SAPIENZA
UNIVERSITÀ DI ROMA

Taming Complex Bugs in Secure Systems

School of Information Engineering, Computer Science and Statistics
Ph.D. in Engineering in Computer Science (XXXV cycle)

Pietro Borrello
ID number 1647357

Advisor
Prof. Leonardo Querzoni

Co-Advisor
Prof. Daniele Cono D'Elia

Academic Year 2021/2022

Thesis defended on May 18, 2023

Taming Complex Bugs in Secure Systems
PhD thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: borrello@diag.uniroma1.it

To you, the sparkle of my life, constant love, inspiration and strength.
To my family and friends for their constant support.
To all the people I have worked with in this incredible journey:
we've done amazing things.

Abstract

Modern systems rely on several layers of abstraction to implement their expected functionality. Securing complex systems is extremely difficult, as any abstraction layer can be a possible target. While defense-in-depth techniques help, attackers resort to chaining multiple exploits to break the different protection layers. Thus, malicious actors increase their attack complexity to exploit even the most secure systems.

Vulnerabilities may hide in each layer of a modern system. Software applications are often the initial attack target, as they are usually exposed to the external world. Once a malicious actor can exploit a software application, it usually moves to increase their privilege, targetting other running applications or the operating system to exfiltrate data or achieve persistence. From a compromised operating system, an attacker may even target hypervisors and CPU enclaves aimed at protection against privileged attackers.

Several techniques attempt to slow down attackers, make bugs hard to find, or, better, find and fix bugs in the first place. Obfuscation techniques complicate the software under analysis, slowing down program understanding from external actors. Automatic software testing aims to find bugs during development to avoid possible vulnerabilities in production systems. Enclaves protect software from a compromised operating system but require CPUs to support them correctly. However, given enough time, attackers can often break each protection layer by combining several complex vulnerabilities to take control of an entire secure system. Thus, a comprehensive approach needs to tackle vulnerabilities across several system layers to increase the attack complexity required to break modern systems.

This thesis examines and enhances the security of multiple layers of modern systems. We design, implement and evaluate techniques to find and mitigate complex vulnerabilities across most of the layers of secure systems. Throughout the thesis, we build solutions by sequentially breaking assumptions on attacker capabilities. To slow down attackers, we propose techniques to obfuscate software against reverse engineering efficiently. To secure software and operating systems, we improve existing dynamic testing techniques to find vulnerabilities at a scale and introduce new strategies to find novel bug patterns. We design compilation frameworks to protect software from subtle bugs, from type confusion to CPU side-channel bugs. To deepen the understanding of the threats of privileged attackers, we systematize hardware bug classes, comparing architectural and microarchitectural vulnerabilities to software ones. In doing so, we discover the first architectural CPU bug that leaks confidential data without side channels. Finally, we propose the first CPU framework to inspect and customize modern CPU microcode, offering an unprecedented view of the inner working of current systems. As a byproduct of our research, we find, report, and mitigate over one hundred vulnerabilities across most modern system layers, including application software, operating systems, and CPUs.

All the prototypes proposed in this thesis are open-sourced to foster future security research.

Contents

Abstract iii

Contents iv

1 Introduction 1
- 1.1 Topic of This Work . . . 2
- 1.2 Contributions . . . 4

2 Prevent Reverse Engineering Attacks Using Code Reuse 9
- 2.1 Introduction . . . 9
- 2.2 Background . . . 11
  - 2.2.1 Code Obfuscation . . . 11
  - 2.2.2 Return-Oriented Programming . . . 12
- 2.3 Adversarial Model . . . 13
  - 2.3.1 Principles behind Automated Deobfuscation . . . 13
  - 2.3.2 State-of-the-art Deobfuscation Solutions . . . 14
    - 2.3.2.1 General Techniques . . . 14
    - 2.3.2.2 ROP-Aware Techniques . . . 15
- 2.4 Program Encoding with ROP . . . 15
  - 2.4.1 Geometry of a ROP Encoder . . . 16
    - 2.4.1.1 Gadget Sources . . . 16
    - 2.4.1.2 Rewriting . . . 16
    - 2.4.1.3 Control Transfers and Stack Layout . . . 16
    - 2.4.1.4 Chain Embedding . . . 17
  - 2.4.2 Translation, Chain Crafting, and Materialization . . . 17
    - 2.4.2.1 Translation . . . 17
    - 2.4.2.2 Chain Crafting . . . 18
    - 2.4.2.3 Materialization . . . 20
  - 2.4.3 Discussion . . . 20
- 2.5 Strengthening ROP Programs . . . 21
  - 2.5.1 Predicate P_1: Anti-ROP-Disassembly . . . 22
  - 2.5.2 Predicate P_2: Preventing Brute-Force Search . . . 23
  - 2.5.3 Predicate P_3: State Space Widening . . . 24
  - 2.5.4 Gadget Confusion . . . 25
  - 2.5.5 Further Remarks . . . 25
- 2.6 Related Work . . . 26

2.7 Evaluation . . . 26
2.7.1 Efficacy of ROP Strengthening Transformations . . . 27
2.7.1.1 General Attacks . . . 27
2.7.1.2 ROP-Aware Attacks . . . 28
2.7.2 Measuring Obfuscation Resilience . . . 29
2.7.2.1 Secret Finding . . . 30
2.7.2.2 Code Coverage . . . 30
2.7.3 Deployability . . . 31
2.7.3.1 Coverage . . . 31
2.7.3.2 Overhead . . . 31
2.7.3.3 Case Study . . . 31
2.8 Conclusion . . . 32

3 Predictive Context-sensitive Fuzzing 34
3.1 Introduction . . . 34
3.2 Background . . . 36
3.2.1 Coverage-guided Fuzzing . . . 36
3.2.2 Pointer Analysis . . . 37
3.3 Motivation and Open Problems . . . 38
3.4 Predictive Context Sensitivity . . . 41
3.4.1 Function Cloning . . . 42
3.4.2 The Need for Selective Sensitivity . . . 43
3.4.3 Data Flow-based Prediction . . . 44
3.4.4 Discussion . . . 45
3.5 Implementation . . . 46
3.6 Evaluation . . . 47
3.6.1 RQ1: Analysis and Compilation Costs . . . 48
3.6.2 RQ2: Effectiveness in Bug Finding . . . 49
3.6.3 RQ3: Internal Wastage . . . 53
3.6.4 RQ4: New Bugs . . . 54
3.6.5 Discussion . . . 56
3.7 Related Work . . . 57
3.8 Conclusion . . . 58

4 Uncovering Container Confusion Bugs in the Linux Kernel 60
4.1 Introduction . . . 60
4.2 Background . . . 62
4.2.1 Type Confusion Bugs in C++... *and in C* . . . 62
4.2.2 Sanitizers . . . 64
4.3 Container Confusion in the Linux Kernel . . . 64
4.3.1 Security Implications . . . 64
4.3.2 Running Example . . . 65
4.3.3 Type Graph Complexity . . . 66
4.4 UNCONTAINED Overview . . . 67
4.5 Container Confusion Sanitizer . . . 68
4.5.1 Design . . . 68
4.5.2 Implementation . . . 71

4.5.3 Evaluation . . . 72
4.5.3.1 Discovered Cases of Container Confusion . . . 72
4.5.3.2 Runtime Overhead . . . 73
4.6 Retrospective Analysis and Bug Patterns . . . 73
4.7 Static Analyzer . . . 78
4.7.1 Design . . . 78
4.7.2 Implementation . . . 80
4.7.3 Evaluation . . . 81
4.8 Discussion . . . 82
4.9 Related Work . . . 83
4.10 Conclusion . . . 84

5 Automatic Side-Channel Resistance 86
5.1 Introduction . . . 86
5.2 Background . . . 88
5.3 Threat Model . . . 91
5.4 Constantine . . . 92
5.4.1 Overview . . . 92
5.4.2 Control Flow Linearization . . . 93
5.4.2.1 Dummy Execution . . . 94
5.4.2.2 Compiler IR Normalization . . . 95
5.4.2.3 Branch Linearization . . . 95
5.4.2.4 Loop Linearization . . . 96
5.4.2.5 Operand Sanitization . . . 98
5.4.2.6 Code Generation . . . 98
5.4.3 Data Flow Linearization . . . 99
5.4.3.1 Load and Store Wrappers . . . 100
5.4.3.2 Object Lifetime . . . 101
5.4.3.3 Optimizations . . . 101
5.4.4 Support Analyses . . . 102
5.4.4.1 Identifying Sensitive Program Portions . . . 102
5.4.4.2 Points-to Analysis . . . 103
5.4.5 Discussion . . . 104
5.5 Security Analysis . . . 105
5.6 Performance Evaluation . . . 107
5.7 Case Study . . . 111
5.8 Conclusion . . . 113

6 Architectural CPU Vulnerabilities 114
6.1 Introduction . . . 114
6.2 Background . . . 117
6.2.1 APIC . . . 117
6.2.2 Memory Subsystem . . . 117
6.2.3 Intel SGX . . . 118
6.2.4 Transient-Execution Attacks . . . 119
6.3 Software and Hardware Vulnerabilities . . . 119
6.3.1 Types of Vulnerabilities . . . 119

