATT isolates different security domains in the physical memory. In particular, it ensures that different security domains are separated by at least one buffer row that is never used by the system. This means that the single-sided rowhammer adversary can only flip bits in own memory (that it already controls), or flip bits in buffer rows.

8.7.3 *Benchmarks Selection*

We selected our benchmarks to be comparable to the related literature. Moreover, we have done evaluations that go beyond those in the existing work to provide additional insight. Hereby, we considered different evaluation aspects: We executed SPEC CPU2006 to verify that our changes to the operating system impose no overhead of user-mode applications. Further, SPEC CPU2006 is the most common benchmark in the field of memory-corruption defenses, hence, our solutions can be compared to the related work. LMBench3 is specifically designed to evaluate the performance of common system operations, and used by the Linux kernel developers to test whether changes to the kernel affect the performance. As such LMBench3 includes many tests. For our evaluation we included those benchmarks that perform memory operations and are relevant for our defense. Finally, we selected a number of common applications from the Phoronix test suite as macro benchmarks, as well as the *pts/memory* tests which are designed to measure the RAM and cache performance. For all our benchmarks we did not observe any measurable overhead.

8.7.4 *Vicinity-less Rowhammering*

All previous Rowhammer attacks exploit rows which are physically co-located [72, 74, 75, 135]. However, while Kim et al. [63] suggested that physical adjacency accounts for the majority of possible bit flips, they also noted that this was not always the case. More specifically, they attributed potential aggressor rows with a greater row distance to the *re-mapping* of faulty rows: DRAM manufacturers typically equip their modules with around 2% of spare rows, which can be used to physically replace failing rows by re-mapping them to a spare row [262]. This means, that physically adjacent spare rows can be assigned to arbitrary row indices, potentially undermining our isolation policy. For this, an adversary requires a way of determining pairs of defunct rows, which are re-mapped to physically adjacent spare rows. We note that such a methodology can also be used to adjust our policy implementation, e.g., by disallowing any spare rows to be assigned to kernel allocations. Hence, re-mapping of rows does not affect the security guarantees provided by CATT.

9 SIDE-CHANNEL RESILIENT KERNEL-SPACE RANDOMIZATION.

While remote-fault injection attacks, like Rowhammer, are able to corrupt the state of the victim system, micro-architectural side-channel attacks are generally limited to information disclosure. However, they can have a high real-world impact, since an adversary who leaks the root password through a side channel is able to take remote control over the platform as well. In the recent past, a number of side-channel attacks against operating systems have been demonstrated. In this chapter, we present LAZARUS [6], our general side-channel defense for the kernel. As we will show, our software-only defense successfully stops recent attacks based on micro-architectural hardware vulnerabilities against the OS.

9.1 SIDE-CHANNEL ATTACKS AGAINST KASLR

For more than three decades memory-corruption vulnerabilities have challenged computer security. This class of vulnerabilities enable the attacker to overwrite memory in a way that was not intended by the developer, resulting in a malicious control or data flow. In the recent past kernel vulnerabilities became more prevalent in exploits due to advances in hardening user-mode applications. For example, browsers and other popular targets are isolated by executing them in a sandboxed environment. Consequently, the attacker needs to execute a privilege-escalation attack in addition to the initial exploit to take full control over the system [27, 263–265]. Operating system kernels are a natural target for attackers because the kernel is comprised of a large and complex code base and exposes a rich set of functionality, even to low privileged processes. Molinyawe et al. [266] summarized the techniques used in the infamous Pwn2Own exploiting contest, and concluded that a kernel exploit is required for most privilege-escalations attacks.

In the past kernels were hardened using different mitigation techniques to minimize the risk of memory-corruption vulnerabilities: for example, enforcing the address space to be writable or executable (W$\oplus$X), but never both, to prevent the attacker from injecting new code, and enabling new CPU features like Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Protection Enable (SMEP) to prevent user-mode-aided attacks. Modern kernels are also fortified with kernel Address Space Layout Randomization (KASLR) [162] which randomizes the base address of the code and data section of the kernel at boot time. KASLR forces attackers to customize their exploit for each targeted kernel. Specifically, the attack needs to disclose the randomization secret first, before launching a code-reuse attack.

All modern operating systems leverage kernel-space randomization by means of kernel code randomization (KASLR) [162–164]. However, kernel-space randomization has been shown to be vulnerable to a variety of side-channel attacks. These attacks leverage micro-architectural implementation details of the underlying hardware. These side channel exist, since modern processors share physical resources between privileged and unprivileged execution modes.

While in general memory-corruption vulnerabilities can potentially be exploited to disclose information, recent research demonstrated that side-channel attacks are more powerful because they do not rely on any kernel vulnerabilities [80–83, 133]. These attacks exploit properties of the underlying micro architecture to infer the randomization secret of KASLR. In particular, modern processors share resources such as caches between user mode and kernel mode. The general idea of these attacks is to probe different kernel addresses and measure the execution time of the probe. Since probing time for a valid and a invalid kernel addresses is different, the attacker can infer the randomization secret.

The majority of side-channel attacks against KASLR are based on *paging* [80, 82, 83, 133]. Here, the attacker exploits the timing difference between an aborted memory access to an unmapped kernel address and an aborted memory access to a mapped kernel address.

Side-channel attacks against KASLR exploit the fact that the TLB is shared between user applications and the kernel. As a consequence, the TLB will contain page-table entries of the kernel after switching execution from kernel to a user mode. The attacker then probes parts of the address-space that are expected to be kernel addresses. Here, the exact method of the access depends on the concrete side-channel attack implementation. Since the attacker executes the attack with user privileges, this access will be aborted by the platform. However, the timing difference between access attempt and abort depends on whether the guessed address is cached in the TLB or not, and more specifically, at which level in the TLB hierarchy. Further, the attacker can also measure the timing between existing (traversal of the page-table hierarchy) and not existing mappings (immediate abort). These timing differences can be exploited as a side channel to disclose the randomization secret and several attacks have been demonstrated recently [80, 82, 83, 133].

Hund, et al. [80] published the first side-channel attack to defeat KASLR. They trigger a page fault in the kernel from a user process by accessing an address in kernel space. Although this unprivileged access is correctly denied by the page fault handler, the TLBs are queried during processing of the memory request. They show that the timing difference between exceptions for unmapped and mapped pages can be exploited to disclose the random offset.

Additionally, transactional memory extensions introduced by Intel encapsulate a series of memory accesses to provide enhanced safety guarantees, such as rollbacks. While potentially interesting for the implementation of database systems, erroneous accesses within a transaction are not reported to the operating system. More specifically, if the MMU detects an access violation, the transaction is aborted silently. However, an adversary can measure the timing difference between two aborted transactions to identify privileged addresses, which are cached in TLBs. This enables the attacker to significantly improve over the original page fault timing side-channel attack [82, 133]. The reason is that the page fault handler of the OS is never invoked, significantly reducing the noise in the timing signal.

Furthermore, even individual instructions may leak timing information and can be leveraged in attacks against kernel randomization [83]. In particular, the execution of the `prefetch` instruction of recent Intel processors exhibits a timing difference, which depends directly on the state of the TLBs. As in the case of the side-channel attack exploiting transactional memory, this unprivileged instruction can be used to access privileged addresses, but will fail without invoking the page fault handler of the OS. Its execution time differs for cached kernel addresses. This yields another stealthy side channel that leaks the randomization secret.

9.2 ADVERSARY MODEL AND ASSUMPTIONS

We derive our adversary model from the related offensive work [80–83, 133].

- **Writable $\oplus$ Executable Memory**. The kernel enforces Writable $\oplus$ Executable Memory (W$\oplus$X) which prevents code-injection attacks in the kernel space. Further, the kernel utilizes modern CPU features like SMAP and SMEP [267] to prevent user-mode aided code-injection and code-reuse attacks.
- **Base-address Randomization**. The base address of the kernel code and data region is randomized during the boot [162, 163].
- **Absence of Software-based Information-disclosure Vulnerability**. The kernel does not contain any vulnerabilities that can be exploited to disclose the randomization secret.
- **Malicious Kernel Extension**. The attacker cannot load malicious kernel extensions to gain control over the kernel, i.e., only trusted (or signed) extensions can be loaded.
- **Memory-corruption Vulnerability**. This is a standard assumption for many real-world kernel exploits. The kernel, or a kernel extension contains a memory-corruption vulnerability. The attacker has full control over a user-mode process from which it can exploit this vulnerability. The vulnerability enables the attacker to overwrite a code pointer of the kernel to hijack the control-flow of the kernel. However, the attacker cannot use this vulnerability to disclose any addresses.

While modern kernels suffer from software-based information-disclosure vulnerabilities, information-disclosure attacks based on side channels pose a more severe threat because they can be exploited to disclose information in the absence of software vulnerabilities. We address the problem of side channels, and treat software-based information-disclosure vulnerabilities as an orthogonal problem.

9.3 OUR SIDE-CHANNEL DEFENSE FOR THE KERNEL

In this section, we give an overview of the idea and architecture of LAZARUS, elaborate on the main challenges, and explain in detail how we tackle these challenges.

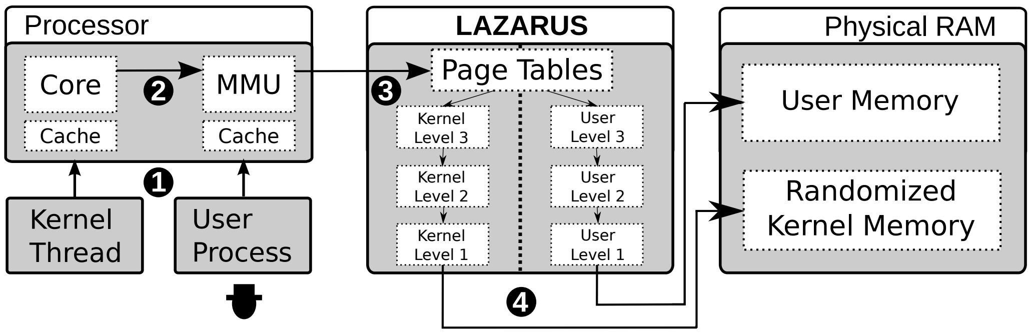


Figure 9.1: The idea behind our side channel protection: An unprivileged user process (1) can exploit the timing side channel for kernel addresses through shared cache access in the MMU paging caches (2). Our defense mitigates this by enforcing (3) a separation between different privilege levels for randomized addresses (4).

9.3.1 *Overview*

Usually, kernel and user mode share the same virtual address space. While legitimate accesses to kernel addresses require higher privilege, these addresses still occupy some parts of the virtual memory space that is visible to user processes. The idea behind our side-channel defense is to strictly and efficiently separate randomized kernel memory from virtual memory in user space.

Our idea is depicted in Figure 9.1. Kernel execution and user space execution usually share a common set of architectural resources, such as the execution unit (Core), and the MMU. The attacker leverages these shared resources in the following way: in step 1, the attacker sets up the user process and memory setting that will leak the randomization secret. The user process then initiates a virtual memory access to a kernel address.

Next, the processor invokes the MMU to check the required privilege level in step 2. Since a user space process does not possess the required privileges to access kernel memory, any such access will ultimately be denied. However, to deny access the MMU has to look up the required privileges in the page tables. These are structured hierarchically with multiple levels, and separate caches on every level. Hence, even denied accesses constitute a timing side-channel that directly depends on the last cached level.

We address 3 the root of this side channel: we separate the page tables for kernel and user space. This effectively prevents side-channel information from kernel addresses to be leaked to user space, because the MMU uses a different page table hierarchy. Thus, while the processor is in user mode, the MMU will not be able to refer to any information about kernel space addresses, as shown in step 4.

9.3.2 *Challenges for Fine-grained Address Space Isolation*

To enable LAZARUS to separate and isolate both execution domains a number of challenges have to be tackled: first, we must provide a mechanism for switching between kernel and user execution at any point in time without compromising the randomized kernel memory (**C1**). More specifically, while kernel and user space no longer share the randomized parts of privileged virtual memory, the system still has to be able to execute code pages in both execution modes. For this reason, we have to enable switching between kernel and user space. This is challenging, because such a transition can happen either through explicit invocation, such as a system call or an exception, or through hardware events, such as interrupts. As we will show our defense handles both cases securely and efficiently.

Second, we have to prevent the switching mechanism from leaking any side-channel information (**C2**). Unmapping kernel pages is also challenging with respect to side-channel information, i.e., unmapped memory pages still exhibit a timing difference compared to mapped pages. Hence, LAZARUS has to prevent information leakage through probing of unmapped pages.

Third, our approach has to minimize the overhead for running applications to offer a practical defense mechanism (**C3**). Implementing strict separation of address spaces efficiently is involved, since we only separate those parts of the address space that are privileged and randomized. We have to modify only those parts of the page table hierarchy which define translations for randomized addresses.

In the following we explain how our defense meets these challenges.

C1: Kernel-User Transitioning. Processor resources are time-shared between processes and the operating system. Thus, the kernel eventually takes control over these resources, either through explicit invocation, or based on a signaling event. Examples for explicit kernel invocations are *system calls* and *exceptions*. These are synchronous events, meaning that the user process generating the event is suspended and waiting for the kernel code handling the event to finish.

On the one hand, after transitioning from user to kernel mode, the event handler code is no longer mapped in virtual memory because it is located in the kernel. Hence, we have to provide a mechanism to restore this mapping when entering kernel execution from user space.

On the other hand, when the system call or exception handler finishes and returns execution to the user space process, we have to erase those mappings again. Otherwise, paging entries might be shared between privilege levels. Since all system calls enter the kernel through a well-defined hardware interface, we can map and unmap the page tables by modifying this central entry point.

Transitioning between kernel and user space execution can also happen through *interrupts*. A simple example for this type of event is the timer interrupt, which is programmed by the kernel to trigger periodically in fixed intervals. In contrast to system calls or exceptions, interrupts are asynchronously occurring events, which may suspend current kernel or user space execution at any point in time.

Hence, interrupt routines have to store the current process context before handling a pending interrupt. However, interrupts can also occur while the processor executes kernel code. Therefore, we have to distinguish between interrupts during user or kernel execution to only restore and erase the kernel entries upon transitions to and from user space respectively. For this we facilitate the stored state of the interrupted execution context that is saved by the interrupt handler to distinguish privileged from un-privileged contexts.

This enables LAZARUS to still utilize the paging caches for interrupts occuring during kernel execution.

C2: Protecting the Switching Mechanism. The code performing the address space switching has to be mapped during user execution. Otherwise, implementing a switching mechanism in the kernel would not be possible, because the processor could never access the corresponding code pages. For this reason, it is necessary to prevent these mapped code pages from leaking any side-channel information. There are two possibilities for achieving this.

First, we can map the switching code with a separate offset than the rest of the kernel code section. In this case an adversary would be able to disclose the offset of the switching code, while the actual randomization secret would remain protected.

Second, we can eliminate the timing channel by inserting dummy mappings into the unmapped region. This causes the surrounding addresses to exhibit an identical timing signature compared to the switching code.

Since an adversary would still be able to utilize the switching code to conduct a code-reuse attack in the first case, LAZARUS inserts dummy mappings into the user space page table hierarchy.

C3: Minimizing Performance Penalties. Once paging is enabled on a processor, all memory accesses are mediated through the virtual memory subsystem. This means that a page table lookup is required for every memory access. Since an actual page table lookup results in high performance penalties, the MMU caches the most prominent address translations in the Translation Lookaside Buffer (TLB).