6.3.2 Classification of Vulnerabilities . . . 121
6.3.3 Missing Architectural Counterpart Discovery . . . 124
6.4 ÆPIC Leak Overview . . . 124
6.4.1 Attack Overview . . . 124
6.4.2 Threat Model . . . 126
6.4.3 Leakage Analysis . . . 127
6.4.3.1 Ruling out Microarchitectural Elements. . . . 127
6.4.3.2 Performance Counter Analysis. . . . 128
6.4.4 Building Blocks . . . 129
6.4.5 Performance Evaluation . . . 131
6.5 ÆPIC Leak Exploitation . . . 131
6.5.1 Attack Techniques . . . 131
6.5.2 Breaking AES-NI . . . 132
6.5.3 Breaking RSA . . . 133
6.5.4 Breaking SGX Attestation . . . 134
6.6 Mitigations . . . 135
6.6.1 Hardware . . . 135
6.6.2 Firmware . . . 135
6.6.3 Software . . . 136
6.7 Conclusion . . . 138

7 Reverse Engineering and Customization of Intel Microcode 139
7.1 Introduction . . . 139
7.2 Background . . . 142
7.2.1 Microcode Structure . . . 142
7.2.1.1 Microcode Patches . . . 142
7.2.1.2 Microcode Hooks . . . 142
7.2.1.3 Control Register Bus (CRBUS) and Local Data Access Test Port (LDAT) . . . 143
7.2.1.4 Red Unlock . . . 143
7.2.1.5 Undocumented Debug Instructions . . . 143
7.3 Framework . . . 143
7.3.1 Static Analysis . . . 144
7.3.2 Dynamic Analysis . . . 145
7.3.2.1 Reverse-Engineering LDAT Accesses . . . 145
7.3.2.2 Microcode Hooks . . . 146
7.3.2.3 Microcode Patches . . . 146
7.3.2.4 Microcode Traces . . . 146
7.3.2.5 Customizing `rdrand` . . . 147
7.4 Case Study: Reverse-Engineering the Microcode Update Routine . . . 148
7.4.1 Reverse Engineering . . . 149
7.4.2 Security Analysis . . . 151
7.5 Case Study: x86 Pointer Authentication Codes . . . 153
7.5.1 Background . . . 153
7.5.2 Implementation . . . 153
7.5.3 Security Analysis . . . 155
7.6 Case Study: μsoftware Breakpoints . . . 156

7.6.1 Background . . . 156
7.6.2 Implementation . . . 157
7.7 Case Study: Constant-time Hardware Division . . . 157
7.7.1 Background . . . 157
7.7.2 Implementation . . . 159
7.8 Conclusion . . . 159

8 Conclusions and Future Work 160

A Scientific Publications 162

B Practical Contributions 164

C Chapter 2: Additional Material 166
C.1 Implementation Aspects . . . 166
C.1.1 Switch Tables . . . 166
C.1.2 From Native to ROP and Back . . . 167
C.1.3 Tail Jumps . . . 167
C.2 Evaluation Additions . . . 168
C.2.1 Functions for Obfuscation Resilience Experiments . . . 168
C.2.2 VM Obfuscation . . . 168
C.2.3 Additional Deployability Experiments . . . 169

D Chapter 3: Additional Material 171
D.1 Additional Charts and Tables . . . 171
D.2 A Fast Intra-procedural Alternative . . . 173
D.3 Cloning Budget . . . 174

E Chapter 4: Additional Material 177
E.1 LMbench Evaluation . . . 177
E.2 Static Analysis Rules . . . 177

F Chapter 5: Additional Material 180
F.1 Conditional Selection . . . 180
F.2 Striding . . . 181
F.3 Field Sensitivity . . . 183
F.4 Recursion and Thread Safety . . . 184
F.5 Correctness . . . 184
F.6 Complete Run-Time Overhead Data . . . 186

G Chapter 6: Additional Material 187
G.1 Software Workaround . . . 187
G.2 Transient AES Computation . . . 188
G.3 Performance Counters and CPUs . . . 188

Bibliography 191

Chapter 1

Introduction

Modern application software is incredibly complex and depends on the abstractions provided by the underlying operating system for its proper functionality and security. Operating systems are considered some of the most intricate and interdependent software collections the world relies on. In turn, operating systems proper functionality and security rely on the abstractions provided by CPUs at the hardware level.

Each abstraction layer depends on the correct implementation of the layer below it. However, each implementation may deviate from the defined abstraction, and any such deviation in any abstraction layer may lead to vulnerabilities.

For a system to be considered secure, each layer of abstraction must be implemented correctly, all interactions with the underlying layer must adhere to the abstraction specification, and any implementation details of the underlying abstraction must not pose a threat to the security of the abstraction itself.

The software itself may contain flaws that could jeopardize its security. Software bugs can range from semantic bugs caused by incorrect algorithms [110] to implementation bugs in correctly designed algorithms [110] to side-channel bugs in correctly implemented algorithms [394]. It is crucial to detect these bugs as soon as possible before malicious actors can exploit them and take control of the system. Given the complexity of current systems, the question is not whether bugs exist or not but rather how long it will take someone to find them. The security community is constantly striving to increase the time it takes malicious actors to find bugs while simultaneously decreasing the time developers need to detect them.

One way to slow down attackers is through software obfuscation, which makes it more difficult for attackers who do not have access to the source code to understand the target [78]. Software obfuscation additionally hides implementation details, slowing down manual bug finding for attackers interested in exploiting a target. On the other hand, to quickly find bugs in software, automated testing encompasses techniques such as static and dynamic analysis. Static analysis involves looking for static patterns in the codebase to spot bugs, while dynamic analysis collects runtime information about possibly buggy program executions. Software testing can be applied to implementations of different abstraction levels, from software applications to operating systems [139, 373, 399].

However, not all bugs can be easily detected through software testing. First,

silent bugs may not trigger the machinery put in place by dynamic testing techniques, leading to a corrupted state that goes unnoticed, and static testing techniques may not be fine-tuned for specific bug classes. Second, critical software, such as cryptographic software, may not be considered secure even if it does not have implementation bugs if an attacker can *infer* secret information through side channels. In this case, the hardware's abstraction level is violated by its actual implementation, allowing changes in the *microarchitectural* state to become observable.

Finally, some abstractions are inherently challenging to test. CPUs are a prime example, as their inner workings are largely undocumented. An attacker who can reverse engineer these implementations may discover bugs that violate assumptions upon which the operating system relies (e.g., memory protection), breaking these guarantees.

In practice, attackers combine several possibly-complex vulnerabilities in the software stack, operating system, and CPUs to take control of an entire system previously deemed secure.

1.1 Topic of This Work

In this thesis, we pursue a comprehensive approach to examine and enhance the security of modern systems, from software applications to bare-metal hardware. Instead of focusing on issues of a single layer of abstraction or improving a single technique, we approach the problem from multiple perspectives to uncover and address a wide spectrum of complex vulnerabilities in modern systems. We examine several layers of abstraction, from application software to operating systems to CPUs, to gain a comprehensive understanding of modern, complex systems. We tackle the problem by applying a variety of techniques, such as binary translation, static and dynamic analysis, and specialized compilation frameworks to identify and mitigate vulnerabilities.

Throughout the thesis, we build solutions by sequentially breaking assumptions on attacker capabilities. We start with the weakest attacker, who has no access to the application source code and tries to recover application logic. We then move on to attackers who can interact with the applications to exploit them or run local code to exfiltrate secrets or attack the operating system. We end up considering the strongest possible attacker by focusing on trusted execution environments, which try to protect software running on a fully compromised operating system and hypervisor.

In this thesis, we build solutions to find and mitigate vulnerabilities automatically. Using the techniques proposed, we find, disclose and mitigate *more than a hundred vulnerabilities* affecting widely used open-source software, operating systems and CPUs.

We will explore the following topics:

Software Obfuscation. Software obfuscation aims at preventing reverse engineering attacks. It involves protection mechanisms applied to source code or directly at the binary level to prevent an attacker from understanding program logic. It applies transformations that complicate the program structure while preserving the original semantics [78]. While the technique is highly effective against a manual attacker,

automated attacks are often able to reverse the transformations, bypassing code obfuscation.

Dynamic Analysis. Dynamic analysis collects information during program execution and can naturally back automated testing approaches. Several forms of dynamic analysis exist. This thesis will focus on dynamic analyses tailored to bug finding. Fuzzing is one of the most widely used: it randomly provides unexpected inputs to a program to induce crashes while maximizing testing coverage [399]. It is highly effective in finding bugs, and it is generally *sound*, by providing concrete inputs that trigger the crashes found. The analysis usually underapproximates bugs due to the inability to fully explore program states, incurring false negatives. Moreover, a fuzzer may explore a buggy program state without triggering a crash, thus not detecting the bug.

Static Analysis. Static analysis collects patterns in codebases by analyzing static program information. Several static analysis techniques exist, ranging from the simplest ones leveraging source code pattern matching [284] to the most complex ones using symbolic execution to model program states [329]. This thesis will focus on static analyses tailored to bug finding. Static analysis usually tries to be *complete* by overapproximating patterns to scale over huge codebases, thus incurring false positives. However, it usually relies on precise bug models and cannot find bugs that are different from the ones expected.

Software Sanitizers. Sanitizers combine with dynamic software testing techniques to detect undefined behavior, typically by inserting checks at compilation time. Any time a check detects an invalid program state, it induces a program crash. This allows sanitizers to detect bugs that are more subtle and would not incur in a program crash themselves. Several sanitizers exist, ranging from detecting memory corruption errors [320] to race-conditions [321]. They are usually combined with fuzzers to automatically explore program states and detect crashes. However, similarly to static analysis techniques, they only detect the type of bugs they are programmed to, incurring false negatives when hitting a bug they do not recognize.

Software Mitigations. Software mitigations are similar in spirit to sanitizers in combatting bug patterns. However, they are designed to prevent bugs from being exploited instead of detecting them at an early stage. Several mitigations exist, from simple randomization techniques that make exploits harder [288] to compiler passes that change code to detect memory corruption while running with low overhead (e.g., stack canaries [110]) or removing bug patterns (e.g., side-channel mitigations [301]). Unlike sanitizers, software mitigations generally need a negligible overhead to be integrated with the software stack. Due to this need, they are usually more specialized in detecting and preventing subsets of bug patterns.

Hardware Mitigations. Hardware mitigations complement software mitigations by providing similar functionality at the hardware level. They usually guarantee higher efficiency than the software ones but require hardware vendors to integrate

them. Contrary to software mitigations, a bypass on hardware mitigations cannot usually be fixed by a software update, and any bug found can only be mitigated. A partial hardware redesign is usually needed to fix them [236, 302].

1.2 Contributions

We make six main contributions in this thesis. We approach different levels of abstractions to asses complex vulnerabilities in secure systems. We explore different attackers capabilities to find and mitigate over a hundred bugs in a wide range of software, operating systems and CPUs. We propose new techniques to automatically protect software, mitigate subtle issues (e.g., side channels), better find existing bug classes and introduce new ones.

The thesis is divided as follows. Each chapter introduces the relevant background, presents the problem under study and the proposed solutions:

Chapter 2: Prevent Reverse Engineering Attacks Using Code Reuse. We bring novel ideas to the software protection realm, by studying how code reuse exploitation techniques may provide strong software obfuscation against manual and automatic reverse-engineering attacks. We present `raindrop`, a binary obfuscation framework that transforms functions into obfuscated *ropchains*. A ropchain is a sequence of program fragments (or gadgets) that implement a specific functionality, usually leveraged in modern exploitation [323]. We evaluate our framework showing how it can provide strong protection against reverse engineering, with a lower overhead compared to state-of-the-art obfuscation techniques.

The results of this work are presented in:

- P. Borrello, E. Coppa, and D. C. D'Elia. "Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation". In the *Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), 2021.

While software obfuscation helps against attackers that try to find bugs with no access to source code, finding and fixing bugs before the software is released prevents attackers from exploiting them in the first place. Automated techniques, like fuzzing, are among the most effective approaches to find complex bugs in modern software.

Chapter 3: Predictive Context-sensitive Fuzzing. Modern fuzzers use classic code coverage as exploration feedback. This can lead them to overlook interesting program states when also *how* the execution reaches a program location matters. While context-sensitive coverage tracking is a promising solution, it incurs in an inherent state explosion problem. We show that a much more effective approach to context-sensitive fuzzing, which we term *predictive*, is possible, where we select program regions where we predict is worth to add context sensitivity. We focus context sensitivity on regions with high dataflow diversity. We show how our approach largely outperforms state-of-the-art coverage-guided fuzzing embodiments, unveiling more and different bugs without incurring explosion or other internal

wastage. Using our approach, we found *18 security issues* in 11 OSS-Fuzz subjects, with 11 CVE identifiers issued.

The results of this work are presented in:

- P. Borrello, A. Fioraldi, D.C. D'Elia, D. Balzarotti, L. Querzoni, C. Giuffrida. "Predictive Context-sensitive Fuzzing". *Under review*, 2023.

While fuzzing is highly effective to find bugs resulting in memory corruption, subtle bugs that do not result in a corrupted state are easily missed. In Chapter 4, we focus on finding type-confusion bugs missed by state of the art tools.

Chapter 4: Uncovering Container Confusion Bugs in the Linux Kernel. Type confusion bugs are a common source of security problems whenever software makes use of type hierarchies. Where existing research mostly studies type confusion in the context of object-oriented languages such as C++, we analyze how similar bugs affect complex C projects such as the *Linux kernel*, where structure embedding emulates type inheritance between typed structures. We take a systematic approach to discover type confusion vulnerabilities resulting from incorrect downcasting on structure embeddings, which we call *container confusion*. We design a novel sanitizer to detect such issues at runtime and evaluate it on the Linux kernel. Using the patterns in the bugs detected by the sanitizer, we then develop a static analyzer to find container confusion bugs in code that dynamic analysis fails to reach. We use our framework, UNCONTAINED, to find 89 *container confusion bugs* in the Linux kernel and submit patches to fix all of them. At the time of writing 94 of our patches have been merged.

The results of this work are presented in:

- J. Koschel*, P. Borrello*, D.C. D'Elia, H. Bos, C. Giuffrida. "UNCONTAINED: Uncovering Container Confusion in the Linux Kernel". *USENIX Security Symposium*, 2023.
  **Equal contribution joint first authors.*

While the absence of software vulnerabilities deems most software as secure, particularly security-sensitive software, like cryptographic one, needs stronger guarantees. The absence of implementation bugs, may still hide side channel vulnerabilities, where attackers are able to exfiltrate sensitive data.

Chapter 5: Automatic Side-Channel Resistance. In the era of microarchitectural side channels, vendors scramble to deploy mitigations for transient execution attacks, but leave traditional side-channel attacks against sensitive software (e.g., crypto programs) to be fixed by developers by means of *constant-time programming* (i.e., absence of secret-dependent code/data patterns). Unfortunately, writing constant-time code by hand is hard, as evidenced by the many flaws discovered in production side channel-resistant code, and existing automated solutions offer limited security or compatibility guarantees. We present CONSTANTINE, a compiler-based system to automatically harden programs against microarchitectural side channels. CONSTANTINE pursues a radical design point where secret-dependent control and data flows are *completely linearized*. This strategy provides strong security and

compatibility guarantees by construction. To address state explosion challenges in real-world programs, CONSTANTINE relies on carefully designed optimizations that lead to an efficient and compatible solution. CONSTANTINE yields overheads as low as 16% on standard benchmarks and can handle a fully-fledged component from the production wolfSSL library.

The results of this work are presented in:

- P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida. "Constantine: Automatic side-channel resistance using efficient control and data flow linearization.". In the *ACM SIGSAC Conference on Computer and Communications Security* (CCS), 2021.

While software mitigations can address some microarchitectural vulnerabilities that cause side-channels, CPU vulnerabilities undermine the security guarantees provided by software- and hardware-security improvements.

Chapter 6: Architectural CPU Vulnerabilities. The discovery of transient-execution attacks increased the interest in CPU vulnerabilities on a microarchitectural level, however, architectural CPU vulnerabilities are still understudied. We systematically analyze existing CPU vulnerabilities showing that CPUs suffer from vulnerabilities whose root causes match with those in complex software. We show that transient-execution attacks and architectural vulnerabilities often arise from the same type of bug and identify the blank spots. Investigating the blank spots, we focus on architecturally improperly initialized data locations. We discover ÆPIC Leak, the first architectural CPU bug that leaks stale data from the microarchitecture without using a side channel. ÆPIC Leak works on all recent Sunny-Cove-based Intel CPUs (*i.e.*, Ice Lake and Alder Lake), not requiring hyperthreading or any side-channel. It architecturally leaks stale data incorrectly returned by reading undefined APIC-register ranges. We present two new techniques, Cache Line Freezing and Enclave Shaking, to leak specific data from the cache hierarchy. We present end-to-end attacks which extract AES-NI and RSA keys, and even the Intel SGX attestation keys from enclaves within a few seconds.

The results of this work are presented in:

- P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In the *USENIX Security Symposium*, 2022.

- P. Borrello, A. Kogler. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *BlackHat USA*, 2022.

This line of research was awarded the 2022 Pwnie Award for *Best Desktop Bug*.

We thus show how the black box nature of existing CPUs may hide vulnerabilities that destroy guarantees provided to all the abstractions that rely on them.

Chapter 7: Reverse Engineering and Customization of Intel Microcode. CPU microcode provides an abstraction layer over the instruction set to decompose complex instructions into simpler micro-operations that can be more easily implemented in hardware. The microcode details are confidential to the manufacturers,

preventing independent auditing or customization of the microcode. Moreover, microcode patches are signed and encrypted to prevent unauthorized patching and reverse engineering. We present the first framework for static and dynamic analysis of Intel microcode. We reverse-engineer Goldmont microcode semantics and reconstruct the patching primitives for microcode customization. We implement a Ghidra processor module for decompilation and analysis of decrypted microcode, and create a UEFI application that can trace and patch microcode to provide complete microcode control on Goldmont systems. Leveraging our framework, we reverse-engineer the confidential Intel microcode update algorithm and perform the first security analysis of its design and implementation. In three further case studies, we illustrate the potential security and performance benefits of microcode customization. We provide the first x86 Pointer Authentication Code (PAC) microcode implementation and its security evaluation, design and implement fast software breakpoints that are more than 1000x faster than standard breakpoints, and present constant-time microcode division, illustrating the potential security and performance benefits of microcode customization.