LAZARUS removes kernel addresses from the page table hierarchy upon user space execution. Hence, the respective TLB entries need to be invalidated. As a result, subsequent memory accesses to kernel space will be slower, once kernel execution is resumed.

To minimize these perfomance penalties, we have to reduce the amount of invalidated TLB entries to a minimum but still enforce a clear separation between kernel and user space addresses. In particular, we only remove those virtual mappings, which fall into the location of a randomized kernel area, such as the kernel code segment.

9.4 PROTOTYPE IMPLEMENTATION

We implemented LAZARUS as a prototype for the Linux kernel, version 4.8 for the 64 bit variant of the x86 architecture. However, the techniques we used are generic and can be applied to all architectures employing multi-level page tables. Our patch consists of around 300 changes to seven files, where most of the code results from initialization. Hence, LAZARUS should be easily portable to other architectures. Next, we will explain our implementation details. It consists of the initialization setup, switching mechanism, and how we minimize performance impact.

9.4.1 *Initialization*

We first setup a second set of page tables, which can be used when execution switches to user space. These page tables must not include the randomized portions of the address space that belong to the kernel. However, switching between privileged and unprivileged execution requires some code in the kernel to be mapped upon transitions from user space. We explicitly create dedicated entry points mapped in the user page tables, which point to the required switching routines.

Fixed Mappings. Additionally, there are kernel addresses, which are mapped to fixed locations in the top address space ranges. These *fixmap* entries essentially represent an address-based interface: even if the physical address is determined at boot time, their virtual address is fixed at compile time. Some of these addresses are mapped readable to user space, and we have to explicitly add these entries as well.

We setup this second set of page tables only once at boot time, before the first user process is started. Every process then switches to this set of page tables during user execution.

Dummy Mappings. As explained in Section 9.3, one way of protecting the code pages of the switching mechanism is to insert dummy mappings into the user space page table hierarchy. In particular, we create mappings for randomly picked virtual kernel addresses to span the entire code section. We distribute these mappings in 2M intervals to cover all third-level page table entries, which are used to map the code section. Hence, the entire address range which potentially contains the randomized kernel code section will be mapped in the page table hierarchy used during user space execution.

System Calls. There is a single entry point in the Linux kernel for system calls, which is called the system call handler. We add an assembly routine to execute immediately after execution enters the system call handler. It switches from the predefined user page tables to the kernel page tables and continues to dispatch the requested system call. We added a second assembly routine shortly before the return of the system call handler to remove the kernel page tables from the page table hierarchy of the process and insert our predefined user page tables.

However, contrary to its single entry, there are multiple exit points for the system call handler. For instance, there is a dedicated error path, and fast and slow paths for regular execution. We instrument all of these exit points to ensure that the kernel page tables are not used during user execution.

9.4.2 *Interrupts*

Just like the system call handler, we need to modify the interrupt handler to restore the kernel page tables. However, unlike system calls, interrupts can occur when the processor is in privileged execution mode as well. Thus, to handle interrupts, we need to distinguish both cases. Basically we could look up the current privilege level easily by querying a register. However, this approach provides information about the current execution context, whereas to distinguish the two cases we require the privilege level of the interrupted context.

Fortunately, the processor saves some hardware context information, such as the instruction pointer, stack pointer, and the code segment register before invoking the interrupt handler routine. This means that we can utilize the stored privilege level associated with the previous code segment selector to test the privilege level of the interrupted execution context. We then only restore the kernel page tables if it was a user context.

We still have to handle one exceptional case however: the non-maskable interrupt (NMI). Because NMIs are never maskable, they are handled by a dedicated interrupt handler. Hence, we modify this dedicated NMI handler in the kernel to include our mechanism as well.

9.4.3 *Fine-grained Page Table Switching*

As a software-only defense technique, one of the main goals of LAZARUS is to offer practical performance. While separating the entire page table hierarchy between kernel and user mode is tempting, this approach is impractical.

In particular, switching the entire page table hierarchy invalidates all of the cached TLB entries. This means, that the caches are reset every time and can never be utilized after a context switch. For this reason, we only replace those parts of the page table hierarchy, which define virtual memory mappings for randomized addresses. In the case of KASLR, this corresponds to the code section of the kernel. More specifically, the kernel code section is managed by the last of the 512 level 4 entries.

Thus, we replace only this entry during a context switch between privileged and unprivileged execution. As a result, the caches can still be shared between different privilege levels for non-randomized addresses. As we will discuss in Section 9.5, this does not impact our security guarantees in any way.

9.5 EVALUATION

In this section we evaluate our prototypical implementation for the Linux kernel. First, we show that LAZARUS successfully prevents all of the previously published side-channel attacks. Second, we demonstrate that our defense only incurs negligible performance impact for standard computational workloads.

9.5.1 *Security*

Our main goal is to prevent the leakage of the randomization secret in the kernel to an unprivileged process through paging-based side-channel attacks. For this, we separate the page tables for privileged parts of the address space from the unprivileged parts. We ensure that this separation is enforced for randomized addresses to achieve practical performance.

Because all paging-based exploits rely on the timing difference between cached and uncached entries for privileged virtual addresses, we first conduct a series of timing experiments to measure the remaining side channel in the presence of LAZARUS.

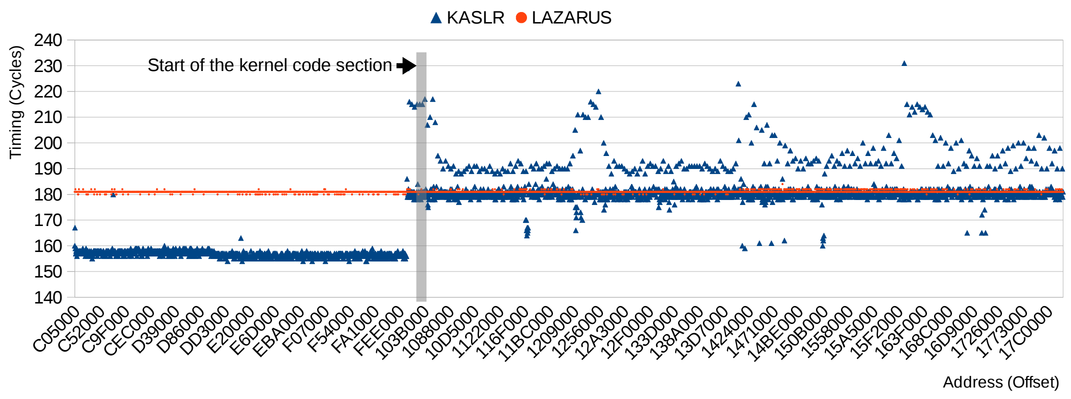


Figure 9.2: Timing side-channel measurements we conducted before (blue) and after (red) we applied LAZARUS.

In a second step, we execute all previously presented side-channel attacks on a system hardened with LAZARUS to verify the effectiveness of our approach.
Effect of LAZARUS on the timing sidechannel. To estimate the remaining timing side-channel information we measure the timing difference for privileged virtual addresses. We access each page in the kernel code section at least once and measure the timing using the `rdtscp` instruction. By probing the privileged address space in this way, we collect a timing series of execution cycles for each kernel code page. The results are shown in Figure 9.2. [1]

The timing side channel is clearly visible for the vanilla KASLR implementation: the start of the actual code section mapping is located around the first visible jump from 160 cycles up to 180 cycles. Given a reference timing for a corresponding kernel image, the attacker can easily calculate the random offset by subtracting the address of the peak from the address in the reference timing.

In contrast to this, the timing of LAZARUS shows a straight line, with a maximum cycle distance of two cycles. In particular, there is no correlation between any addresses and peaks in the timing signal of the hardened kernel. This indicates that our defense approach indeed closes the paging-induced timing channel successfully. We note, that the average number of cycles depicted for LAZARUS are also in line with the timings for cached page table entries reported by related work [82, 83]. To further evaluate the security of our approach, we additionally test it against all previous side-channel attacks.
Real-world side-channel attacks.

We implemented and ran all of the previous side-channel attacks against a system hardened with LAZARUS, to experimentally assess the effectiveness of our approach against real-world attacks.

The first real-world side-channel attack against KASLR was published by Hund et al. [80]. They noted that the execution time of the page fault handler in the OS kernel depends on the state of the paging caches. More specifically, they access kernel addresses from user space which results in a page fault. While this would usually terminate the process causing the access violation, the POSIX standard

1 For brevity, we display the addresses on the x-axis as offsets to the start of the code section (i.e., `0xffffffff80000000`). We further corrected the addresses by their random offset, so that both data series can be shown on top of each other.

allows for processes to handle such events via *signals*. By installing a signal handler for the segmentation violation (`SIGSEGV`), the user process can recover from the fault and measure the timing difference from the initial memory access to the delivery of the signal back to user space. In this way, the entire virtual kernel code section can be scanned and each address associated with its corresponding timing measurement, allowing a user space process to reconstruct the start address of the kernel code section. We implemented and successfully tested the attack against a vanilla Linux kernel with KASLR. In particular, we found that page fault handler exhibits a timing difference of around 30 cycles for mapped and unmapped pages, with an average time of around 2200 cycles. While this represents a rather small difference compared to the other attacks, this is due to the high amount of noise that is caused by the execution path of the page fault handler code in the kernel. [2] When we applied LAZARUS to the kernel the attack no longer succeeded.

Recently, the `prefetch` instruction featured on many Intel x86 processors was shown to enable side-channel attacks against KASLR [83]. It is intended to provide a benign way of instrumenting the caches: the programmer (or the compiler) can use the instruction to provide a hint to the processor to cache a given virtual address.

Although there is no guarantee that this hint will influence the caches in any way, the instruction can be used with arbitrary addresses in principle. This means that a user mode program can prefetch a kernel space address, and execution of the instruction will fail siltently, i.e., the page fault handler in the kernel will not be executed, and no exception will be raised.

However, the MMU still has to perform a privilege check on the provided virtual address, hence the execution time of the `prefetch` instruction depends directly on the state of the TLBs.

We implemented the prefetch attack against KASLR for Linux, and succesfully executed it against a vanilla system to disclose the random offset. Executing the attack against a system hardened with LAZARUS we found the attack to be unsuccessful.

Jang et al. [82] demonstrated yet another KASLR bypass using the Transactional Synchronization Extension (TSX) present in recent Intel x86 CPUs. TSX provides a hardware mechanism which boosts the performance of multi-threaded applications through lock elision [157]. Originally released in Haswell processors, TSX-enabled processors are capable of dynamically determining to serialize threads through lock-protected critical sections if necessary. A processor may abort a TSX transaction if it can not guarantee an *atomic* view from the software's perspective. This may occur when there are conflicting accesses between a logical processor executing a transaction and another logical processor.

TSX will suppress any faults that must be exposed to software if they occur within a transactional region. Memory accesses that cause a page-table walk may abort a transaction, and according to Intel's manual, *will not be made architecturally visible through the behavior of structures such as TLBs.* The timing characteristics of the abort, however, reveal information as to what took place. Herein lies the low-level workings of the *DrK* attack. By causing a page-table walk inside a transactional

2 This was also noted in the original exploit [80].

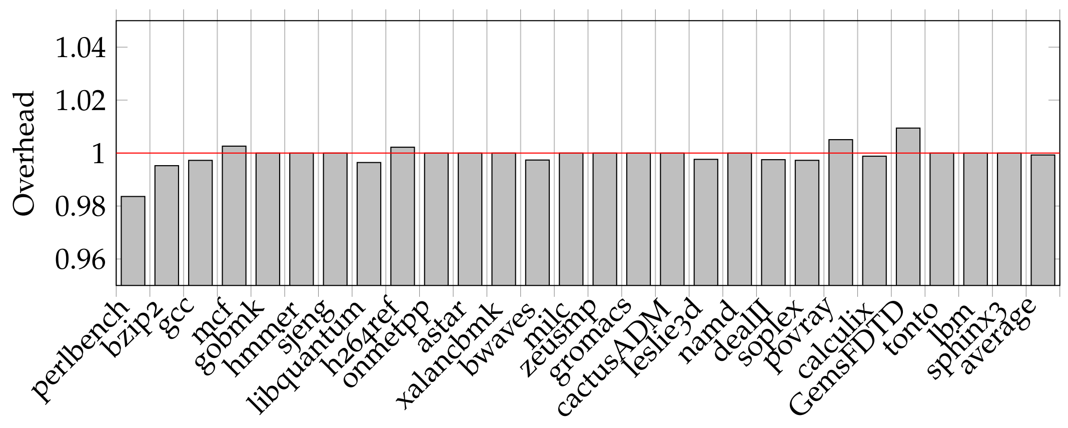


Figure 9.3: SPEC2006 Benchmark Results

block, DrK utilizes timing information on the aborted transaction to disclose the position of kernel pages that are mapped into a program.

The attack proceeds as follows: attempt a memory access to kernel pages inside a transactional block. This causes both a page-table walk and a segmentation fault. Because the TSX hardware masks the segmentation fault, the operating system is never made aware of the event and the CPU executes the abort handler provided by the attacking application. The application captures timing information on how fast the fault aborted. Jang et al. report that a transaction aborts in about 220 or less cycles if the probed page is mapped, whereas it aborts in about 235 cycles or more if unmapped [82]. This sidechannel is used to expose the kernel virtual address space to the userland application.

Probing pages this way under LAZARUS reveals no information. This is because LAZARUS unmaps all kernel code pages from the process, rendering the timing side channel useless. Consequently, any probes to kernel pages return as unmapped. Only the location of LAZARUS and its dummy mappings can be leaked, which proves insufficient for the attacker to disclose the location in memory of the kernel.

9.5.2 *Performance*

We evaluated LAZARUS on a machine with an Intel Core i7-6820HQ CPU clocked at 2.70GHz and 16GB of memory. The machine runs a current release of Arch Linux with kernel version 4.8.14. For our testing, we enabled KASLR in the Linux kernel that Arch Linux ships. We also compiled a secondary kernel with the same configuration and the LAZARUS modifications applied.

We first examine the performance overhead with respect to the industry standard SPEC2006 benchmark [261]. We first ran both the integer and floating point benchmarks in our test platform under the stock kernel with KASLR enabled. We collected these results and performed the test again under the LAZARUS kernel. Our results are shown in Figure 9.3.