The results of this work are presented in:

- P. Borrello, M. Schwarzl. "CustomProcessingUnit: Tracing and Patching Intel Atom Microcode". *BlackHat USA*, 2022.
- P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, M. Schwarz. "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode". *IEEE Workshop on Offensive Technologies* (WOOT), 2023.
- P. Borrello. "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode". *OffensiveCon*, 2023.

This line of research was awarded the 2022 Pwnie Award for *Most Innovative Research*.

To foster future research, we release all our contributions as open source.

While this thesis is based on the six papers presented above, Appendix A reports the complete list of scientific publications the author was involved in, both as main author and non-main author, at the time of writing.

Figure 1.1 provides a pictorial representation of the relationships between the different contributions in this thesis. On the x-axis, we measure the attacker's capability we assume in the corresponding work, from only being able to interact externally with the application, to (unprivileged) local code execution, up to privileged code execution on the machine. On the y-axis, we capture the abstraction that is the target of the attack, from the software to the hardware. Our work starts at the application level, and ends up beyond the CPU abstraction, by targeting CPU microcode.

In Appendix B, we report in Table B.1 the list of open-source software developed, and in Table B.2 the list of vulnerabilities we responsibly disclosed to the vendors while working on the main contributions of this thesis.

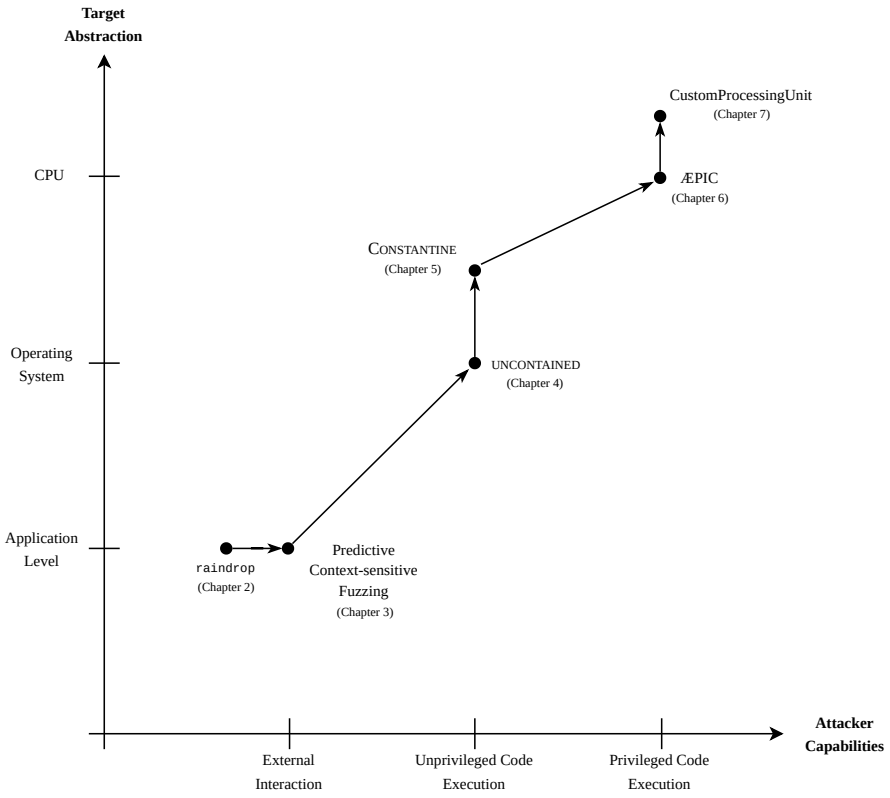


Figure 1.1. Relationship between the different contributions.

Chapter 2

Prevent Reverse Engineering Attacks Using Code Reuse

2.1 Introduction

Memory errors are historically among the most abused software vulnerabilities for arbitrary code execution exploits [365]. Since the introduction of system defenses against code injection attempts, code reuse techniques earned the spotlight for their ability in reassembling existing code fragments of a program to build the execution sequence an attacker desires.

Return-oriented programming (ROP) [323] is the most eminent code reuse technique. Thanks to its rich expressivity, ROP has also seen several uses besides exploitation. Researchers have used it constructively, for instance, in code integrity verification [11], or maliciously to embed hidden functionality in code that undergoes auditing [47, 379]. Security firms have reported cases of malware in the wild written in ROP [133].

Some literature considers ROP code bothersome to analyze: humans may struggle with the exoticism of the representation, and the vast majority of tools used for code understanding and reverse engineering have no provisions for code reuse payloads [39, 95, 148, 392]. Automatic proposals for analyzing complex ROP code started to emerge only recently [95, 148, 392].

We believe that the quirks of the ROP paradigm offer promising opportunities to realize effective code obfuscation schemes. In this chapter we present a protection mechanism that builds on ROP to hide implementation details of a program from motivated attackers that can resort to a plethora of automated code analyses. We analyze what qualities make ROP appealing for obfuscation, and address its weak links to make it robust in the face of an adversary that can symbiotically combine general and ROP-aware code analysis methods.

Motivation

From a code analysis perspective, we observe that the control flow of a ROP sequence is naturally destructured. Each ROP gadget ends with a `ret` instruction that operates like a dispatcher in a language interpreter: `ret` reads from the top

of the stack the address of the next gadget and transfers control to it. The stack pointer RSP becomes a *virtual program counter* for the execution, sidelining the standard instruction pointer RIP, while gadget addresses become the instructions supported by this custom language.

This level of *indirection* makes the identification of basic blocks and of control transfers between them not immediate. This challenges humans and classic disassembly and decompilation approaches, but may not be an issue for dynamic deobfuscation approaches that explore the program state space systematically (e.g., symbolic execution [26]) or try to extricate the original control flow from the dispatching logic (e.g., [392]), nor for ROP-aware analyses that dissect RSP and RIP changes. Protecting transfers is critical for program obfuscations to withstand advanced deobfuscation methods, and we introduce three ROP transformations that address this weak link.

Another benefit from using ROP for obfuscation is the *code diversity* [226] it can bring. Obfuscations may randomize the instructions emitted at specific points, but can incur a limited transformation space [29]. We can use multiple equivalent gadgets in the encoding to serve one same purpose in different program points. But one gadget can also serve different purposes in different points: the instructions in it that concur to the program semantics will depend on the surrounding chain portion, while the others are dynamically dead. This not only complicates manual analysis, but helps also against pattern attacks that may try to recognize specific gadget sequences to deem the location of ROP branches and blocks in the chain.

Such attacks often complement an attacker's toolbox [278]: for instance, an adversary may heuristically look for distinctive instructions in memory and try to patch away parts that hinder semantic attacks. We identify a distinctive benefit of ROP: the adversary only sees bytes that form gadget addresses or data operands, and because of indirection needs to dereference addresses to retrieve the actual instructions. With a careful encoding we can induce *gadget confusion* that makes it harder also to locate the position of gadget addresses in the chain.

Contributions

We bring novel ideas to the software protection realm, presenting a protection mechanism that significantly slows down or deters current automated deobfuscation attacks. We show how to transform entire program functions into ROP chains that interact seamlessly with standard code components, introducing novel natural encoding transformations that raise the bar for general classes of attacks. We evaluate our techniques over synthetic functions for two common deobfuscation tasks, putting the computational effort for succeeding into perspective with different configurations of the prominent virtualization obfuscation [29]. We also analyze their slowdowns on performance-sensitive code, and their coverage on a heterogeneous real-world code base. In summary, over the next sections we present:

- a rewriter that turns compiled functions into ROP chains;
- an analysis of ROP in the face of three attack surfaces for general deobufscation, and three encoding predicates that increase the resistance against such attacks;
- a resistance study for secret finding and code coverage goals with symbolic, taint-driven, and ROP-aware tools;

- a coverage study where we transform 95.1% of the unique functions composing the `coreutils` Linux suite.

In the hope of fostering further work in program protection, we make our system available to researchers. Details for access can be found at https://github.com/pietroborrello/raindrop/.

2.2 Background

This section details key concepts from code obfuscation and ROP research that are relevant to the ideas behind this chapter.

2.2.1 Code Obfuscation

Software obfuscation protects digital assets [342] from malicious entities that some literature identifies as MATE (man-at-the-end) attackers [29]. Before a research community was even born, in the '80s these entities challenged and subverted anti-piracy schemes from vendors, and shielded their own malware.

Today it represents an active research area, with heterogeneous protection mechanisms challenged by increasingly powerful program analyses [310]. Data transformations alter the position and representation of values and variables, while code transformations affect the selection, orchestration, and arrangement of instructions. Our focus are code transformations that prevent an adversary from understanding the program logic.

The interpretation capabilities of an attacker can be syntactic, semantic, or both. This distinction makes a great impact: for instance, instruction substitution or the insertion of spurious computations get in the way of syntax-driven attacks, but may hardly affect a semantic interpretation as we discuss in Section 2.3. When facing mixed capabilities, the most resilient protection schemes are often heavy-duty transformations that deeply affect the control flow and instructions of a program.

Such transformations commonly operate at the granularity of individual functions [29]. *Control-flow flattening* [71, 376] collapses all the basic blocks of the control-flow graph (CFG) into a single layer, introducing a dispatcher block that picks the next block to execute based on an augmented program state. After the successful deobfuscation attack of [355], present variants try to complicate the analysis of the dispatcher [198].

Virtualization obfuscation [29] completely removes the original layout and instructions: it transforms code into instructions for a randomly generated architecture and synthesizes an interpreter for it [204]. The instructions form a bytecode representation in memory, and the interpreter maintains a virtual program counter over it: it reads each instruction and dispatches an *opcode handler* function that achieves the desired semantics for it. As its working resembles a virtual machine, the transformation is also known as VM obfuscation. This technique has lately monopolized the agenda of much deobfuscation research in security conferences (e.g., [39, 80, 309, 325, 391]). VM obfuscation tools have three main strengths: complex code used in opcode handlers to conceal their semantics, obfuscated virtual program counter updates,

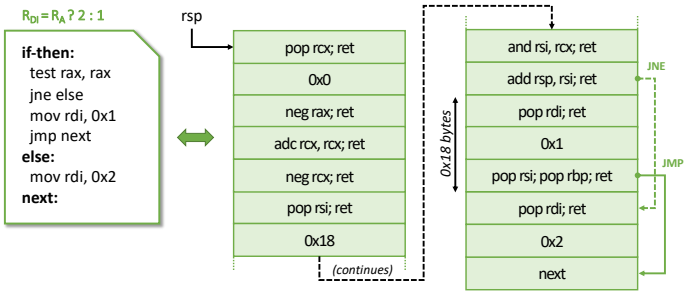


Figure 2.1. ROP chain with non-linear control flow. For readability pointed-to instructions appear in place of gadget addresses.

and scarce reuse of deobfuscation knowledge as the instruction set and the code for opcode handlers are generated randomly for each program.

Best practices often use data transformations at strategic points (e.g., VM dispatcher) in the implementation of a code transformation. The most common instance are *opaque predicates* [77]: expressions whose outcome is independent of their constituents, but hard to determine statically for an attacker. Opaque predicates can build around mathematical formulas and conjectures, mixed boolean-arithmetic (MBA) expressions, and instances of other hard problems like *aliasing* [300].

2.2.2 Return-Oriented Programming

ROP is a technique to encode arbitrary behavior in a program by borrowing and rearranging code fragments, also called gadgets, that are already in the program [323]. Each gadget delivers a piece of the desired computation and terminates with a `ret` instruction, which gives the name to the technique. A ROP payload comprises a sequence of gadget addresses interleaved with immediate data operands. The key to ignition is a pivoting sequence that hijacks the stack pointer, so that on a function return event the CPU fetches the instructions from the first gadget. Each gadget eventually transfers execution to the next using its own `ret` instruction, realizing a ROP chain.

Figure 2.1 features a chain that assigns register RDI with 1 when register RAX==0, and with 2 otherwise. The example showcases exoticisms of the representation with branch encoding and path-dependent semantics of chain items. The first gadget writes the immediate value `0x0` to RCX, and RSP advances by `0x10` bytes for its `pop` and `ret` instructions. The next two gadgets check if RAX is zero with `neg rax`: the carry flag becomes 0 when RAX==0 and 1 otherwise, then an addition with carry writes this quantity into RCX.

ROP control-flow branches are variable RSP addends computed over a leaked CPU condition flag. The chain determines whether to skip over the `0x18` byte-long portion that sets RDI to 1: it computes in RSI an addend that is equal to 0 when RAX==0, and to `0x18` otherwise, using a two's complement and a bitwise AND on RCX. If the branch is taken, RSP reaches a `pop rdi` gadget that reads and assigns 2 to RDI as desired. When execution falls through, a similar sequence sets RDI to 1, then unconditionally jumps over the alternative assignment sequence: this time we find no RSP addition, but a gadget disposes of the alternative $0x10$ byte-long

segment by popping two junk immediates to RSI and RBP.

Attackers can find Turing-complete sets of gadgets in mainstream software [313, 323]. While a few works address automatic generation of ROP payloads [313], publicly available tools often produce incomplete chains in real-world scenarios [14] or do not support branches. Reasons for this failure are side effects from undesired code in found gadgets, register conflicts during chaining [14], and unavailability of "straightforward" gadgets for some tasks [307]. Albeit improved tools continue to appear (e.g., [374]), no general solution for automatic ROP code generation seems available to date.

ROP is the most popular but not the sole realization of code reuse: `jmp`-ended gadgets (JOP) [41], counterfeit C++ objects (COOP) [312], and other elements can be abused as well. But most importantly, ROP today is no longer only a popular mean to get around and disable code injection defenses.

Researchers and threat actors used its expressivity to create userland [133], kernel [169, 371], and enclave [316] malware, and to fool antivirus engines [47, 276] and application review [379]. The sophistication of these payloads went in some cases beyond what a human analyst can manually investigate [148], and researchers in the meantime explored automated approaches to untangle ROP chains: we discuss these works in detail in Section 2.3.

2.3 Adversarial Model

This chapter considers a motivated and experienced attacker that can examine a program both statically and dynamically. The attacker is aware of the design of the used obfuscation, but not of the obfuscation-time choices made when instantiating the approach over a specific program to be protected (e.g. at which program locations we applied some transformation).

While the ultimate end goal of a reverse engineering attempt can be disparate, we follow prior deobfuscation literature (e.g., [27, 28, 278]) in considering two deobfuscation goals that are sufficiently generic and analytically measurable:

G_1 Secret finding. The program performs a complex computation on the input, such as a license key validation, and the attacker wishes to guess the correct value;

G_2 Code coverage. The attacker exercises enough (obfuscated) paths to cover all reachable (original) program code, e.g. to later analyze execution traces.

The attacker has access to state-of-the-art systems suitable for automated deobfuscation and can attempt to *symbiotically* combine them, using one to ease another. In the following we describe the most powerful and promising approaches available to attackers, and enucleate three attack surfaces for general deobfuscation. Those will drive our ROP encoding techniques to build chains that may withstand such attacks.

2.3.1 Principles behind Automated Deobfuscation

Banescu et al. [28] identify a common pre-requisite in automated attacks perpetrated by reverse engineers: the need for building a suite of inputs that exercise the different

paths a protected program can actually take. Achieving a coverage as high as 100% represents G_2 for our attacker, while for G_1 depending on the specific function fewer paths may suffice but also data dependencies should be solved. Slowing down the generation of a "test suite" for the attacker is a first cut of the effectiveness of an obfuscation [28], as it is a key step in most deobfuscation pipelines for utterly disparate end goals.

By analyzing deobfuscation research, we abstract three general attack surfaces that a "good" obfuscation shall consider:

A_1 Disassembly. It should not be immediate for an attacker to discover code portions using static analysis techniques;

A_2 Brute-force search. Syntactic code manipulations such as tracking and "inverting" the direction of control transfers should not reveal new code, but further dependencies must be solved in order to take valid alternate paths;

A_3 State space. When an obfuscation makes provisions to artificially extend the program state space to be explored, analyses based on forward and backward dependencies of program variables should fail to simplify them away.

In the next section we present eminent approaches for such automated attacks, which we then consider in Section 2.7 to evaluate our ROP obfuscation. How to transform an existing program function into a ROP chain and make it robust against these three attack types are the subject of Section 2.4 and Section 2.5, respectively.

2.3.2 State-of-the-art Deobfuscation Solutions

2.3.2.1 General Techniques

Deobfuscation attacks can draw from static and dynamic program analyses. Several static techniques are capable of reasoning about run-time properties of the program, and may be the only avenue when the attacker cannot readily bring execution to a protected program portion or control its inputs. In this context, *symbolic execution* (SE) reveals the multiple paths a piece of code may take by making it read symbolic instead of concrete input values, and by collecting and reasoning on path constraints over the symbols at every encountered branch. Upon termination of each path, an SMT solver generates a concrete input to exercise it [26].

Scalability issues often cripple static approaches, and dynamic solutions may try to get around them by leveraging facts observed in a concrete execution. *Dynamic symbolic execution* (DSE) interleaves concrete and symbolic execution, collecting constraints at branch decisions that are now determined by the concrete input values, and generates new inputs by negating the constraints collected for branching decisions.

Obfuscators can however induce constraints that are hard for a solver, or expand the program state space artificially. Building on the intuition that these transformations are not part of the original program semantics, *taint-driven simplification* (TDS) tracks explicit and implicit flows of values from inputs to program outputs, untangling the control flow of an obfuscation method apart from that of the original program [392]. TDS is a general, dynamic, and semantics-based technique: it applies

a selection of semantics-preserving simplifications to a recorded trace and produces a simplified CFG. TDS can operate symbiotically with DSE to uncover new code by feeding DSE with the simplified trace: in [392] this symbiosis turned out effective in cases that DSE alone could not handle. TDS succeeded on code protected by state-of-the-art VM obfuscators, as well as on four hand-written ROP programs.

We consider SE, DSE, and TDS as they represent powerful tools available to adversaries, and embody concepts seen also in attacks against specific obfuscations (e.g., [390]). Prior literature [27, 28, 278] uses SE and DSE to evaluate and compare obfuscation techniques on goals akin to G_1 and G_2, as both approaches are powerful and driven only by the semantics of the code (i.e., syntactic changes have little effect).