The observed performance overhead can be attributed to measurement inaccuracies. Our computed worst case overhead is of 0.943%. We should note that

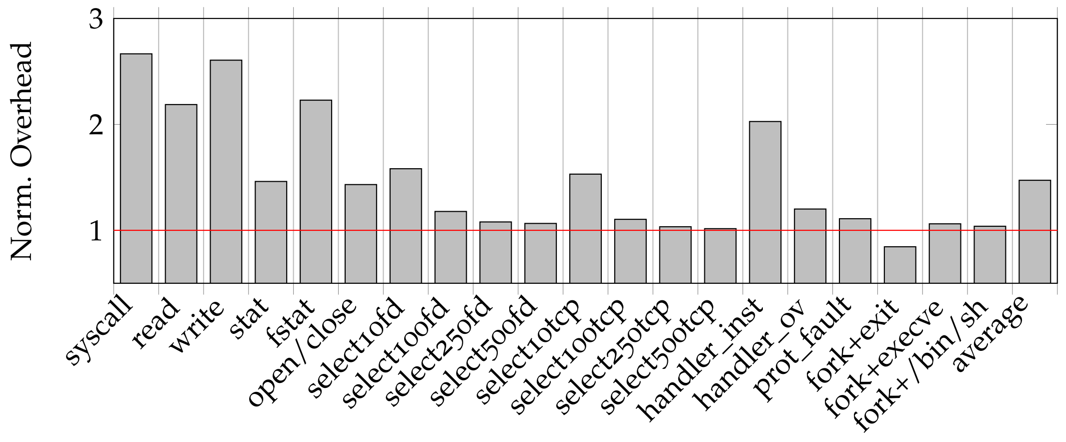


Figure 9.4: LMBench3 Benchmark Results

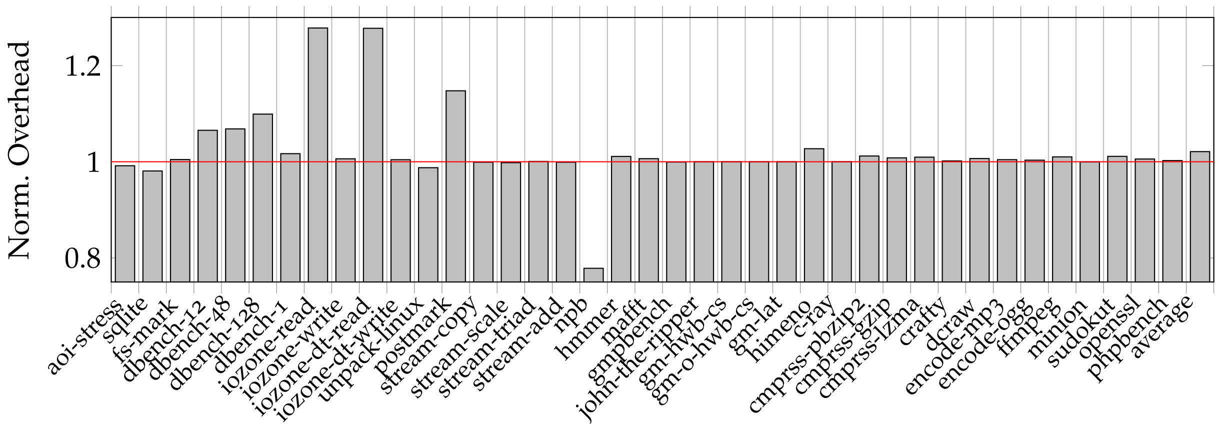


Figure 9.5: Phoronix Benchmark Suite

SPEC2006 is meant to test computational workloads and performs little in terms of context switching.

To better gauge the effects of LAZARUS on the system, we ran the system benchmarks provided by LMBench3 [168]. LMBench3 improves on the context switching benchmarks by eliminating some of the issues present in previous versions of the benchmark, albeit it still suffers issues with multiprocessor machines. For this reason, we disabled SMP during our testing. Our results are presented in Figure 9.4.

We can see how a system call intensive application is affected the most under LAZARUS. This is to be expected, as the page tables belonging to the kernel must be remapped upon entering kernel space. In general, we show a 47% performance overhead when running these benchmarks. We would like to remind the reader, however, that these benchmarks are meant to stress test the performance of the operating system when handling interrupts and do not reflect normal system operation.

In order to get a more realistic estimate of LAZARUS, we ran the Phoronix Test Suite [169]. Specifically, we ran the system and disk benchmarks to test application performance. Our results are shown in Figure 9.5. We show an average performance overhead of 2.1% on this benchmark. This is in agreement with the results provided by the SPEC and LMBench benchmarks. The worst performers were benchmarks that are bound to read operations. We speculate that this is due to the amount of context switches that happen while the read operation is taking place, as a buffer in kernel space needs to be copied into a buffer from userspace or remapped there.

Lastly, we ran the `pgbench` benchmark on a test PostgreSQL databaseand computed a performance overhead of 2.386%.

9.6 DISCUSSION

Next we briefly discuss various real-world aspects of LAZARUS, such as portability and impact on related attacks against the OS.

9.6.1 *Applying LAZARUS to different KASLR implementations*

Relocation of kernel code is an example of how randomization approaches can be used as a defense building block which is implemented by practically all real-world operating systems [162–164]. While a kernel employing control-flow integrity (CFI) [41, 45, 160] does not gain security benefit from randomizing the code section, it might still randomize the memory layout of other kernel memory regions: for instance, it can be applied to the module section, to hide the start address of the code of dynamically loadable kernel modules. Further, randomization was recently proposed as a means to protect the page tables against malicious modification through data-only attacks [2].

Since all of the publicly available attacks focus on disclosing the random offset of the kernel code section, we implemented our proof of concept for KASLR as well. Nonetheless, we note that LAZARUS is not limited to hardening kernel code randomization, but can be applied to other randomization implementations as well. In contrast to the case of protecting KASLR, our defense does not require any special treatment for hiding the low-level switching code if applied to other memory regions.

9.6.2 *Speculative Execution and Side-channel Attacks*

Recently, several attacks demonstrated that vulnerable speculative execution engines of many processors can be exploited to bypass existing access-control features and memory protection [66, 67, 141]. As described in Chapter 2 hardware usually provides primitives for isolating user processes from the operating system by means of virtual memory and privilege separation. When activated, both techniques are usually enforced by the hardware for all software running on the platform. However, due to a bug in current architectures this virtual memory protection can be bypassed during speculative execution with some measurable probability. While violations during speculative execution are not committed to the final instruction stream, and hence, do not have any direct effects, they still have indirect effects, e.g., on the cache hierarchy. The Meltdown attack [66] proceeds in two steps: (i) the unprivileged attacker forces speculative execution of memory accesses that violate the enforced memory protection policy on kernel memory. Afterwards, the attacker (ii) extracts those accessed memory contents through a micro-architectural sidechannel by probing the processor cache. Crucially, such cache loads caused by erroneous speculation are never rolled back. The authors show that an adversary can recover arbitrary physical memory contents through accesses to the direct mapping that is maintained by the OS with up to 500KB/s. Since LAZARUS breaks up the virtual-address space into privileged and unprivileged memory it successfully mitigates Meltdown by blocking the second step of the attack [66].

9.6.3 *Other side-channel attacks on KASLR*

As explained earlier, almost all previously presented side-channel attacks on KASLR exploit the paging subsystem. LAZARUS isolates kernel addresses from the virtual address of user processes by separating their page tables. However, Evtyushkin et al. [81] recently presented the branch target buffer (BTB) side-channel attack, which does not exploit the paging subsystem for virtual kernel addresses.

In particular, they demonstrated how to exploit collisions between branch targets for user and kernel addresses. The attack works by constructing a malicious chain of branch targets in user space, to fill up the BTB, and then executing a previously chosen kernel code path. This evicts branch targets previously executed in kernel mode from the BTB, thus their subsequent execution will take longer.

While the BTB attack was shown to bypass KASLR on Linux, it differs from the paging-based side channels by making a series of assumptions: 1) the BTB has a limited capacity of 10 bits, hence it requires KASLR implementations to deploy a low amount of entropy in order to succeed. 2) it requires the attacker to craft a chain of branch targets, which cause kernel addresses to be evicted from the BTB. For this an adversary needs to reverse engineer the hashing algorithm used to index the BTB. These hashing algorithms are different for every micro architecture, which limits the potential set of targets. 3) the result of the attack can be ambiguous, because any change in the execution path directly effects the BTB contents.

There are multiple ways of mitigating the BTB side-channel attack against KASLR. A straightforward approach is to increase the amount of entropy for KASLR, as noted by Evtyushkin et al. [81]. A more general approach would be to introduce a separation between privileged an unprivileged addresses in the BTB. This could be achieved by offering a dedicated flush operation, however this requires changes to the hardware. Alternatively, this flush operation can emulated in software, if the hashing algorithm used for indexing the BTB has been reverse engineered. We implemented this approach against the BTB attack by calling a function which performs a series of jump instructions along with our page tables switching routine and were unable to recover the correct randomization offset through the BTB attack in our tests.

10 THE GROWING PROBLEM OF SOFTWARE-EXPLOITABLE HARDWARE BUGS.

Lastly, we look towards the future of software-exploitable hardware bugs by reviewing micro-architectural security from a design perspective. In particular, we categorize common features of different hardware vulnerabilities and how they affect security. We then discuss quality assurance techniques that are traditionally employed by vendors to assess the security of large hardware designs and illustrate how such vulnerabilities may slip through the net to be exploited from software using the popular RISC-V architecture as a test-bed [7].

10.1 HARDWARE VULNERABILITIES FROM A SOFTWARE PERSPECTIVE

Until recently, hardware and software security research were mostly conducted separately due to the inherently different threat models traditionally associated with these distinct fields—hardware attacks assumed physical presence while software attacks could be launched remotely, e.g., by connecting to a vulnerable service running on any machine on the internet.

However, a number of recently discovered, new attack classes challenge this assumption: for instance, (i) Rowhammer attacks [72–76, 134, 135] provoke electromagnetic coupling effects to induce hardware faults in DRAM modules, (ii) speculative execution attacks [66, 67, 141] deliberately make transient micro-architectural states visible through cache sidechannels, and (iii) remote-fault injection attacks such as CLKScrew [65] are able to cause power glitches from software.

These new attack classes have serious repercussions for production systems and even enable attackers to break dedicated and deployed hardware-security extensions such as Intel SGX [67] and ARM TrustZone [65]. They also affect a wide range of targets with attacks against low-end devices, mobile phones, and laptops, all the way to high-end server machines.

10.1.1 *The Origin of the Hardware Bug*

What these new types of vulnerabilities have in common is that they expose structural design or implementation weaknesses inherently built into the hardware components to unprivileged software, and hence, attackers achieve effects that bear resemblance to those of physical attacks while operating completely from software. Additionally, in stark contrast to software-based exploits these attacks do not rely on bugs in the software code, but focus on the hardware implementation instead.

Interestingly, sufficiently complex hardware logic is indeed implemented through human-readable hardware-description languages (HDL) that are in many respects very similar to software programming languages: register-transfer level (RTL) code specifies variables, conditions, branches, and even loop constructs that are then compiled into a lower-level representation suitable for hardware synthesis. The higher-level RTL statements can also be organized in sub-procedures and grouped into modules, which can be nested, connected, and copied.

Consequently, RTL implementations of modern hardware can grow quite complex with all the different features that are incorporated into and supported by recent chips. Indeed, real-world system-on-chips (SoCs) can easily outgrow a million lines of RTL code [268] and like software vulnerabilities, bugs in RTL code can be introduced at any point due to human error. However, while software bugs typically result in termination of the affected program through various fallback routines which protect software running on the platform, such precautions do not exists in the case of a hardware bug.

10.1.2 *Quality Assurance Techniques for Hardware*

Since today practically all market-relevant SoCs are based on highly proprietary code bases, their implementation is not available for research or inspection. Consequently, security researchers have to resort to black-box testing and reverse engineering, which are time consuming, error-prone, and costly techniques. Unfortunately, this means that many vulnerabilities can only be identified by the manufacturer in a meaningful way.

For this reason, semiconductor manufacturers have established a standardized process for the security development of hardware components and that matches their development schedules and production cycles [269–272]. Usually, this process starts with the outline of a product sketch based on market research and competitor analysis. This draft is then refined using cycle-accurate simulations and tweaked or optimized with respect to key metrics such as performance and power consumption. At that point, the targeted use cases are typically narrowed down sufficiently to allow for formulation of the security objectives.

The architecture draft forms the basis for the micro-architectural design and implementation, which undergoes pre-silicon functional validation and security testing using various industry-standard tools such as Incisive [273], Solidify [274], Questa [275], OneSpin [276], and JasperGold [277]. Most of these tools were originally developed towards functional verification, and hence, security-relevant features are often not the focus.

Especially in view of the widespread availability of tool-based approaches for the security analysis of software code [180, 181, 196, 236, 238, 239, 278] the current state-of-the-art in RTL security verification is still lacking many features [269, 279, 280]. Although historically speaking hardware bugs are scarce [281, 282] compared to the large number of software bugs, the ever growing complexity of hardware implementations is starting to have a severe impact on the security of real-world products [62, 64–66, 141, 283–292].

We can already see that real-world hardware designs raise a number of difficult challenges to the available validation techniques: many modern architectures leverage complex and subtle interactions between different components and heavily optimize for run-time performance to outsmart rivaling designs. Since existing tools require costly analysis techniques to make accurate assessments, they do not scale with the increasing size and complexity of real-world chips.

Currently, these approaches rely on human expertise to select and guide high-priority analysis towards sensitive areas in the implementation. This largely manual task is both tedious and error-prone since modern processors often exhibit surprising side effects at run time.

10.2 ASSESSING THE STATE-OF-THE-ART

In joint work [7] with colleagues as well as industry collaborators from Intel we recently started a systematic effort of assessing existing bug detection techniques to analyze the effectiveness of the deployed quality assurance measures. To qualitatively evaluate how well industry-standard tools and state-of-the-art methods can detect these bugs, we conducted a series of case studies targeted towards security auditing of RTL code.

Since the majority of hardware designs and implementations used in practice are not available for open inspection, we leveraged the publicly available RISC-V architecture [293] in these tests. We analyzed common pitfalls for popular detection techniques using deliberately planted hardware bugs for RISC-V-based processor implementations. We then qualitatively evaluated how well these inserted vulnerabilities could be detected in practice using industry-standard tools and dynamic and static analysis by expert teams.

Many vulnerabilities in RTL code are introduced due to simple errors in regularly used code constructs, similar to software bugs, such as erroneous assignments associated with variable, registers, and parameter names being swapped or misspelled, timing errors resulting from concurrency or clock-domain crossings, flawed case statements such as incorrect or incomplete selection criteria, or wrong behavior within a case, faulty branching like false boolean conditions in if-else statements or incorrect behavior described within either branch, or simply poorly specified or unspecified behavior.

Crucially, even minor errors can result in severe vulnerabilities that are also hard to detect during testing or verification. While some parts of modern hardware are in fact configurable and can be patched, e.g., through micro-code updates, most components are hard-wired in the form of integrated circuits on the chip. As a result, hardware vulnerabilities can generally not be patched, and hence, bugs in RTL code represent a difficult and costly problem in practice.

10.3 EXPLOITING HARDWARE BUGS FROM SOFTWARE

For our in-depth studies and definition of HardFails, we investigate specific micro-architectural details at the RTL level. As all vendors keep their proprietary industry designs and implementations inaccessible, we use the popular open-source RISC-V architecture as a baseline [293].

RISC-V supports a wide range of possible configurations with many standard features that are also available in modern processor designs, such as privilege level separation, virtual memory, and multi-threading, as well as more advanced features such as configurable branch prediction and non-blocking data caches [294], and out-of-order execution [295], making the platform a suitable target for our study.

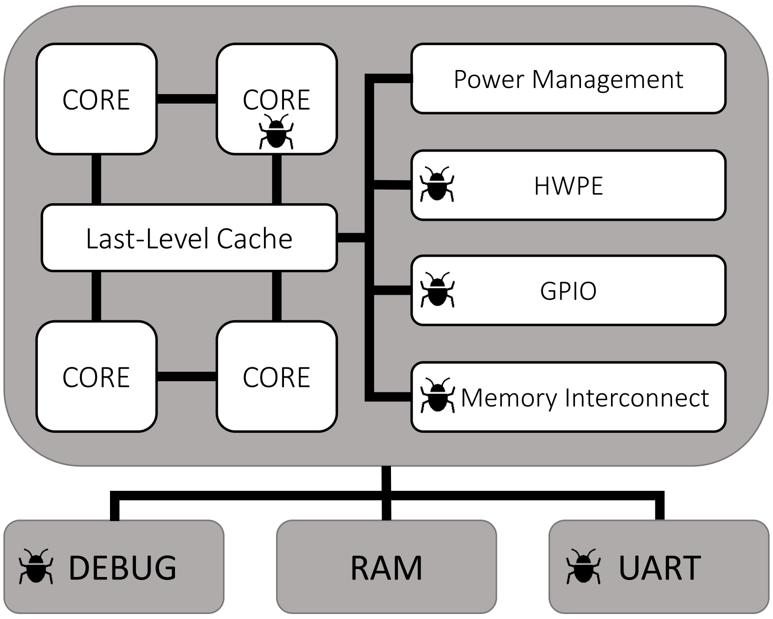


Figure 10.1: So far, practically all hardware components of modern system-on-chips have been demonstrated to contain software-exploitable vulnerabilities. In joint work [7] with colleagues as well as industry collaborators we attempted to qualitatively evaluate existing auditing tools for hardware designs based on more than 30 bugs injected in different parts of the open-source RTL implementation of an open-source RISC-V-based hardware test-bed.

10.3.1 *Threat Model and Assumptions*

RISC-V RTL is freely available and open to inspection and modification. We note that while this is not necessarily the case for industry-leading chip designs, an adversary might be able to reverse engineer parts of the chip. Although a highly cumbersome and difficult task in practice, this possibility cannot be excluded in principle. Hence, we allow an adversary to inspect the RTL code in our model.

In particular, we make the following assumptions to evaluate both existing verification approaches and possible attacks:

HARDWARE VULNERABILITY The attacker has knowledge of a vulnerability in the hardware implementation of the SoC (i.e., at the RTL level) and can trigger the bug from software.

USER ACCESS The attacker has complete control over a user-space process, i.e., can issue unprivileged instructions and system calls. For RISC-V, this means the attacker can execute any instruction in the basic instruction set.

SECURE SOFTWARE Software vulnerabilities and resulting attacks such as code-reuse [10, 22, 115, 117, 118] and data-only attacks [2, 13, 68, 208] against the software stack are orthogonal to the problem of cross-layer bugs, which leverage *hardware vulnerabilities* from the software layer.

Under our model, all platform software could be protected by defenses such as control-flow integrity [296] and data-flow integrity [297], or be formally verified. The goal of an adversary under this model is to leverage the vulnerability on the chip to provoke unintended functionality, e.g., access to protected memory locations, code execution with elevated privileges, breaking the isolation of other processes running on the platform, or permanently denying services at the hardware level. RTL bugs in certain modules of the chip might only be exploitable with physical access to the victim device, for instance, bugs in the implementation of debugging interfaces.

However, software-exploitable vulnerabilities can also be exploited completely remotely by software means, and hence, have a higher impact in practice. For this reason, we focus on software-exploitable RTL vulnerabilities. We also note that an adversary with unprivileged access is a realistic model for real-world SoCs: many platforms provide services to other devices over the local network, or even over the internet. Consequently, the attacker can obtain some limited software access to the platform already, e.g., through a webserver or an RPC interface.

The goal of the various verification approaches in this setting is to catch all of the bugs that would be exploitable by such an adversary before the chip design enters the production phase.

10.3.2 *Attack Details*

We show how selected hardware bugs from Hack@DAC 2018 hardware security competition [7] can be used to craft a real-world exploit. Such exploits allow unprivileged attackers to undermine the entire system by escalating privileges in an entirely remote setting. The attack is depicted in Figure 10.2 in which we assume the memory bus decoder unit (unit of the memory interconnect) to have a bug, which causes errors to be ignored under certain conditions (i.e., bug #7 [7]).

This RTL vulnerability manifests in the hardware behaving in the following way. When an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This represents a severe vulnerability, as it allows erroneous memory accesses to slip through hardware checks at run time. Despite this fact, we only managed to detect this vulnerability after significant efforts using the available tools based on prior knowledge of the exact location of the vulnerability. Additionally, the tool-based (but interactive) verification procedure represented a significant costly time investment.

Since vulnerabilities are usually not known a priori in practice, this would even be more difficult. Therefore, it is easily conceivable and realistic to assume that such a vulnerability could slip through verification and evade detection in larger real-world SoCs.

Armed with the knowledge about this vulnerability in a real-world processor, an adversary could now force memory access errors to slip through the checks as we describe in the following. In the first step ①, the attacker generates a user program (Task A) that registers a dummy signal handler for the segmentation fault (SIGSEGV) access violation. This first program then executes a loop with ② a faulting memory access to an invalid memory address (e.g., $LWx5, 0x0$).

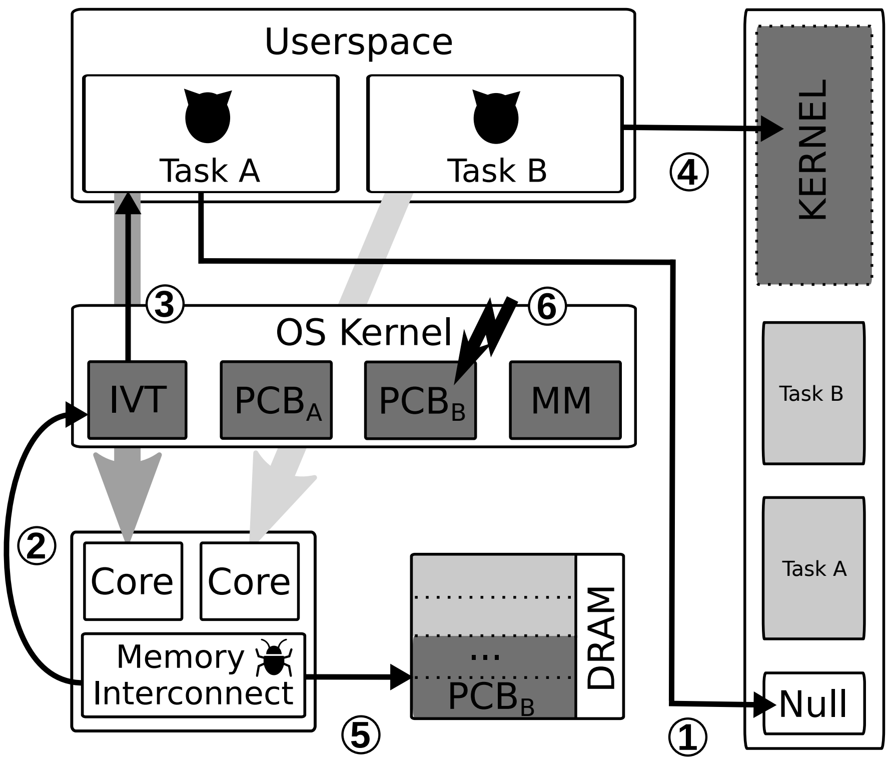


Figure 10.2: Our attack exploits a bug in the implementation of the memory bus of the PULPissimo SoC: by ② *spamming* the bus with invalid transactions, and ③ handling interrupts due to memory access violations to avoid termination of the malicious program, an adversary can make ④ malicious write requests be set to `operational`.

This will generate an error in the memory subsystem of the processor and issue an invalid memory access interrupt (i.e., `0x0000008C`) to the processor. The processor raises this interrupt to the running software (in this case the OS), using the pre-configured interrupt handler routines in software. The interrupt handler in the OS will then forward this as a signal to the faulting task ③, which keeps looping and continuously generating invalid accesses. Meanwhile, the attacker launches a separate Task B, which will then issue single memory access ④ to a privileged memory location (e.g., $LWx6, 0xf77c3000$).

In that situation, multiple outstanding memory transactions will be generated on the memory bus; all but one of which the address decoder will signal an error. An invalid memory access will always proceed the single access of the second task. Due to the bug in the memory bus address decoder, ⑤ the malicious memory access will become `operational` instead of triggering an error.

As a result, the attacker can issue read and write instructions to arbitrary privileged (and unprivileged) memory by forcing the malicious, illegal access with preceding faulty access. Using this technique the attacker can eventually leverage this read-write primitive, e.g., ⑥ to escalate privileges by writing the process control block (PCB_B) for his task to elevate the corresponding process to root.

Since the underlying bug leaves an adversary with the possibility of gaining access to a root process the exploitation strategy described above, the attacker is able to gain control over the entire platform and potentially compromise other processes running on the system or even the OS.

11 RELATED WORK

In this chapter, we review Rowhammer-based attack and defenses proposed by the literature, as well as side-channel attacks against the OS, and finally compare and classify attacks that exploit micro-architectural vulnerabilities in processor hardware designs and implementations presented recently.

11.1 ROWHAMMER

In this section, we provide an overview of existing rowhammer attack techniques, their evolution, and proposed defenses. Thereafter, we discuss the shortcomings of existing work on mitigating rowhammer attacks and compare them to our software-based defense.

11.1.1 *Attacks*

Kim et al. [63] were the first to conduct experiments and analyze the effect of bit flipping due to repeated memory reads. They found that this vulnerability can be exploited on Intel and AMD-based systems. Their results show that over 85% of the analyzed DRAM modules are vulnerable. The authors highlight the impact on memory isolation, but they do not provide any practical attack. Seaborn and Dullien [72] published the first practical rowhammer-based privilege-escalation attacks using the x86 `clflush` instruction. In their first attack, they use rowhammer to escape the Native Client (NaCl) [298] sandbox. NaCl aims to safely execute native applications by 3rd-party developers in the browser. Using rowhammer malicious developers can escape the sandbox, and achieve remote code execution on the target system. With their second attack Seaborn and Dullien utilize rowhammer to compromise the kernel from an unprivileged user-mode application. Combined with the first attack, the attacker can remotely compromise the kernel without exploiting any software vulnerabilities. To compromise the kernel, the attacker first fills the physical memory with page-table entries by allocating a large amount of memory. Next, the attacker uses rowhammer to flip a bit in memory. Since the physical memory is filled with page-table entries, there is a high probability that an individual page-table entry is modified by the bit flip in a way that enables the attacker to access other page-table entries, modify arbitrary (kernel) memory, and eventually completely compromise the system. Qiao and Seaborn [73] implemented a rowhammer attack with the x86 `movnti` instruction. Since the `memcpy` function of `libc` – which is linked to nearly all C programs – utilizes the `movnti` instruction, the attacker can exploit the rowhammer bug with code-reuse attack techniques [10]. Hence, the attacker is not required to inject her own code but can reuse existing code to conduct the attack. Aweke et al. [253] showed how to execute the rowhammer attack without using any special instruction (e.g., `clflush` and `movnti`). The authors use a specific memory-access pattern that forces the

CPU to evict certain cache sets in a fast and reliable way. They also concluded that a higher refresh rate for the memory would not stop rowhammer attacks. Gruss et al. [75] demonstrated that rowhammer can be launched from JavaScript. Specifically, they were able to launch an attack against the page tables in a recent Firefox version. Similar to Seaborn and Dullien's exploit this attack is mitigated by CATT. Later, Bosman et al. [135] extended this work by exploiting the memory deduplication feature of Windows 10 to create counterfeit JavaScript objects, and corrupting these objects through rowhammer to gain arbitrary read/write access within the browser. In their follow-up work, Razavi et al. [134] applied the same attack technique to compromise cryptographic (private) keys of co-located virtual machines. Concurrently, Xiao et al. [76] presented another cross virtual machine attack where they use rowhammer to manipulate page-table entries of Xen. Further, they presented a methodology to automatically reverse engineer the relationship between physical addresses and rows and banks. Independently, Pessl et al. [243] also presented a methodology to reverse engineer this relationship. Based on their findings, they demonstrated cross-CPU rowhammer attacks, and practical attacks on DDR4. Van der Veen et al. [74] recently demonstrated how to adapt the rowhammer exploit to escalate privileges in Android on smartphones. Since the authors use the same exploitation strategy of Seaborn and Dullien, CATT can successfully prevent this privilege escalation attack. While the authors conclude that it is challenging to mitigate rowhammer in software, we present a viable implementation that can mitigate practical user-land privilege escalation rowhammer attacks.

Note that all these attacks require memory belonging to a higher-privileged domain (e.g., kernel) to be physically co-located to memory that is under the attacker's control. Since our defense prevents direct co-location, we mitigate these rowhammer attacks.

11.1.2 *Defenses*

Kim et al. [63] present a number of possible mitigation strategies. Most of their solutions involve changes to the hardware, i.e., improved chips, refreshing rows more frequently, or error-correcting code memory. However, these solutions are not very practical: the production of improved chips requires an improved design, and a new manufacturing process which would be costly, and hence, is unlikely to be implemented. The idea behind refreshing the rows more frequently (every 32ms instead of 64ms) is that the attacker needs to hammer rows many times to destabilize an adjacent memory cell which eventually causes the bit flip. Hence, refreshing (stabilizing) rows more frequently could prevent attacks because the attacker would not have enough time to destabilize individual memory cells. Nevertheless, Aweke et al. [253] were able to conduct a rowhammer attack within 32ms. Therefore, a higher refresh rate alone cannot be considered as an effective countermeasure against rowhammer. Error-correcting code (ECC) memory is able to detect and correct single-bit errors. As observed by Kim et al. [63] rowhammer can induce multiple bit flips which cannot be detected by ECC memory. Further, ECC memory has an additional space overhead of around 12% and is more expensive than usual DRAM, therefore it is rarely used.

Kim et al. [63] suggest to use probabilistic adjacent row activation (PARA) to mitigate rowhammer attacks. As the name suggests, reading from a row will trigger an activation of adjacent rows with a low probability. During the attack, the malicious rows are activated many times. Hence, with high probability the victim row gets refreshed (stabilized) during the attack. The main advantage of this approach is its low performance overhead. However, it requires changes to the memory controller. Thus, PARA is not suited to protect legacy systems.

To the best of our knowledge Aweke et al. [253] proposed the only other software-based mitigation against rowhammer. Their mitigation, coined ANVIL, uses performance counters to detect high cache-eviction rates which serves as an indicator of rowhammer attacks [253]. However, this defense strategy has three disadvantages: (1) it requires the CPU to feature performance counters. In contrast, our defense does not rely on any special hardware features. (2) ANVIL's worst case run-time overhead for SPEC CPU2006 is 8%, whereas our worst case overhead is 0.29%. (3) ANVIL is a heuristic-based approach. Hence, it naturally suffers from false positives (although the FP rate is below 1% on average). In contrast, we provide a deterministic approach that is guaranteed to stop rowhammer-based kernel-privilege escalation attacks.

11.2 SIDE-CHANNEL ATTACKS AGAINST THE OS

In this section we provide an overview of the related work regarding side-channel attacks, focusing towards attacks against the operating system. We further discuss proposed software and hardware mitigations against side-channel attacks.

11.2.1 *Paging-based Side-channel Attacks on KASLR*

All modern operating systems leverage kernel-space randomization by means of kernel code randomization (KASLR) [162–164]. However, kernel-space randomization has been shown to be vulnerable to a variety of side-channel attacks. These attacks leverage micro-architectural implementation details of the underlying hardware. More specifically, modern processors share virtual memory resources between privileged and unprivileged execution modes.

In the following we briefly describe recent paging-based side-channel attacks that aim to disclose the kASLR randomization secret. All these attacks exploit the fact that the TLB is shared between user applications and the kernel. As a consequence, the TLB will contain page-table entries of the kernel after switching the execution from kernel to a user mode application. Henceforth, the attacker uses special instructions (depending on the concrete side-channel attack implementation) to access kernel addresses. Since the attacker executes the attack with user privileges, the access will be aborted. However, the time difference between access attempt and abort depends on whether the guessed address is cached in the TLB or not. Further, the attacker can also measure the timing between existing (traversal of the page-table hierarchy) and not existing mappings (immediate abort). These timing differences can be exploited by the attacker as a side channel to disclose the randomization secret as shown recently [80, 82, 83, 133].