2.3.2.2 ROP-Aware Techniques

Expressing programs as ROP payloads affects analysis techniques that account for the syntactic representation of code. For instance, even commercial disassemblers and decompilers are not equipped to deal with this exotic representation and would fail to produce meaningful outputs for ROP chains [267]. Currently researchers have come up with two solutions to handle complex ROP code.

ROPMEMU [148] attempts dynamic multi-path exploration by looking for sequences that leak condition flags from the CPU status register, as they may take part in branching sequences (Section 2.2.2): it flips their value and tries to generate alternate execution traces that explore new code. ROPMEMU is not the sole embodiment of this technique, seen in, e.g., crash-free binary exploration [290] and malware unpacking [356] research for RIP-driven code. ROPMEMU eventually removes the ROP dispatching logic (i.e., the `ret` sequences) and performs further simplifications, reconstructing a CFG representation.

ROPDissector [95] addresses shortcoming of ROPMEMU in branch identification, with a data-flow analysis for identifying sequences that build variable RSP offsets, so to flip all and only the operations taking part in the process. ROPDissector builds a ROP CFG highlighting branching points and basic blocks in a chain, and operates as a static technique as it does not require a valid execution context as starting point.

In our evaluation we will consider a combination of the two approaches, speculating on extensions tailored to our design.

2.4 Program Encoding with ROP

We design a binary rewriter for protecting compiled programs: the user specifies one or more functions of interest that the rewriter encodes as self-contained ROP chains stored in a data section of the binary. Our implementation supports compiler-generated, possibly stripped x64 Linux binaries. To ensure compatibility between ROP chains and non-ROP code modules, we intercept and preserve stack manipulations and use a separate stack for the chain. This section details the design of the rewriter (Figure 2.2), how it encodes generic functions as self-contained chains, and its present limitations.

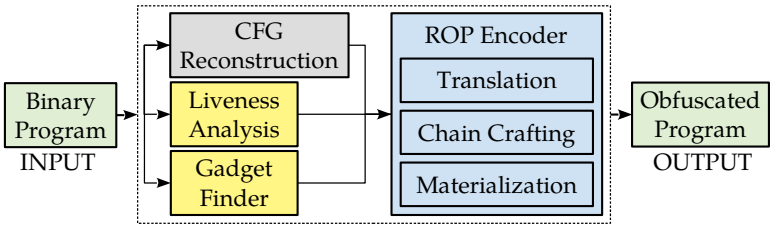


Figure 2.2. Architecture of the ROP rewriter.

2.4.1 Geometry of a ROP Encoder

2.4.1.1 Gadget Sources

The first decision to face in the design of a ROP encoder is where to find gadgets. These may be found in statically and dynamically linked libraries, in program parts left unobfuscated, or in custom code added to the program. We ruled out static libraries as a binary might not have any, and dynamic ones to avoid dependencies on specific library versions that must be present in any target system.

Exploitation research suggests that program code as small as 20-100KB may already contain minimal gadget sets for attacks [313]. Our scenario however is ideal: the possibility of controlling and altering the binary grants us more wiggle room compared to attack scenarios, as we can add missing gadgets—and most importantly create diversified alternatives—as dead code in the `.text` section of the program. We thus pick gadgets from a pool of artificial gadgets combined with gadgets already available in program parts left unobfuscated.

2.4.1.2 Rewriting

The second decision concerns deploying the encoder as a binary rewriter (as we do) or a compiler pass. Binary rewriting can handle a larger pool of programs, including proprietary software and programs with a custom compilation toolchain, and builds on analyses that extract facts necessary to assist the rewriting. A compiler pass has some such analyses (e.g., liveness) already available during compilation, and possibly more control over code shape. However, in order to be able to rewrite an *entire* function, we believe a pass may have to operate as last step (modifying or directly emitting machine instructions) and/or constrain or rewrite several pieces of upper passes (e.g., instruction selection, register allocation). This would lead to a pass that is platform-dependent and that faces similar challenges to a rewriter while being less general.

2.4.1.3 Control Transfers and Stack Layout

Obfuscated functions get expressed in ROP, but may need to interact with surrounding components, calling (or being called by) non-ROP program/library functions or other ROP functions. In this respect native code makes assumptions on the stack layout of the functions, e.g., when writing return addresses or referencing stack objects in the scope of a function and its callees.

Reassembleable disassembling literature [32,103,378,388] describes known hurdles when trying to turn hard-coded stack references into symbols that can be moved around. In our design we instead preserve the original stack behavior of the program: we place the chain in a separate region, and rewrite RSP dereferences and value updates to use a `other_rsp` value that mimics how the original code would see RSP (Figure 2.3).

This choice ensures a great deal of compatibility, and avoids that calls to native functions may overwrite parts of the ROP chain when executing. We keep `other_rsp` in a *stack-switching array* `ss` that ensures smooth transitions between the ROP and native domains and supports multiple concurrently active calls to ROP functions, including (mutual) recursion and interleavings with native calls.

We store the number of active ROP function instances in the first cell of the array, making the last one accessible as `*(ss+*ss)`. When upon a call we need to switch to the native domain, we use `other_rsp` to store the resumption point for the ROP call site, and move its old value in RSP so to switch stacks. Upon function return, a special gadget switches RSP and `other_rsp` again (Figure 2.4).

2.4.1.4 Chain Embedding

Upon generation of a ROP chain, we replace the original function body in the program with a stub that switches the stack and activates the chain. We opt for chains without destructive side effects, avoiding to have to restore fresh copies across subsequent invocations. We place the generated chains at the end of the executable's `.data` section or in a dedicated one.

2.4.2 Translation, Chain Crafting, and Materialization

This section describes the rewriting pipeline we use in the ROP encoder of Figure 2.2. Although we operate on compiled code, the pipeline mirrors typical steps of compiler architectures [4]: we use a number of support analyses (yellow and grey boxes) and *translate* the original instructions to a simple custom representation made of *roplets*, which we process in the *chain crafting* stage by selecting suitable gadgets for their lowering and then allocating registers and other operands. A final *materialization* step instantiates symbolic offsets in the chain and embeds the output raw bytes in the binary.

2.4.2.1 Translation

The unit of transformation is the function. We identify code blocks and branches in it using off-the-shelf disassemblers (*CFG reconstruction* element of Figure 2.2): Ghidra [275] worked flawlessly in our tests when analyzing indirect branches, and we support angr [329] and radare2 [7] as alternatives. We then translate one basic block at a time, turning its instructions into a sequence of roplets.

A *roplet* is a basic operation of one of the following kinds:

- *intra-procedural transfer*, for direct branches and for indirect branches coming from switch tables (Appendix C);

- *inter-procedural transfer*, for calls to non-ROP and ROP functions (including `jmp`-optimized tail recursion cases);
- *epilogue*, for handling instructions like `ret` and `leave`;
- *direct stack access*, when dereferencing and updating RSP with dedicated read or write primitives (e.g., `push`, `pop`);
- *stack pointer reference*, when the original program reads the RSP value as source or destination operand in an instruction, or alters it by, e.g., adding a quantity to it;
- *instruction pointer reference*, to handle RIP-relative addressing typical of accesses to global storage in `.data`;
- *data movement*, for `mov`-like data transfers that do not fall in any of the three cases above;
- *ALU*, for arithmetic and logic operations.

One roplet is usually sufficient to describe the majority of program instructions. In some cases we break them down in multiple operations: for instance, for a `mov qword [rsp+8], rax` we generate a stack pointer reference and a data movement roplet. To ease the later register allocation step, we annotate each roplet with the list of live registers[1] found for the original instruction via *liveness analysis*.

At this stage we parametrically rewire every stack-related operation to use `other_rsp`, and transform RIP-relative addressing instances in absolute references to global storage.

2.4.2.2 Chain Crafting

When the representation enters the chain crafting stage, we lower the roplets in each basic block by drawing from suitable gadgets for each roplet type (using the *gadget finder* element of Figure 2.2). For instance, to translate a conditional (left) or unconditional (right) intra-procedural transfer we combine gadgets to achieve:

```
pop {reg1}  ## L                  ||
mov {reg2}, 0x0                   ||
cmov{ncc} {reg1}, {reg2}          ||   pop {reg1}  ## L
add rsp, {reg1}                   ||   add rsp, {reg1}
```

where a gadget may cover one or more consecutive lines (so we omit `ret` above). In both codes the `pop` gadget will read from the stack an operand `L` (placed as an immediate between the addresses of the first and second gadget) that represents the offset of the destination block. `L` is a symbol that we materialize once the layout of the chain is finalized, similarly to what a compiler assembler does with labels.

Following the analogy, when choosing gadgets for roplets we operate as when in the *instruction selection* stage of a compiler [4], with `{regX}` representing a virtual

[1] A backward analysis deems a register *live* if the function may later read it before writing to it, ending, or making a call that may clobber it [97, 295].

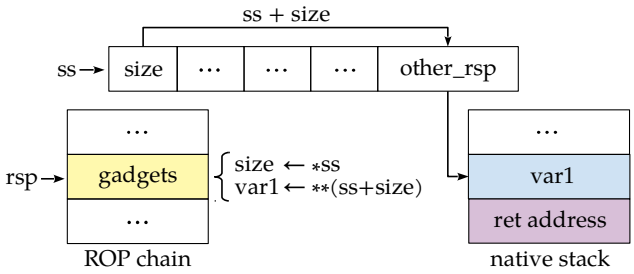


Figure 2.3. Reading a stack variable from top of native stack.

register, roplets the middle-level representation, and gadgets the low-level one. When it comes to *instruction scheduling*, we follow the order of the original instructions in the block.