PAGE FAULT HANDLER (PFH) Hund, et al. [80] published the first side-channel attack to defeat KASLR. They trigger a page fault in the kernel from a user process by accessing an address in kernel space. Although this unprivileged access is correctly denied by the page fault handler, the TLBs are queried during processing of the memory request. They show that the timing difference between exceptions for unmapped and mapped pages can be exploited to disclose the random offset.

INTEL'S TSX Transactional memory extensions introduced by Intel encapsulate a series of memory accesses to provide enhanced safety guarantees, such as rollbacks. While potentially interesting for the implementation of database systems, erroneous accesses within a transaction are not reported to the operating system. More specifically, if the MMU detects an access violation, the transaction is aborted silently. However, an adversary can measure the timing difference between two aborted transactions to identify privileged addresses, which are cached in TLBs. This enables the attacker to significantly improve over the original page fault timing side-channel attack [82, 133]. The reason is that the page fault handler of the OS is never invoked, significantly reducing the noise in the timing signal.

PREFETCH INSTRUCTION Furthermore, even individual instructions may leak timing information and can be exploited [83]. More specifically, the execution of the `prefetch` instruction of recent Intel processors exhibits a timing difference, which depends directly on the state of the TLBs. As in the case of the side-channel attack exploiting transactional memory, this unprivileged instruction can be used to access privileged addresses, but will fail without invoking the page fault handler of the OS. Its execution time differs for cached kernel addresses. This yields another stealthy side channel that leaks the randomization secret.

11.2.2 *Software Mitigations*

SEPARATING ADDRESS SPACES Unmapping the kernel page tables during userland execution is a natural way of separating their respective address spaces, as suggested in [82, 83]. However, Jang et al. [82] considered the approach impractical, due to the expected performance degradation. Gruss et al. [83] estimated the performance impact of reloading the entire page table hierarchy up to 5%, by reloading the top level of the page table hierarchy (via the `CR3` register) during a context switch. They subsequently implemented this approach independently and in parallel to our work LAZARUS [84]. Reloading the top level of the page tables results in a higher performance overhead, because it requires the processor to flush all of the cached entries. Full address-space separation has also been implemented by Apple for their iOS platform [92]. Because the ARM platform supports multiple sets of page table hierarchies, the implementation is straightforward on mobile devices. For the first time we provide an improved and highly practical method of implementing address space separation on the x86 platform.

INCREASING KASLR ENTROPY Some of the presented side-channel attacks benefit from the fact that the KASLR implementation in the Linux kernel suffers from a relatively low entropy [80, 81]. Thus, increasing the amount of entropy represent a way of mitigating those attacks in practice. While this approach was suggested by Hund et al. [80] and Evtyushkin et al. [81], it does not eliminate the side channel. Additionally, the mitigating effect is limited to attacks which exploit low entropy randomization. In contrast, LAZARUS mitigates all previously presented paging side-channel attacks.

MODIFYING THE PAGE FAULT HANDLER Hund et al. [80] exploited the timing difference through invoking the page fault handler. They suggested to enforce its execution time to an equal timing for all kernel addresses through software. However, this approach is ineffective against attacks which do not invoke the kernel [82, 83]. Our mitigation reorganizes the cache layout in software to successfully stop the attacks, that exploit hardware features to leak side channel information, even for attacks that do not rely on the execution time of any software.

11.2.3 *Hardware Mitigations*

PRIVILEGE LEVEL ISOLATION IN THE CACHES Eliminating the paging side channel is also possible by modifying the underlying hardware cache implementation. This was first noted by Hund et al. [80]. However, modern architectures organize caches to be optimized for performance. Additionally, changes to the hardware are very costly, and it takes many years to widely deploy these new systems. Hence, it is unlikely that such a change will be implemented, and even if it is, existing production systems will remain vulnerable for a long time. Our software-only mitigation can be deployed instantly by patching the kernel.

DISABLING DETAILED TIMING FOR UNPRIVILEGED USERS All previously presented paging side-channel attacks rely on detailed timing functionality, which is provided to unprivileged users by default. For this reason, Hund et al. [80] suggested to disable the `rdtsc` instruction for user mode processes. While this can be done from software, it effectively changes the ABI of the machine. Since modern platforms offer support for a large body of legacy software, implementing such a change would introduce problems for many real-world user applications. As we demonstrate in our extensive evaluation, LAZARUS is transparent to user-level programs and does not disrupt the usual workflow of legacy software.

11.3 RECENT HARDWARE EXPLOITS

As outlined in Section 2.5.3, many recent attacks combine different problems, e.g., inherent cache leakage and hardware implementation errors at the chip level. Here, we explain and classify these previously presented remote hardware exploits (see Table 11.1). Since we already discussed the related work regarding Rowhammer in Section 11.1, we focus on CPU-based attacks in this section.

Yarom et al. demonstrate that software-visible side channels can exist even below cache-line granularity in their CacheBleed [283] attack—undermining a core assumption of prior defenses such as scatter-gather [299].

The recent TLBleed [284] attack demonstrates that current TLB implementations can be abused to break state-of-the-art cache side-channel protections. As outlined

Attack	**Priv. Level**	**Memory Corruption**	**Inf. Discl.**	**Cache**	**Firmware**	**Memory Interconnect**	**CPU**
Cachebleed [283]	user	✗	✓	✓	✗	✗	✗
TLBleed [284]	user	✗	✓	✓	✗	✗	✗
BranchScope [285]	user	✗	✓	✗	✗	✗	✓
Spectre [141]	user	✗	✓	✓	✗	✗	✓
Meltdown [66]	user	✗	✓	✓	✗	✗	✓
MemJam [62]	OS	✗	✓	✗	✗	✓	✗
CLKScrew [65]	OS	✓	✓	✗	✓	✗	✓
Foreshadow [67]	OS	✓	✓	✓	✓	✗	✓

Table 11.1: **Classification of existing attacks:** when reviewing recent hardware-oriented exploits that are possible from software, we observe that many of them exploit hardware vulnerabilities that affect one of the components we investigated in our recent case studies (cf., Chapter 10).

in Chapter 2, TLBs are typically highly interconnected with complex processor modules such as the cache controller and memory management unit, making vulnerabilities therein very hard to detect through automated verification or manual inspection.

BranchScope [285] extracts information through the directional branch predictor which is a hardware resource used implicitly by physical cores but leaks information to software in the form of a timing channel. The authors also propose alternative design strategies for the Branch-Target Buffers, such as randomizing the patter-history table.

As explained in Section 2.5.2, modern processors heavily optimize performance, e.g., through increasing instruction-level parallelism and a common set of techniques fall under the category of *Out-of-Order* (OoO) execution. For instance, the architecture may maximize utilization of idle execution units on a physical core by speculatively scheduling pipelined instructions ahead of time. While erroneous or unauthorized memory accesses can be forced from malicious software, they are correctly rolled-back (i.e., they are not *committed* to the final instruction stream). However, these unauthorized accesses during speculation may affect cache states, and hence, can remain visible to software through side-channel attacks on the caches in a second step. This was shown to be exploitable in the Meltdown attack [66], which targeted vulnerable out-of-order implementations allowing software to bypass the usually enforced memory access restrictions in the OS. As noted by the authors our side-channel defense for the kernel, LAZARUS [6], successfully stops the attack [66].

Shortly thereafter, Van Bulck et al. demonstrated that a similar technique, dubbed Foreshadow [67], can be applied to completely compromise the security goals of Intel's Software-Guard Extensions (SGX), a popular hardware security extension.

Unfortunately, by stealing the attestation keys used by the SGX firmware an adversary is able to craft malicious enclave executions, thereby allowing memory corruption.

Moreover, Kocher et al. [141] showed that speculative execution can be exploited in user-space in a related attack, as a wide range of instruction sequences can be constructed and forced to continue execution under the vulnerable out-of-order implementations on the affected processors.

MemJam [62] exploits false read-after-write dependencies of subsequent memory accesses within the CPU to maliciously slow down victim accesses. Similar to Cachebleed, this breaks any constant-time implementations that rely on cache-line granularity.

CLKScrew [65] abuses low-level power-management functionality that is exposed to software on many ARM-based devices, e.g., to optimize battery life. Tang et al. demonstrated that this can be exploited by malicious users to induce faults and glitches dynamically at runtime in the processor. By maliciously tweaking clock frequency and voltage parameters, they were able to make the execution of individual instructions fail with a high probability. The authors constructed an end-to-end attack that works completely from software and breaks the TrustZone isolation boundary, e.g., to access secure-world memory from the normal world. We categorize CLKScrew as a firmware issue since it directly exposes power-management functionality to attacker-controlled software. Currently, CLKScrew is one of the only two known processor-based hardware vulnerabilities that allows an software adversary to corrupt memory.

Part IV

DISCUSSION AND CONCLUSION

12 DISSERTATION SUMMARY

Software and hardware vulnerabilities pose significant challenges to the security of computer systems. While attacks based on memory-corruption vulnerabilities in widely deployed software have been researched extensively in the past, hardware-based attack represent a recently emerging attack paradigm. The goal of this dissertation is two-fold: we explore the limitations of existing defenses against memory-corruption exploits and possible mitigations in light of upcoming hardware-based attacks. For this, we focus on the operating system context, since it typically implements and enforces security policies for a computer system at run time.

In Part II, we demonstrated data-only attacks that completely bypass state-of-the-art code-reuse defenses, such as control-flow integrity. To this end, we show that enforcing CFI securely in complex application settings like modern browsers poses a series of challenges due to dynamic code generation. In particular, many applications that require frequent user interaction utilize scripting language environments. These scripting environments often leverage just-in-time compilers to enhance run-time performance and user experience. However, we show that an adversary can modify the intermediate representation deployed by these compilers to generate malicious native code on-the-fly in a data-only attack. Since our attack does not tamper with any code pointers, it cannot be prevented by state-of-the-art code-reuse defenses and works despite CFI being deployed. We then turn towards existing code-reuse defenses in the kernel context and show that attackers can completely deactivate *any* memory protection by modifying the page tables. Since page tables are data object, this is again possible without modifying code pointers, and hence, cannot be prevented by CFI. We also design and implement a randomization-based defense to protect the page tables against such data-only attacks. As modern OS kernels typically consists of a huge and highly complex code base, that is difficult to audit by manual review, we then introduce the first data-flow analysis framework for kernel code. Our compiler-based framework is able to automatically uncover memory-corruption vulnerabilities present in complex real-world code.

In Part III, we turned towards software-exploitable hardware vulnerabilities by presenting novel Rowhammer attacks against DRAM-based physically-unclonable functions. Our results show that DRAM may not be a suitable candidate for the construction of security-sensitive applications. Further, several Rowhammer-based exploits were recently presented that specifically target OS memory to compromise computer platforms remotely. For this reason, we design and implement the first practical software defense against Rowhammer attacks that target kernel memory. As we are able to show, our scheme successfully mitigates real-world Rowhammer attacks efficiently. Moreover, a number of attacks with a high impact on operating system security recently demonstrated that micro-architectural side-channel attacks represent a realistic threat. Hence, we introduce a general side-channel defense for operating system kernels, to harden OS software against such attacks in practice. Finally, we investigate possible root causes of hardware vulnerabilities by assessing the effectiveness of state-of-the-art hardware verification tools. In particular, we found that specific vulnerability classes can slip through existing quality assurance tests in practice due to fundamental limitations and increasingly severe scalability issues.

13 FUTURE WORK

While memory-corruption-based attacks have been known and subject of intense research for more than three decades, the underlying problems are not fully solved yet and many challenges still remain. Among the many run-time exploit paradigms, code-reuse attacks have been the focus of attention for a long time. Consequently, a large number of code-reuse defenses have been proposed in the related work and many mitigations are deployed and readily available to harden production systems through various software or compiler-based schemes, even including occasional hardware support. However, as we demonstrate data-only attacks still pose a significant threat to the security of application and kernel-level software. So far, general defenses are lacking and the available solutions are either application-specific or incur extreme performance overheads, rendering them impractical. Hence, additional research is required to comprehensively defend software against data-oriented exploits at run time.

Another promising line of research is software verification to eliminate the underlying root causes of memory corruption in the code, e.g., by means of dynamic testing or static analysis. While dynamic approaches, such as random testing (often called *fuzzing*) or sanitization represent important tools that are also increasingly used in practice they are typically not comprehensive and prohibitively expensive in terms of resources. On the other hand, static analysis approaches promise formal security guarantees but lack the required scalability and expressiveness to be widely applicable. Hence, researching novel approaches and practical techniques to automatically check real-world code for potential memory-corruption vulnerabilities at a large scale appears to be a highly relevant area for future directions.

Even if system software is assumed to be completely secure, hardware-oriented exploits are becoming increasingly relevant as they currently represent an emerging attack vector. Unfortunately, recent research demonstrated that a growing number of hardware vulnerabilities can be exploited completely remotely, and hence, software adversaries no longer require software vulnerabilities to launch successful attacks. As hardware grows more complex the problems of design and implementation techniques for hardware are also prone to the problem of scaling quality assurance measures to real-world projects which has proven uniquely challenging in the software realm. Moreover, many hardware platforms do not function correctly without a large amount of firmware which is even exposed to the exact same attacks as traditional system software.

While methods for systematic security analysis and verification that were developed in the software domain could in principle be applied to identify vulnerabilities at the hardware level, this field is only slowly gaining traction as practically all hardware designs available today are strictly and entirely proprietary. Consequently, the approaches used by the industry are significantly lagging behind the state-of-the-art in software verification and testing, which is largely driven by the open-source community and the possibility of freely inspecting, testing, and analyzing real-world implementations. Finally, both fields historically represent completely disconnected areas of research and increasing cross-communication, transparency, and mutual exchange of ideas could provide beneficial insights.

Part V

APPENDIX

A ABOUT THE AUTHOR

David Gens is a research assistant at the Technische Universität Darmstadt and the Intel Collaborative Research Institute (ICRI-CARS), Germany. His research has concentrated on operating system security in the context of memory-corruption exploits and mitigation of emerging threats such as sidechannel attacks and remote-fault injection attacks from an OS perspective.

SHORT BIO

- M.Sc. in Computer Science, Technische Universität Darmstadt, 2016
- B.Sc. in Computer Science, Hochschule RheinMain Wiesbaden, 2013

AWARDS AND NOMINATIONS

- Google Best Paper Award at the MobiSys PhD Forum 2018
- Travel Scholarship from ACM SIGDA to attend DAC 2018
- Student PC Member IEEE Security and Privacy 2018
- Top 10 Finalist at CSAW Applied Research Competition 2017

PEER-REVIEWED PUBLICATIONS

FastKitten: Practical Smart Contracts on Bitcoin. Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostakova, Patrick Jauernig, Sebastian Faust, Ahmad-Reza Sadeghi. In *Proceedings of the 28th USENIX Security Symposium*, August 2019.

SANCTUARY: ARMing TrustZone with User-space Enclaves. Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, Emmanuel Stapf. In *Proceedings of the 26th Annual Network & Distributed System Security Symposium (NDSS)*, February 2019.

It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs. Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. In *Proceedings of the 55th Design Automation Conference (DAC'18)*, June 2018.

OS-level Software & Hardware Attacks and Defenses. David Gens. In *Proceedings of the MobiSys PhD Forum*, June 2018.

K-Miner: Uncovering Memory Corruption in Linux. David Gens, Simon Schmitt, Lucas Davi, Ahmad-Reza Sadeghi. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium (NDSS)*, February 2018.

JITGuard: Hardening Just-in-time Compilers with SGX. Tommaso Frassetto, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, November 2017.

LAZARUS: Practical Side-channel Resilient Kernel-Space Randomization. David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, Ahmad-Reza Sadeghi. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, September 2017.

CATT: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. In *Proceedings of the 26th USENIX Security Symposium*, August 2017.

PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. In *Proceedings of the 24th Annual Network & Distributed System Security Symposium (NDSS)*, February 2017.

Privately computing set-union and set-intersection cardinality via bloom filters. Rolf Egert, Marc Fischlin, David Gens, Sven Jacob, Matthias Senker, Jörg Tillmans. In *Lecture Notes, Foo E., Stebila D. (eds) Australasian Conference on Information Security and Privacy (ACISP)*, June 2015.

TECHNICAL REPORTS

When a Patch is Not Enough — HardFails: Software-Exploitable Hardware Bugs. Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Karthik Kanuparthi, Hareesh Khattri, Jason M. Fung, Jeyavijayan Rajendran, Ahmad-Reza Sadeghi. Technical Report *arXiv:1812.00197*, December 2018.

FastKitten: Practical Smart Contracts on Bitcoin. Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostakova, Patrick Jauernig, Sebastian Faust, Ahmad-Reza Sadeghi. Technical Report.

Execution Integrity with In-Place Encryption. Dean Sullivan, Orlando Arias, David Gens, Lucas Davi, Ahmad-Reza Sadeghi, Yier Jin. Technical Report *arXiv:1703.02698*, March 2017.

CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. Technical Report *arXiv:1611.08396*, November 2016.

POSTERS

CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. In *Cyber Security Awareness Worldwide, Applied Research Competition (CSAW)*, August 2017.

OS-level Software & Hardware-based Attacks and Defenses. David Gens. In *Design Automation Conference PhD Forum*, June 2018.

B LIST OF FIGURES

Figure 2.1 Virtual memory represents a page-based memory-protection scheme that can be enforced in hardware on many platforms. 10

Figure 2.2 Today, many attacks involve multi-step exploits, in which the attacker successively migrates from a *remote* to a *local* adversary model to gain full control over the platform. Since the OS implements most of the security-relevant hardening techniques, attacks against kernel software in this second step are increasingly common in practice. 11

Figure 2.3 Programs can be represented as finite-state machines **(a)**, with vulnerable programs containing hidden states **(b)**, which only become reachable through undefined behavior to form *weird machines* [103, 104]. 14

Figure 2.4 The vulnerable program **(a)** exhibits undefined behavior for inputs exceeding the length of the buffer. An adversary can exploit this by providing malicious input **(b)** to bring the program into a weird state **(c)** that will spawn a terminal—a behavior that was never specified anywhere in the program code. 15

Figure 2.5 Modern computer platforms feature complex SoC architectures that combine multi-core designs with performance-critical elements *uncore* inside the CPU package. Components inside the CPU package share a large last-level cache and are tightly coupled through high-speed interconnect fabric on the die. Peripherals are connected to the processor package via the system bus. 18

Figure 2.6 Semiconductor manufacturers aim to enhance single-core performance by maximizing *instruction-level parallelism*. For this, chip designers leverage techniques such as super-scalar processing, simultaneous multi-threading, and out-of-order execution, which heavily increase implementation complexity of modern processors. 19

Figure 3.1 Our attack bypasses CFI enforcement by maliciously modifying the intermediate representation (IR), which resides in data memory and is never executed directly. Instead, it is used by the compiler to output machine code for an attack payload which the attacker can then invoke in a subsequent step. 27

Figure 3.2 Within the JIT engine the currently optimized instruction stream is represented in the form of a doubly linked list of data objects. By inserting crafted or modifying existing objects an adversary can inject arbitrary instructions into the generated machine code. Since DOJITA [1] only involves data accesses it cannot be prevented by means of code-reuse defenses and completely bypasses CFI for the browser. 28
Figure 4.1 Our page-table exploit enables code injection attacks in kernel mode despite CFI being deployed and active in the kernel. This motivated us to design an efficient randomization-based defense approach to complement kernel CFI implementations by protecting the page tables against such attacks. 33
Figure 4.2 Our design of PT-Rand involves several boot-time and run-time components to set up the randomized mapping for page tables in the kernel. 35
Figure 4.3 We modify the virtual memory location of page table pages by providing a randomization wrapper around the standard page allocator in the kernel. While randomized pages are removed from the one-to-one mapping, regular page allocation requests still fall within this region. 38
Figure 4.4 Overhead of page table randomization on SPEC CPU2006 43
Figure 5.1 Data-flow analyses utilize graphs to reason about program behavior at compile time. 48
Figure 5.2 Overview of the different components of K-Miner. 52
Figure 5.3 Overview of the K-Miner implementation: we conduct complex data-flow analysis of the Linux kernel in stages, re-using intermediate results. 55
Figure 5.4 Example of a Dangling Pointer vulnerability in a (simplified) system call definition. 56
Figure 5.5 Overview of results in the web user interface of K-Miner. 63
Figure 5.6 Individual bug report in the web user interface of K-Miner. 63
Figure 7.1 Organization of a DRAM module. 78
Figure 7.2 DRAM chips are structured hierarchically, with *banks* forming one of the intermediate components that can be replicated for individual chips. This diagram shows the organization inside a bank, which is only accessed via its *row buffer* from the outside. 79

Figure 7.3 What makes Rowhammer-based attacks [4, 72–76, 134, 135] so severe is the fact that only *adjacent* memory rows (i.e., above and below the attacked memory region) are accessed by the attacker. Due to the physical-to-DRAM mapping, these physically neighbouring rows can reside in completely separate parts of the address space. However, through electro-magnetic coupling effects an adversary can influence these otherwise inaccessible memory cells which can even result in flipped bits in the attacked row. 80
Figure 7.4 Workflow of ouf bootloader extensions 83
Figure 8.1 CATT constrains bit flips to the process' security domain. 87
Figure 9.1 The idea behind our side channel protection: An unprivileged user process (❶) can exploit the timing side channel for kernel addresses through shared cache access in the MMU paging caches (❷). Our defense mitigates this by enforcing (❸) a separation between different privilege levels for randomized addresses (❹). 104
Figure 9.2 Timing side-channel measurements we conducted before (blue) and after (red) we applied LAZARUS. 109
Figure 9.3 SPEC2006 Benchmark Results 111
Figure 9.4 LMBench3 Benchmark Results 112
Figure 9.5 Phoronix Benchmark Suite 112
Figure 10.1 So far, practically all hardware components of modern system-on-chips have been demonstrated to contain software-exploitable vulnerabilities. In joint work [7] with colleagues as well as industry collaborators we attempted to qualitatively evaluate existing auditing tools for hardware designs based on more than 30 bugs injected in different parts of the open-source RTL implementation of an open-source RISC-V-based hardware test-bed. 120
Figure 10.2 Our attack exploits a bug in the implementation of the memory bus of the PULPissimo SoC: by ② *spamming* the bus with invalid transactions, and ③ handling interrupts due to memory access violations to avoid termination of the malicious program, an adversary can make ④ malicious write requests be set to `operational`. 122

C

LIST OF TABLES

Table 4.1 Phoronix benchmark results. 44
Table 5.1 Overview of the specifications, resource requirements, and results for the different kernel versions and data-flow passes we used in our evaluation of K-Miner. 58
Table 5.2 Average and maximum memory usage of K-Miner 62
Table 6.1 Comparison of defense approaches ensuring the integrity of page tables. In contrast to the related work, PT-Rand [2] does not require hardware trust anchors such as Virtual-Machine Monitors (VMMs) or Trusted-Execution Environments (TEEs). Since our scheme additionally does not require costly policy checks at run time, but is based on leakage-resilient randomization instead it also incurs a low overhead of only 0.22% on average. 70
Table 8.1 Model numbers of the vulnerable systems used for our evaluation. 92
Table 8.2 Technical specifications of the vulnerable systems used for our evaluation. 94
Table 8.3 Results of our security evaluation. We executed the exploit continuously on each system for more than 48 hours and found that CATT mitigates real-world Rowhammer exploits against the kernel. 94
Table 8.4 Result for individual stress tests from the Linux Test Project. 97
Table 11.1 **Classification of existing attacks:** when reviewing recent hardware-oriented exploits that are possible from software, we observe that many of them exploit hardware vulnerabilities that affect one of the components we investigated in our recent case studies (cf., Chapter 10). 130

BIBLIOGRAPHY

[1] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. "JITGuard: Hardening Just-in-time Compilers with SGX." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM. 2017. URL: https://dl.acm.org/citation.cfm?id=3134037.

[2] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables." In: *24th Annual Network and Distributed System Security Symposium*. NDSS. 2017. URL: https://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/ndss2017_05B-4_Davi_paper.pdf.

[3] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. "K-Miner: Uncovering Memory Corruption in Linux." In: *25th Annual Network and Distributed System Security Symposium*. NDSS. 2018. URL: https://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-1_Gens_paper.pdf.

[4] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. "It's hammer time: how to attack (rowhammer-based) DRAM-PUFs." In: *Proceedings of the 55th Annual Design Automation Conference*. ACM. 2018, p. 65. URL: https://dl.acm.org/citation.cfm?id=3196065.

[5] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory." In: *Proceedings of the 26th USENIX Security Symposium (Security). Vancouver, BC, Canada*. USENIX Sec. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser.

[6] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. "LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization." In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2017, pp. 238–258. URL: https://link.springer.com/chapter/10.1007/978-3-319-66332-6_11.

[7] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason Fung, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. "When a Patch is Not Enough - HardFails: Software-Exploitable Hardware Bugs." In: *CoRR* (2018). URL: https://arxiv.org/abs/1812.00197.

[8] Donald C Latham. "Department of defense trusted computer system evaluation criteria." In: *Department of Defense* (1986).

[9] Aleph One. "Smashing the Stack for Fun and Profit." In: *Phrack Magazine* 49 (2000).

[10] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2007.

[11] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. "Memory errors: the past, the present, and the future." In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2012, pp. 86–106.

[12] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *34th IEEE Symposium on Security and Privacy*. S&P. 2013.

[13] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. "Non-Control-Data Attacks Are Realistic Threats." In: *14th USENIX Security Symposium*. USENIX Sec. 2005.

[14] Ralf Hund, Thorsten Holz, and Felix C Freiling. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." In: *USENIX Security Symposium*. 2009, pp. 383–398.

[15] Jidong Xiao, Hai Huang, and Haining Wang. "Kernel Data Attack Is a Realistic Security Threat." In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2015, pp. 135–154.

[16] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 414–425.

[17] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization." In: *34th IEEE Symposium on Security and Privacy*. S&P. 2013.

[18] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *36th IEEE Symposium on Security and Privacy*. S&P. 2015.

[19] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. "Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying." In: NDSS. 2017.

[20] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. "Authentication in distributed systems: Theory and practice." In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 265–310.

[21] Stanley R Ames Jr, Morrie Gasser, and Roger R Schell. "Security kernel design and implementation: An introduction." In: *IEEE computer* 16.7 (1983), pp. 14–22.

[22] Ralf Hund, Thorsten Holz, and Felix C. Freiling. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." In: *18th USENIX Security Symposium*. USENIX Sec. 2009.

[23] MWR Labs. *MWR Labs Pwn2Own 2013 Write-up - Kernel Exploit*. `http://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit`. 2013.

[24] Sebastien Renaud. *Technical Analysis of the Windows Win32K.sys Keyboard Layout Stuxnet Exploit*. `http://web.archive.org/web/20141015182927/http://www.vupen.com/blog/20101018.Stuxnet_Win32k_Windows_Kernel_0Day_Exploit_CVE-2010-2743.php`. 2010.

[25] Stefan Esser. "iOS Kernel Exploitation." In: *Blackhat Europe*. BH EU. 2011.

[26] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. "A survey of mobile malware in the wild." In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pp. 3–14.

[27] MITRE. *CVE-2016-5195*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195`. 2016.

[28] Dan Goodin. *Android phones rooted by "most serious" Linux escalation bug ever*. `https://arstechnica.com/information-technology/2016/10/android-phones-rooted-by-most-serious-linux-escalation-bug-ever`. 2016.

[29] Ian Beer. *iOS/MacOS kernel double free due to IOSurfaceRootUserClient not respecting MIG ownership rules*. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1417`. 2017.

[30] Dan Rosenberg. *Reflections on Trusting TrustZone*. `https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf`. 2014.

[31] Charles Holmes Josh Thomas Nathan Keltner. *Reflections on Trusting TrustZone*. `https://pacsec.jp/psj14/PSJ2014_Josh_PacSec2014-v1.pdf`. 2014.

[32] Gal Beniamini. *QSEE Privilege Escalation Vulnerabilitiy*. `http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html`. 2015.

[33] Sunshine LLC. *TRUSTNONE*. `http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf`. 2015.

[34] Di Shen. *Exploiting TrustZone on Android*. `https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf`. 2015.

[35] Gal Beniamini. *QSEE Privilege Escalation Vulnerabilitiy*. `http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html`. 2016.

[36] Lev Aronsky. "KNOXout - bypassing Samsung KNOX." In: (2016).

[37] Nick Stephens. *Behind the PWN of a TrustZone*. `https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose`. 2016.

[38] Tencent. *Defeating Samsung KNOX with Zero Privilege*. `https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf`. 2017.

[39] Project Zero. *Lifting the Hyper Visor*. https://googleprojectzero.blogspot.de/2017/02/lifting-hyper-visor-bypassing-samsungs.html. 2017.

[40] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2005.

[41] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications." In: *ACM Transactions on Information System Security* 13 (2009).

[42] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation." In: *Design Automation Conference*. DAC. 2014.

[43] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. "Fine-Grained Control-Flow Integrity for Kernel Software." In: *1st IEEE European Symposium on Security and Privacy*. Euro S&P. 2016.

[44] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." In: *19th Annual Network and Distributed System Security Symposium*. NDSS. 2012.

[45] J. Criswell, N. Dautenhahn, and V. Adve. "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels." In: *35th IEEE Symposium on Security and Privacy*. S&P. 2014.

[46] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity." In: *11th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2014.

[47] J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. "ILR: Where'd My Gadgets Go." In: *33rd IEEE Symposium on Security and Privacy*. S&P. 2012.

[48] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, George C. Necula, and Michael Vrable. "XFI: Software Guards for System Address Spaces." In: *7th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2006.

[49] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity." In: *7th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2006.

[50] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software." In: *22nd Annual Computer Security Applications Conference*. ACSAC. 2006.

[51] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization." In: *33rd IEEE Symposium on Security and Privacy*. S&P. 2012.

[52] Aditi Gupta, Sam Kerr, MichaelS. Kirkpatrick, and Elisa Bertino. "Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks." In: *Network and System Security*. Lecture Notes in Computer Science. 2013.

[53] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2015.

[54] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. "You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2014.

[55] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. "Readactor: Practical Code Randomization Resilient to Memory Disclosure." In: *36th IEEE Symposium on Security and Privacy*. S&P. 2015.

[56] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2015.

[57] Jannik Pewny and Thorsten Holz. "Control-flow Restrictor: Compiler-based CFI for iOS." In: *29th Annual Computer Security Applications Conference*. ACSAC. 2013.

[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *USENIX Security Symposium*. 2014, pp. 941–955.

[59] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. "CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters." In: *IEEE/IFIP Conference on Dependable Systems and Networks*. DSN. 2012.