Native function calls see a special treatment, as we have to switch stacks and set up the return address in a way to make another switch and resume the chain (Section 2.4.1). For the call we combine gadgets as in the following:

```
pop {reg1}  ## ss
add {reg1}, qword ptr [{reg1}]  ## step A ends
sub qword ptr [{reg1}], 0x8
mov {reg2}, qword ptr [{reg1}]
pop {reg3}  ## addr. of return gadget
mov qword ptr [{reg2}], {reg3}  ## step B ends
pop {reg2}  ## function address
xchg rsp, qword ptr [{reg1}]; jmp {reg2} ## step C
```

where we pop from the stack the addresses of: the stack-switching array, a *function-return gadget*, and the function to call. Gadgets may cover one or more consecutive lines, except for the last one which describes an independent single JOP gadget (Section 2.2.2): `xchg` and `jmp` switch stacks and jump into the native function at once. Figure 2.4 shows the effects of the three main steps carried by the sequence.

The called native function sees as return address (top entry of its stack frame) the address of the function-return gadget. This is a synthetic gadget with a statically hard-wired `ss` address that reads the RSP value saved by the `xchg` at call time and swaps stacks again:

```
mov {reg1}, ss; add {reg1}, qword ptr [{reg1}];
xchg rsp, qword ptr [{reg1}]; ret
```

For space limitations we omit details on the lowering of other roplet types: their handling becomes ordinary once we translated RSP and RIP-related manipulations (Section 2.4.2.1).

Register allocation is the next main step: we choose among candidates available for a desired gadget operation by taking into account the registers they operate on and those originally used in the program, trying to preserve the original choices whenever possible. When we find conflicts that may clobber a register, we use scratch registers when available (i.e., non-live ones) or spill it to an inlined 8-byte chain

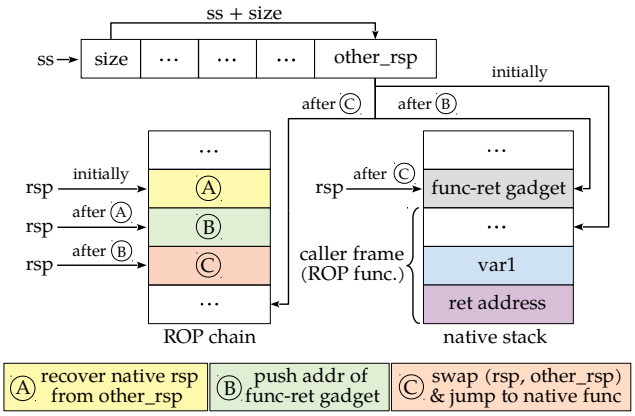


Figure 2.4. Call to a native function from ROP code.

slot as a fallback. We then ensure a *reconciliation of register bindings* [243] at the granularity of basic blocks: when execution leaves a block, the CPU register contents reflect the expected locations for program values that are live in the remainder of the function.

Another relevant detail is to preserve the status register if the program may read it later. While most instructions alter CPU flags, our liveness analysis points out the sole statements that may concur to a later read: whenever in between we introduce gadgets that pollute the flags[2], we spill and later restore them.

2.4.2.3 Materialization

At the end of the crafting stage a chain is almost readily executable. As its branching labels are still symbolic, we may optionally rearrange basic blocks: then once we fix the layout the labels become concrete RSP-relative displacements. We then embed the chain in the binary, allocating space for it in a data section and replacing the original function code with a pivoting sequence to the ROP chain. The sequence extends the stack-switching array and saves the native RSP value, then the chain upon termination executes a symmetric unpivoting scheme (details in Appendix C).

2.4.3 Discussion

Our design makes limited assumptions on the input code: it hinges on off-the-shelf binary analyses to identify intra-procedural branch targets, and obliviously translates stack accesses and dereferences to preserve execution correctness when interacting with the surrounding software stack.

Our implementation could rewrite a large deal of real-world programs (Section 2.7.3.1), even when we supplied it code already protected by the control-flow flattening and/or (nested) VM obfuscations of the Tigress framework [76]. We experimentally observed (Section 2.7.3.1) that the analyses of Ghidra are remarkably effective in recovering intra-procedural indirect branch targets, which in several high-level languages derive from optimized switch constructs. Whenever those may fail, one could couple the rewriter with a dynamic tracer for recovering the intended targets by running the original program using expected inputs. Transfers to other

functions via indirect calls or tail jumps are instead straightforward, as the chain transfers control to the prologue of the callee as it happens with direct calls.

A limitation of the design, shared with static rewriting and instrumentation schemes [94, 103], is lastly the inability to handle self-modifying and dynamically generated code.

As for register conflicts, the high number of x64 registers give us wiggle room to perform register renaming within blocks with modest spilling. An area larger than the 1-word one we use may help with code with very high register pressure cases (Section 2.7.3) or 32-bit implementations; instruction reordering and function-wide register renaming may also help.

The implementation incurs two main limitations that one can address with moderate effort. The spilling slots and the `ss` array area are not thread-private, but we may recur to thread-local storage primitives. Rewritten binaries are compatible with address space layout randomization for libraries, while the body of the program is currently loaded at fixed addresses. To ship position-independent executables we may add relocation information to headers so to have the loader patch gadget addresses in the chains, or use the online patching for chains from [47] to have the program itself do the update.

In terms of compatibility with ROP defenses of modern operating systems, our context is different to an exploitation one where the program stack gets altered and the choice of gadgets is limited. On Windows, for instance, our stack switching upon API calls would already comply with the RSP range checks of *StackPivot* [273]; our liberty to synthesize gadgets would be decisive against *CallerCheck*, which checks if the instruction preceding an API's return address is a call [273]. We refer to prior work [47] for details. A potential issue, which may require the user to whitelist the program, could be instead coarse-grained defenses that monitor branches [91] or micro-architectural effects [120]. However, those are yet to become mainstream as they face robustness and accuracy issues.

Finally, our readers may question if the use of ROP introduces obvious security risks. An attacker needs a write primitive pointing to a chain in order to alter it. In our protected programs, ROP-encoded parts use write operations only for spilling slots, and those cannot go out of bounds. Non-ROP parts never reference chains in write (or read) operations: an attacker would thus have to search for an arbitrary memory write primitive in such parts. Its presence, however, would be an important source of concern even for the original program. Our implementation also supports the generation of read-only chains, which use a slightly longer spilling machinery.

2.5 Strengthening ROP Programs

In the furrow of prior works (e.g., [39, 148, 392]) that highlighted the hindrances from the ROP paradigm to reverse engineering attempts, one could anticipate that the design of Section 2.4 may challenge manual deobfuscation and code understanding attempts. The common thread of their observations is that the exoticism of the representation—ROP defines a *weird machine* [225]—disturbs humans when com-

[2]This happens mainly when an intervening instruction involves RSP and we emit gadgets to do pointer arithmetic for stack pointer reference roplets.

pared to native code. The rewriter makes use of all motivating factors for ROP that we outlined in Section 2.1, such as destructured control flow and diversity and reuse of gadgets (including gadget confusion that we describe next).

Quantifying the effectiveness of an obfuscation is however a difficult task, as it depends not only on the available tools, but also on the knowledge of the human operating them [28]. A well-established practice in the deobfuscation literature is to measure the resilience to automated deobfuscation techniques, which in most attack scenarios are the fulcrum of reverse engineering attempts and ease subsequent manual inspections [310].

ROP encoding alone is not sufficient for obfuscation. We find control transfers between basic blocks to be its weak link.

Even when diversifying the used gadget instances, an attacker aware of the design may follow the ROPMEMU approach (Section 2.3.2) to spot in an execution trace what gadgets add variable quantities to RSP (thus exposing basic blocks), untangle `ret` instructions from the original control flow of the program, and assemble a dynamic CFG from multiple traces.

Protecting control transfers is equally critical in the face of the most effective general-purpose semantics-aware techniques like SE, DSE and TDS, which try to reason on the parts essential for program functionality while sifting out the irrelevant obfuscation constructs and instructions [28,392], such as side effects and dynamically dead portions from gadgets.

One way to hinder the automated approaches of Section 2.3.2 would be to target weaknesses of each technique individually. For instance, researchers proposed hard-to-solve predicates for SE (e.g. MBA expressions [36], cryptographic functions [324]), and code transformations that impact concolic variants like DSE too [391]. But an experienced attacker can symbiotically combine methods to defeat this approach, for instance using TDS or similar techniques (e.g., program synthesis for MBA predicates [36]) to feed DSE with tractable traces as in [392].

In this section instead we present three rewrite predicates, naturally meshed with RSP update actions, that bring protection against generic, increasingly powerful automated attacks that cover the principled classes $\mathsf{A}_{1\text{-}2\text{-}3}$ from Section 2.3.1. We then introduce gadget confusion and share some general reflections.

2.5.1 Predicate P_1: Anti-ROP-Disassembly

Our first predicate uses an array of opaque values [78] to hide branch targets (A_1). The array contains seemingly random values generated such that a periodic invariant holds, and backs the extraction of a quantity a that we use to compute the displacement in the chain for one of the n branches in the code. Suppose we need to extract a for branch $b \in \{0..n-1\}$: starting with cell b, in every p-th cell of the array we store a random number q such that $q \equiv a \bmod m$, with $m > n$ and p chosen at obfuscation time.