[60] Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl, Debayan Paul, Ahmad-Reza Sadeghi, and Dean Sullivan. "HAFIX: Hardware-Assisted Flow Integrity Extension." In: *52nd Design Automation Conference*. DAC. 2015.

[61] Intel Corporation. *Control-flow enforcement technology preview*. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. 2016.

[62] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MemJam: A false dependency attack against constant-time crypto implementations in SGX." In: *Cryptographers' Track at the RSA Conference*. 10.1007/978-3-319-76953-0_2. Springer. 2018, pp. 21–44.

[63] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors." In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, pp. 361–372.

[64] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges.” In: *Black Hat* 15 (2015).

[65] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: exposing the perils of security-oblivious energy management.” In: (2017), pp. 1057–1074.

[66] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown.” In: *USENIX Security* (2018).

[67] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.” In: USENIX Security 18 (2018).

[68] Hong Hu, Shweta Shinde, Adrian Sendroiu, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. “Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks.” In: *37th IEEE Symposium on Security and Privacy*. S&P. 2016.

[69] David Gens, Simon Schmitt, Lucas Davi, and Ahmad Sadeghi. *K-Miner Source Code*.
https://github.com/ssl-tud/k-miner. 2017.

[70] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. “Razzer: Finding Kernel Race Bugs through Fuzzing.” In: *40th IEEE Symposium on Security and Privacy*. IEEE. 2019.

[71] Mark Lanteigne. *How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware*. https://www.thirdio.com/rowhammer.pdf. 2016.

[72] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges.” In: *Black Hat* (2015).

[73] Rui Qiao and Mark Seaborn. “A New Approach for Rowhammer Attacks.” In: *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. HOST. 2016.

[74] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Commodity Mobile Platforms.” In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2016.

[75] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer. js: A remote software-induced fault attack in javascript.” In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 300–321.

[76] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. “One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation.” In: *USENIX Security Symposium*. 2016, pp. 19–35.

[77] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Throwhammer: Rowhammer Attacks over the Network and Defenses.” In: *2018 USENIX Annual Technical Conference, (USENIX ATC18)*. USENIX Association. 2018.

[78] Christoph Baumann and Thorsten Bormer. “Verifying the PikeOS microkernel: First results in the Verisoft XT Avionics project.” In: *Doctoral Symposium on Systems Software Verification (DS SSV'09) Real Software, Real Problems, Real Solutions*. 2009, p. 20.

[79] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. “seL4: Formal verification of an OS kernel.” In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[80] R. Hund, C. Willems, and T. Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR.” In: *34th IEEE Symposium on Security and Privacy*. S&P. 2013.

[81] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR.” In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016.

[82] Yeongjin Jang, Sangho Lee, and Taesoo Kim. “Breaking Kernel Address Space Layout Randomization with Intel TSX.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 380–392.

[83] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 368–379.

[84] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “Kaslr is dead: long live kaslr.” In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, pp. 161–176.

[85] IEEE Computer Society - Austin Joint Working Group. *1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))*. http://standards.ieee.org/findstds/standard/1003.1-2008.html. 2008.

[86] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*. Vol. 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[87] National Security Agency. *Security-Enhanced Linux (SELinux)*.

[88] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. “A Secure Environment for Untrusted Helper Applications.” In: *6th USENIX Security Symposium*. USENIX Sec. 1996.

[89] Jonathan Corbet. *Yet another new approach to seccomp*. https://lwn.net/Articles/475043/. 2012.

[90] PaX. *PaX Address Space Layout Randomization*. 2003.

[91] Georg Hotz. *towelroot by geohot*. https://towelroot.com. 2015.

[92] Tarjei Mandt. *Attacking the iOS Kernel: A Look at "evasion"*. http://www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf. 2013.

[93] MacObserver. *Bungie Recalls Myth II: Installer Error*. https://www.macobserver.com/news/98/december/981229/bungierecall.html. 1998.

[94] PCWorld. *Scary Steam for Linux bug erases all the personal files on your PC*. https://www.pcworld.com/article/2871653/scary-steam-for-linux-bug-erases-all-the-personal-files-on-your-pc.html. 2015.

[95] Forbes. *Microsoft Warns Windows 10 Update Deletes Personal Data*. https://www.forbes.com/sites/gordonkelly/2018/10/06/microsoft-windows-10-update-lost-data-upgrade-windows-7-windows-xp-free-upgrade/. 2018.

[96] Benjamin Mayo. *Apple officially acknowledges reports of personal music files being deleted*. https://9to5mac.com/2016/05/13/apple-officially-acknowledges-reports-of-personal-music-files-being-deleted-itunes-update-coming-next-week-to-hopefully-fix-the-bug/. 2016.

[97] Tom Phillips. *Fallout 76 beta extended after bug which deleted 50GB data*. https://www.eurogamer.net/articles/2018-10-31-fallout-76-beta-extended-after-bug-which-deleted-50gb-data. 2018.

[98] Steve Christey and Robert A Martin. "Vulnerability type distributions in CVE." In: *Mitre report, May* (2007).

[99] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. "Linux kernel vulnerabilities: State-of-the-art defenses and open problems." In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM. 2011, p. 5.

[100] ANSI Committee X3J11. *ANSI C / C89 / ISO C90*. http://port70.net/~nsz/c/c89/c89-draft.html. 1989.

[101] ISO WG14. *ISO/IEC C99*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf. 1999.

[102] ISO WG14. *ISO/IEC C11*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf. 2011.

[103] Sergey Bratus, Michael Locasto, Meredith Patterson, Len Sassaman, and Anna Shubina. "Exploit programming: From buffer overflows to weird machines and theory of computation." In: *USENIX :login:* (2011).

[104] Thomas F Dullien. "Weird machines, exploitability, and provable unexploitability." In: *IEEE Transactions on Emerging Topics in Computing* (2017).

[105] Ralf Hund. "Analysis and Retrofitting of Security Properties for Proprietary Software Systems." PhD thesis. Ruhr-Universität Bochum, 2013.

[106] Lucas Davi and Ahmad-Reza Sadeghi. *Building Secure Defenses Against Code-Reuse Attacks*. Springer, 2015.

[107] Lucas Vincenzo Davi. "Code-reuse attacks and defenses." PhD thesis. Technische Universität Darmstadt, 2015.

[108] Felix Schuster. "Securing application software in modern adversarial settings." PhD thesis. Ruhr-Universität Bochum, 2016.

[109] Robert Gawlik. "On the impact of memory corruption vulnerabilities in client applications." PhD thesis. Ruhr-Universität Bochum, 2016.

[110] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. "SoK: Automated Software Diversity." In: *35th IEEE Symposium on Security and Privacy*. S&P. 2014.

[111] Christopher Liebchen. "Advancing Memory-corruption Attacks and Defenses." PhD thesis. Technische Universität Darmstadt, 2018.

[112] OpenBSD. *OpenBSD 3.3*. OpenBSD. 2003.

[113] Microsoft. *Data Execution Prevention (DEP)*. `http://support.microsoft.com/kb/875352/EN-US/`. 2006.

[114] Solar Designer. *Getting aroudn non-executable stack (and fix)*. `https://seclists.org/bugtraq/1997/Aug/63`. 1997.

[115] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. "Return-oriented programming without returns." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2010.

[116] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack." In: *6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS. 2011.

[117] Erik Bosman and Herbert Bos. "Framing Signals—A Return to Portable Shellcode." In: *35th IEEE Symposium on Security and Privacy*. S&P. 2014.

[118] Andrea Bittau, Adam Belay, Ali José Mashtizadeh, David Mazières, and Dan Boneh. "Hacking Blind." In: *35th IEEE Symposium on Security and Privacy*. S&P. 2014.

[119] Stephanie Forrest, Anil Somayaji, and David H. Ackley. "Building Diverse Computer Systems." In: *6th Workshop on Hot Topics in Operating Systems*. HotOS. 1997.

[120] Fermin J. Serna. "The Info Leak Era on Software Exploitation." In: *Blackhat USA*. BH US. 2012.

[121] Xiaobo Chen and Dan Caselden. *CVE-2013-3346/5065 Technical Analysis*. `http://www.fireeye.com/blog/technical/cyber-exploits/2013/12/cve-2013-33465065-technical-analysis.html`. 2013.

[122] Nicolas Joly. *Advanced Exploitation of Internet Explorer 10 / Windows 8 Overflow (Pwn2Own 2013)*. `http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php`. 2013.

[123] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. "Binary stirring: self-randomizing instruction addresses of legacy x86 binary code." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2012.

[124] Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity." In: *36th IEEE Symposium on Security and Privacy*. S&P. 2015.

[125] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection." In: *23rd USENIX Security Symposium*. USENIX Sec. 2014.

[126] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity." In: *24th USENIX Security Symposium*. USENIX Sec. 2015.

[127] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C." In: *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2009.

[128] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. "CETS: compiler enforced temporal safety for C." In: *International Symposium on Memory Management*. ISMM. 2010.

[129] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.

[130] Scott E Thompson and Srivatsan Parthasarathy. "Moore's law: the future of Si microelectronics." In: *Materials today* 9.6 (2006), pp. 20–25.

[131] Dean M Tullsen, Susan J Eggers, and Henry M Levy. "Simultaneous multithreading: Maximizing on-chip parallelism." In: *ACM SIGARCH Computer Architecture News*. Vol. 23. 2. ACM. 1995, pp. 392–403.

[132] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious." In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[133] Rafal Wojtczuk. *TSX improves timing attacks against KASLR*. https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/. 2014.

[134] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack." In: *25th USENIX Security Symposium*. USENIX Sec. 2016.

[135] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector." In: *37th IEEE Symposium on Security and Privacy*. S&P. 2016.

[136] ARM. *Security technology building a secure system using trustzone technology (white paper)*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. 2009.

[137] Positive Technologies. *Vulnerability enabling disclosure of Intel ME encryption keys*. http://blog.ptsecurity.com/2018/09/intel-me-encryption-vulnerability.html. 2018.

[138] NIST. *Buffer Overflow in BootROM Recovery Mode of NVIDIA Tegra mobile processors*. https://nvd.nist.gov/vuln/detail/CVE-2018-6242. 2018.

[139] *How to hack a turned off computer*. https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf. 2017.

[140] *Cease and DeSwitch - Fusée Gelée*. https://github.com/Cease-and-DeSwitch/fusee-launcher. 2018.

[141] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." In: *CoRR* (2018). URL: http://arxiv.org/abs/1801.01203.

[142] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX winter*. Vol. 46. 1993.

[143] Guang Gong. *Pwn a Nexus Device With a Single Vulnerability*. https://cansecwest.com/slides/2016/CSW2016_Gong_Pwn_a_Nexus_device_with_a_single_vulnerability.pdf. 2016.

[144] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. "Exploiting and Protecting Dynamic Code Generation." In: *22nd Annual Network and Distributed System Security Symposium*. NDSS. 2015.

[145] Mozilla. *W xor X JIT-code enabled in Firefox*. https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox. 2015.

[146] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines." In: *22nd Annual Network and Distributed System Security Symposium*. NDSS. 2015.

[147] Dion Blazakis. "Interpreter exploitation: Pointer inference and JIT spraying." In: *Blackhat DC*. BH DC. 2010.

[148] Ben Niu and Gang Tan. "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2014.

[149] Giorgi Maisuradze, Michael Backes, and Christian Rossow. "What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses." In: *25th USENIX Security Symposium*. USENIX Sec. 2016.

[150] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *23rd USENIX Security Symposium*. USENIX Sec. 2014.

[151] Ben Niu and Gang Tan. "Modular Control-flow Integrity." In: *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2014.

[152] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2015.

[153] Microsoft. *ChakraCore*. https://github.com/Microsoft/ChakraCore. 2015.

[154] Theori. *Chakra JIT CFG Bypass*. http://theori.io/research/chakra-jit-cfg-bypass. 2016.

[155] Matt Miller. *Mitigating arbitrary native code execution in Microsoft Edge*. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/. 2017.

[156] Microsoft. *Control Flow Guard*. http://msdn.microsoft.com/en-us/library/Dn919635.aspx. 2015.

[157] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. 2015.

[158] ARM. *ARM Architecture Reference Manual*. http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf. 2015.

[159] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-User Attacks." In: *USENIX Security Symposium*. Vol. 16. 2012.

[160] PaX Team. *RAP: RIP ROP*. GRSecurity. 2015.

[161] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. "ret2dir: Rethinking Kernel Isolation." In: *USENIX Security*. 2014, pp. 957–972.

[162] Jake Edge. *Kernel address space layout randomization*. http://lwn.net/Articles/569635. 2013.

[163] Ken Johnson and Matt Miller. *Exploit mitigation improvements in Windows 8*. https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf. 2012.

[164] Apple Inc. *OS X Mountain Lion Core Technologies Overview*. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf. 2012.

[165] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. "kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 420–436.

[166] Tilo Müller, Felix C. Freiling, and Andreas Dewald. "TRESOR Runs Encryption Securely Outside RAM." In: *20th USENIX Security Symposium*. USENIX Sec. 2011.

[167] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing." In: *22nd USENIX Security Symposium*. USENIX Sec. 2013.

[168] Larry McVoy and Carl Staelin. "Lmbench: Portable Tools for Performance Analysis." In: *USENIX Technical Conference*. ATEC. 1996.

[169] Phoronix. *Phoronix Test Suite*. http://www.phoronix-test-suite.com/. 2016.

[170] LTP developer. *The Linux Test Project*. https://linux-test-project.github.io/. 2016.

[171] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. “Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization.” In: *21st USENIX Security Symposium*. USENIX Sec. 2012.

[172] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. “Enforcing Kernel Security Invariants with Data Flow Integrity.” In: *23rd Annual Network and Distributed System Security Symposium*. NDSS. 2016.

[173] Ahmed Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. “SKEE: A lightweight Secure Kernel-level Execution Environment for ARM.” In: *23rd Annual Network and Distributed System Security Symposium*. NDSS. 2016.

[174] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. “Nested kernel: An operating system architecture for intra-kernel privilege separation.” In: *ACM SIGPLAN Notices* 50.4 (2015), pp. 191–206.

[175] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. “Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security.” In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 3. ACM. 2011, pp. 189–200.

[176] Gerwin Klein. “Operating System Verification — An Overview.” In: *Sādhanā* 34 (2009), pp. 27–69.

[177] Dave Jones. “Trinity: A system call fuzzer.” In: *Proceedings of the 13th Ottawa Linux Symposium, pages*. 2011.

[178] Mateusz Jurczyk and Gynvael Coldwind. “Identifying and exploiting windows kernel race conditions via memory access patterns.” In: (2013).

[179] David Drysdale. *Coverage-guided kernel fuzzing with syzkaller*. https://lwn.net/Articles/677764/. 2016.

[180] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The ASTRÉE analyzer.” In: *European Symposium on Programming*. Springer. 2005, pp. 21–30.

[181] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[182] John B Kam and Jeffrey D Ullman. “Monotone data flow analysis frameworks.” In: *Acta Informatica* 7.3 (1977), pp. 305–317.

[183] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability.” In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1995, pp. 49–61.

[184] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. "Documenting and automating collateral evolutions in Linux device drivers." In: *Acm sigops operating systems review*. Vol. 42. 4. ACM. 2008, pp. 247–260.

[185] Dan Carpenter. *Smatch - The Source Matcher*. http://smatch.sourceforge.net. 2009.

[186] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. "TypeChef: toward type checking# ifdef variability in C." In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. ACM. 2010, pp. 25–32.

[187] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. "APISan: Sanitizing API Usages through Semantic Cross-checking." In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, pp. 363–378.

[188] Iago Abal, Claus Brabrand, and Andrzej Wąsowski. "Effective Bug Finding in C Programs with Shape and Effect Abstractions." In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2017, pp. 34–54.

[189] Jonathan Corbet and Greg Kroah-Hartman. *Linux Kernel Development Report 2016*. http://go.linuxfoundation.org/linux-kernel-development-report-2016. 2016.

[190] Michael Hind. "Pointer analysis: Haven't we solved this problem yet?" In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2001, pp. 54–61.

[191] Ben Hardekopf and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code." In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE. 2011, pp. 289–298.

[192] Teck Bok Tok, Samuel Z Guyer, and Calvin Lin. "Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers." In: *International Conference on Compiler Construction*. Springer. 2006, pp. 17–31.

[193] Greg Kroah-Hartman. *Signed Kernel Modules*. http://www.linuxjournal.com/article/7130. 2004.

[194] Microsof. *Kernel-Mode Code Signing Walkthrough*. https://msdn.microsoft.com/en-us/library/windows/hardware/dn653569(v=vs.85).aspx. 2007.

[195] Apple. *Kernel Extensions*. https://developer.apple.com/library/content/documentation/Security/Conceptual/System_Integrity_Protection_Guide/KernelExtensions/KernelExtensions.html. 2014.

[196] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *IEEE/ACM International Symposium on Code Generation and Optimization*. CGO. 2004.

[197] Yulei Sui and Jingling Xue. "SVF: interprocedural static value-flow analysis in LLVM." In: *Proceedings of the 25th International Conference on Compiler Construction*. ACM. 2016, pp. 265–266.

[198] Behan Webster. *LLVMLinux*. http://llvm.linuxfoundation.org. 2014.

[199] Woerner Trevor. *How the Linux Kernel initcall Mechanism Works*. http://www.compsoc.man.ac.uk/~moz/kernelnewbies/documents/initcall/kernel.html. 2003.

[200] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. "An efficient method of computing static single assignment form." In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 25–35.

[201] ArchitectureReviewBoards. *OpenMP*. http://www.openmp.org/. 2017.

[202] Hacking Team. *Hacking Team Futex Exploit*. https://wikileaks.org/hackingteam/emails/emailid/312357. 2014.

[203] Google. *Android Security Bulletin - November 2016*. https://source.android.com/security/bulletin/2016-11-01. 2016.

[204] Zijlstra Peter. *printk: avoid double lock acquire*. https://github.com/torvalds/linux/commit/09dc3cf. 2011.

[205] Hellwig Christopher. *hfsplus: fix double lock typo in ioctl*. https://github.com/torvalds/linux/commit/e50fb58. 2010.

[206] Carpenter Dan. *drm/prime: double lock typo*. https://github.com/torvalds/linux/commit/0adb237. 2013.

[207] Tikhomirov Anton. *usb: phy: Fix double lock in OTG FSM*. https://github.com/torvalds/linux/commit/16da4b1. 2013.

[208] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. "Automatic Generation of Data-oriented Exploits." In: *24th USENIX Security Symposium*. USENIX Sec. 2015.

[209] Jonathan Salwan. *ROPgadget - gadgets finder and auto-roper*. http://shell-storm.org/project/ROPgadget/. 2011.

[210] Coseinc Nguyen Anh Quynh. *Capstone: next-gen dissassembly framework*. http://www.capstone-engine.org/. 2014.

[211] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. *Data Randomization*. Tech. rep. MSR-TR-2008-120. Microsoft Research, 2008. URL: \url{http://research.microsoft.com/apps/pubs/default.aspx?id=70626}.

[212] Sandeep Bhatkar and R. Sekar. "Data Space Randomization." In: *5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*. DIMVA. 2008.

[213] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. "Timely Rerandomization for Mitigating Memory Disclosures." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2015.

[214] Michael Backes and Stefan Nürnberger. "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing." In: *23rd USENIX Security Symposium*. USENIX Sec. 2014.

[215] Ping Chen, Yi Fang, Bing Mao, and Li Xie. "JITDefender: A Defense against JIT Spraying Attacks." In: *26th International Conference on ICT Systems Security and Privacy Protection*. IFIP SEC. 2011.

[216] P. Chen, R. Wu, and B. Mao. "JITSafe: a framework against Just-in-time spraying attacks." In: *IET Information Security* 7.4 (2013).

[217] Microsoft. *Out-of-process JIT support*. https://github.com/Microsoft/ChakraCore/pull/1561. 2016.

[218] Limin Zhu. *Chromium adotpion in Microsoft Edge and future of ChakraCore*. https://github.com/Microsoft/ChakraCore/issues/5865. 2018.

[219] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Librando: transparent code randomization for just-in-time compilers." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2013.

[220] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2015.

[221] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. "No-Execute-After-Read: Preventing Code Disclosure in Commodity Software." In: *11th ACM Symposium on Information, Computer and Communications Security*. ASIACCS. 2016.

[222] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. "Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks." In: *37th IEEE Symposium on Security and Privacy*. S&P. 2016.

[223] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. "Language-independent sandboxing of just-in-time compilation and self-modifying code." In: *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2011.

[224] Nicolas A. Economou and Enrique E. Nissim. *Getting Physical Extreme abuse of Intel based Paging Systems*. https://www.coresecurity.com/system/files/publications/2016/05/CSW2016%20-%20Getting%20Physical%20-%20Extended%20Version.pdf. 2016.

[225] MWR Labs. *Windows 8 Kernel Memory Protections Bypass*. http://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass. 2014.

[226] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. "SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture." In: *Mobile Security Technologies*. MoST. 2014.

[227] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World." In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2014.

[228] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes." In: *ACM SIGOPS Operating Systems Review* 41.6 (2007), pp. 335–350.

[229] Zhi Wang and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 380–395.

[230] Rafal Wojtczuk. "Subverting the Xen hypervisor." In: *Blackhat USA*. BH US. 2008.

[231] Alex Ionescu. *Owning the Image Object File Format, the Compiler Toolchain, and the Operating System: Solving Intractable Performance Problems Through Vertical Engineering*. www.alex-ionescu.com/?p=323. 2016.

[232] The kernel development community. *The Kernel Address Sanitizer (KASAN)*. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html. 2014.

[233] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. "TypeSan: Practical Type Confusion Detection." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 517–528.

[234] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. "CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 529–540.

[235] The kernel development community. *The Undefined Behavior Sanitizer (UBSAN)*. https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html. 2014.

[236] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[237] Linus Torvalds. *Sparse - a semantic parser for C*. https://sparse.wiki.kernel.org/index.php/Main_Page. 2006.

[238] Bhargava Shastry, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. "Towards Vulnerability Discovery Using Staged Program Analysis." In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 78–97.

[239] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. "Soot-a Java bytecode optimization framework." In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. 1999, p. 13.

[240] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms." In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 60–71.

[241] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field." In: *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE. 2015, pp. 415–426.

[242] Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens, and Wouter Joosen. "Bitsquatting: Exploiting bit-flips for fun, or profit?" In: *Proceedings of the 22nd international conference on World Wide Web*. ACM. 2013, pp. 989–998.

[243] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *25th USENIX Security Symposium*. USENIX Sec. 2016.

[244] GB Clarke, D Van Dijk, and SM Devadas. "Controlled physical random functions." In: *Proceedings. 18th Annual Computer Security Applications* (2002), pp. 149–160.

[245] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. "A technique to build a secret key in integrated circuits for identification and authentication applications." In: *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*. IEEE. 2004, pp. 176–179.

[246] G Edward Suh and Srinivas Devadas. "Physical unclonable functions for device authentication and secret key generation." In: *Proceedings of the 44th annual design automation conference*. ACM. 2007, pp. 9–14.

[247] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Berk Sunar, and Pim Tuyls. "Memory leakage-resilient encryption based on physically unclonable functions." In: *Towards Hardware-Intrinsic Security*. Springer, 2010, pp. 135–164.

[248] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. "PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon." In: *Cryptographic Hardware and Embedded Systems–CHES 2012* (2012), pp. 283–301.

[249] Wenjie Xiong, André Schaller, Nikolaos Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. "Practical DRAM PUFs in Commodity Devices." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 253.

[250] Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. "Run-time Accessible DRAM PUFs in Commodity Devices." In: *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2016, pp. 432–453.

[251] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. "Intrinsic Rowhammer PUFs: Leveraging the Rowhammer Effect for Improved Security." In: *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2017, pp. 1–7.

[252] Fatemeh Tehranipoor, Nima Karimian, Wei Yan, and John A Chandy. "DRAM-Based Intrinsic Physically Unclonable Functions for System-Level Security and Authentication." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.3 (2017), pp. 1085–1097.

[253] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. "ANVIL: Software-based protection against next-generation rowhammer attacks." In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2016, pp. 743–755.

[254] Yoongu Kim et. al. *Rowhammer Memtest*. https://github.com/CMU-SAFARI/rowhammer. 2014.

[255] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. *ACPI 4.0 - System Address Interface*. http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf. 2010.

[256] Free Software Foundation. *GNU GRUB*. https://www.gnu.org/software/grub. 2010.

[257] Jiewen Yao and Vincent J Zimmer. *A Tour Beyond BIOS Memory Map*. https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_inUEFI_BIOS.pdf. 2015.

[258] AMD. *Intel 64 and IA-32 Architectures Software Developer's Manual - Chapter 15 Secure Virtual Machine Nested Paging*. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals. 2012.

[259] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Chapter 28 VMX Support for address translation*. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[260] I AMD. "I/O Virtualization Technology (IOMMU) Specification." In: *AMD Pub* 34434 (2007).

[261] John L. Henning. "SPEC CPU2006 memory footprint." In: *SIGARCH Computer Architecture News* 35 (2007).

[262] Ad J Van De Goor and Ivo Schanstra. "Address and data scrambling: Causes and impact on memory tests." In: *The First IEEE International Workshop on Electronic Design, Test and Applications, 2002. Proceedings.* IEEE. 2002, pp. 128–136.

[263] MITRE. *CVE-2015-1328*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1328. 2015.

[264] MITRE. *CVE-2016-0728*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728. 2016.

[265] CVEDetails. *CVE-2016-4557*. http://www.cvedetails.com/cve/cve-2016-4557. 2016.

[266] Matt Molinyawe, Abdul-Aziz Hariri, and Jasiel Spelman. "$hell on Earth: From Browser to System Compromise." In: *Blackhat USA*. BH US. 2016.

[267] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes 3A, 3B, and 3C: System Programming Guide*. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf. 2016.

[268] Oracle. *What are some distinguishing characteristics of OpenSPARC T1 and T2*. https://www.oracle.com/technetwork/systems/opensparc/opensparc-faq-1444660.html. 2018.

[269] Hareesh Khattri, Narasimha Kumar V Mangipudi, and Salvador Mandujano. "Hsdl: A security development lifecycle for hardware technologies." In: (2012), pp. 116–121.

[270] Cisco. *Cisco: Strengthening Cisco Products*. https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle.html. 2017.

[271] Lenovo. *Lenovo: Taking Action on Product Security*. https://www.lenovo.com/us/en/product-security/about-lenovo-product-security. 2017.

[272] Jason Oberg. *Secure Development Lifecycle for Hardware Becomes an Imperative*. https://www.eetimes.com/author.asp?section_id=36&doc_id=1332962. 2018.

[273] Cadence. *Incisive Enterprise Simulator*. 2014.

[274] Averant. *Solidify*. http://www.averant.com/storage/documents/Solidify.pdf. 2018.

[275] Mentor. *Questa Verification Solution*. https://www.mentor.com/products/fv/questa-verification-platform. 2018.

[276] OneSpin Solutions. *OneSpin 360*. https://www.onespin.com/fileadmin/user_upload/pdf/datasheet_dv_web.pdf. 2013.

[277] Cadence. *JasperGold Formal Verification Platform*. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html. 2014.

[278] David Evans and David Larochelle. "Improving security using extensible lightweight static analysis." In: *IEEE software* 19.1 (2002), pp. 42–51.

[279] Jason Oberg. *Secure Development Lifecycle for Hardware Becomes an Imperative*. https://www.eetimes.com/author.asp?section_id=36&doc_id=1332962. 2018.

[280] Caroline Trippel, Yatin A Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA." In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 52.4 (2017), pp. 119–133.

[281] Manuel Blum and Hal Wasserman. "Reflections on the Pentium division bug." In: *IEEE Transactions on Computers* 45.4 (1996), pp. 385–393.

[282] Intel Corporation. “81. Invalid Operand with Locked CMPXCHG8B Instruction.” In: *Pentium® Processor Specification Update* 41 (1998), 51f.

[283] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: a timing attack on OpenSSL constant-time RSA.” In: *Journal of Cryptographic Engineering* 7.2 (2017). 10.1007/s13389-017-0152-y, pp. 99–112.

[284] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks.” In: (2018).

[285] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor.” In: (2018), pp. 693–707.

[286] NIST. *Google: Escalation of Privilege Vulnerability in MediaTek WiFi driver*. https://nvd.nist.gov/vuln/detail/CVE-2016-2453. 2016.

[287] NIST. *Samsung: Page table walks conducted by MMU during Virtual to Physical address translation leaves in trace in LLC*. https://nvd.nist.gov/vuln/detail/CVE-2017-5927. 2017.

[288] NIST. *HP: Remote update feature in HP LaserJet printers does not require password*. https://nvd.nist.gov/vuln/detail/CVE-2004-2439. 2004.

[289] NIST. *Apple: Multiple heap-based buffer overflows in the AudioCodecs library in the iPhone allows remote attackers to execute arbitrary code or cause DoS via a crafted AAC/MP3 file*. https://nvd.nist.gov/vuln/detail/CVE-2009-2206. 2009.

[290] NIST. *Amazon Kindle Touch does not properly restrict access to the NPAPI plugin which allows attackers to have an unspecified impact via certain vectors*. https://nvd.nist.gov/vuln/detail/CVE-2012-4249. 2012.

[291] NIST. *Vulnerabilities in Dell BIOS allows local users to bypass intended BIOS signing requirements and install arbitrary BIOS images*. https://nvd.nist.gov/vuln/detail/CVE-2013-3582. 2013.

[292] NIST. *Microsoft: Hypervisor in Xbox 360 kernel allows attackers with physical access to force execution of the hypervisor syscall with a certain register set, which bypasses intended code protection*. https://nvd.nist.gov/vuln/detail/CVE-2007-1221. 2007.

[293] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Tech. rep. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCES, 2014.

[294] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. “The rocket chip generator.” In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[295] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolić, David A Patterson, and Krste Asanović. “BOOMv2: an open-source out-of-order RISC-V core.” In: (2017).

[296] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. “Control-flow integrity.” In: (2005), pp. 340–353.

[297] Miguel Castro, Manuel Costa, and Tim Harris. “Securing software by enforcing data-flow integrity.” In: (2006), pp. 147–160.

[298] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code.” In: *30th IEEE Symposium on Security and Privacy*. S&P. 2009.

[299] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. “Software mitigations to hedge AES against cache-based software side channel vulnerabilities.” In: *IACR Cryptology ePrint Archive* 2006 (2006), p. 52.

ERKLÄRUNG GEMÄSS §9 DER PROMOTIONSORDNUNG

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendnung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Dezember 2018

David Gens

This document was produced using free software, such as the typesetting system LaTeX, the extensible text editor Emacs, and the operating system GNU/Linux. The typographical style of this thesis was inspired by Robert Bringhurst's seminal book "*The Elements of Typographic Style*" and realized through André Miede's useful template.