10	19	34	45	54	62	66	33	6	59	61	20

Above we encoded information for n=3 branches using p=4 repetitions and m=7. For the branch with ordinal 1 we wanted to memorize $a = 5$: every cell colored in dark gray thus contains a value v such that $v \bmod 7$ equals 5.

During obfuscation we use a period of size $s > n$, with a fraction of the cells containing garbage. We also share a valid cell among multiple branches, so to avoid encoding unique offsets that may aid reversing. To this end we divide an RSP branch offset δ in a fixed part a encoded in the array and a branch-specific part $\delta - a$ computed by the chain, then we compose them upon branching.

This implies that for static disassembly an attacker should recover the array representation and mimic the computations made in every chain segment to extract a and compute the branch-specific part. While this is possible for a semantically rich static technique like SE, periodicity comes to the rescue as it brings *aliasing*: every p-th cell is suitable for extracting a. Our array dereferencing scheme takes the form of:

$$a = A[f(x) * s + n] \; mod m$$

where $f(x)$ depends on the program state and returns a value between 0 and $p - 1$. Its implementation opaquely combines the contents of up to 4 registers that hold input-derived values. SE will thus explore alternative input configurations that ultimately lead to the same rsp $+= \delta$ update; reducing their number by constraining the input would lead instead to missing later portions of program state.

Whenever an attacker may attempt a *points-to* analysis [330] over rsp $+= \delta$, we believe a different index expression based on user-supplied or statically extracted facts on input value ranges would suffice to complicate such analysis significantly.

2.5.2 Predicate P_2: Preventing Brute-Force Search

Our second predicate introduces artificial data dependencies on the control flow, hindering dynamic approaches for brute-force path exploration (A_2) that flip branches from an execution trace. While these techniques do not help in secret finding (G_1) as they neglect data constraints (Section 2.3.1), they may be effective when the focus is code coverage (G_2).

P_1 is not sufficient against A_2: an attacker can record a trace that takes a conditional branch shielded by P_1, analyze it to locate the flags set by the instruction that steered the program along the branch, flip them, and reveal the other path [95].

Without loss of generality, let us assume that a *cmp a, b* instruction determines whether the original program should jump to location `L` when $a == b$ and fall through otherwise. We introduce a data dependency that breaks the control flow when brute-force attempts leave its operands untouched. As we translate the branch in ROP, in the block starting at `L` we manipulate RSP with, e.g., $rsp \mathrel{+}= x * (a - b)$, so that when brute-forcing it without changing the operands, $(a - b) != 0$ and RSP flows into unintended code by some offset multiple of x. Similarly, on the fall-through path we manipulate RSP with, e.g., $rsp \mathrel{+}= x * (1 - notZero(a - b))$, where *notZero* is a flag-independent computation[3] so the attacker cannot flip it.

Different formulations of opaque updates are possible. Whenever an attacker may attempt to learn and override updates locally, we figured a future, more covert P_2 variant that encodes offsets for branches using opaque expressions based on value invariants (obtainable via value set analysis [25]) for some variable that is defined in an unrelated CFG block.

2.5.3 Predicate P_3: State Space Widening

Our third predicate brings a path-oriented protection that artificially extends the program space to explore and is coupled with data (and optionally control) flows of the program, so that techniques like TDS (A_3) cannot remove it without knowledge of the obfuscation-time choices. P_3 comes in two variants.

The first variant is an adaptation of the FOR predicate from [278]. The idea is to introduce state forking points using loops, indexed by input bytes, that opaquely recompute available values that the program may use later. In its simplest formulation, FOR replaces occurrences of an input value `char c` with uses of a new `char fc` instantiated by `for (i=0; i<c; ++i) fc++`. Such loop introduces 2^8 artificial states to explore due to the uncertainty on the value of `c`.

The work explains that targeting 1-byte input portions brings only a slight performance overhead, and choosing independent variables for multiple FOR instances optimizes composition for state explosion. It also argues how to make FOR sequences resilient to pattern attacks, and presents a theorem for robustness against taint analysis and backward slicing, considered for forward and backward code simplification attacks, respectively (the TDS technique we use has provisions for both [392]).

While we refer to it for the formal analysis, for our goals suffice it to say that when the obfuscated variable is input-dependent (for tainting) and is related to the output (for slicing), such analyses cannot simplify away the transformation.

During the rewriting we use a data-flow analysis to identify which live registers contain input-derived data (*symbolic* registers) and may later concur to program outputs[4]. We then introduce value-preserving opaque computations like in the examples below (the right one is adapted from [278]):

```
// clear last byte
dead_reg &= 0xAB00;
for (i=0; i<(char)sym; ++i)
  dead_reg++;
sym |= (char)dead_reg;
```

```
dead_reg = 0;
for (i=0; i<(char)sym; ++i)
   if (i%2) dead_reg-;
   else dead_reg+=3;
if (i%2) dead_reg-=2;
sym = (sym&0xF..F00)+dead_reg;
```

These patterns significantly slow down SE and DSE engines, but also challenge approaches that feed tractable simplified traces to DSE. While one may think of detecting and propagating constant values in the trace, the TDS paper [392] explains that doing it indiscriminately may oversimplify the program: in our scenario it may remove FOR but also pieces of the logic of the original program elsewhere. To avoid oversimplification the TDS authors restrict constant propagation across input-tainted conditional jumps, which is exactly the case with `dead_reg` and `sym` in the examples above.

The authors suggest, as a general way to hamper semantics-based deobfuscation approaches like TDS, to deeply entwine the obfuscation code with the original input-to-output computations. They also state that at the time obfuscation tools had not explored this avenue, possibly for the difficulties in preserving observable program behavior [392].

Our second $\mathsf{P_3}$ variant is new and moves in this direction. Instead of recomputing input-derived variables, we use them to perform *opaque updates* to the array used by $\mathsf{P_1}$. Updates include adding/subtracting quantities multiple of m, swapping the contents of two related cells from different periods, or combining the contents of two cells i and j where $a \equiv A[i] \bmod m$ and $b \equiv A[j] \bmod m$ to update a cell l where $(a+b) \equiv A[l] \bmod m$. For DSE-alike path exploration approaches the effect is tantamount to the FOR transformation described above. For trace simplification it introduces implicit flows, with *fake control dependencies* between program inputs and branch decisions taken later in the code: TDS cannot simplify them without explicit knowledge of the invariants.

2.5.4 Gadget Confusion

ROP encoding brings several advantages when implementing $\mathsf{P_{1\text{-}2\text{-}3}}$. Firstly, it offers significant leeway for diversifying the gadget instances we use to instantiate them. We combine this diversity with *dynamically dead* instructions: we can use gadgets whose each instruction either concurs to implementing a predicate or has no effect depending on the surrounding chain portion. This helps in instantiating many variants of a pattern, challenging syntactic attacks aware of the design.

However, a unique advantage of ROP, as we observed in Section 2.1, is the level of indirection that it brings: this complicates pattern attacks that look for specific instruction bytes, since code is not in plain sight, and attackers need to extract the instruction sequences as if executing the program. What they see are bytes belonging to either gadget addresses or data operands. They may, however, attempt analyses that look for byte sequences resembling addresses from code regions (i.e., plausible gadgets) and try to speculatively execute the chain from there [95, 294]. By trying it at every plausible point, this may eventually reveal some chain portions, nonetheless short thanks to $\mathsf{P_{1\text{-}2}}$.

This is when *gadget confusion* enters the picture. Firstly, we can transform data operands in the chain to look like gadget addresses, having then gadgets recover the desired values at run time (e.g., subtracting two addresses to obtain a constant, applying bitmasks, shifting bits, etc.). This is possible as we control both the layout of the binary (for the addresses) and the pool of artificial gadgets (for the manipulations). Now that virtually every 8-byte chain stride looks like a gadget address, we introduce unaligned RSP updates at random program points, adding a quantity η s.t. $\eta \bmod 8$!= 0. In the end, the attacker may have to execute speculatively at every possible chain offset, obtaining instructions that may or may not be part of the intended execution sequence. We believe such gadget confusion makes pattern attacks on our chains even harder.

2.5.5 Further Remarks

The instantiation of $\mathsf{P_{1\text{-}2\text{-}3}}$ is naturally entwined with RSP dispatching: directly for $\mathsf{P_{1\text{-}2}}$, and indirectly for $\mathsf{P_3}$ through array updates. In the rewriter, $\mathsf{P_1}$ replaces the RSP update sequence we showed in Section 2.4.2.2, while $\mathsf{P_2}$ operates on the

[3] Example: $notZero(n) := \sim(\sim n \,\&\, (n + \sim 0)) \gg 31$ for 32-bit data types.
[4] To this end we use the symbolic execution capabilities of angr [329].

fall-through and target blocks of a branch. Finally, the rewriter can apply either P_3 variant to a user-defined fraction k of the original program points when lowering the associated roplets.

Each predicate targets a main attack surface, but positive externalities are also present. P_2 can protect against possible linear/recursive disassembly algorithms for ROP (A_1), but will not withstand SE-based disassembly. In Section 2.7.3 we discuss how P_1 can slow down state exploration (A_3) by indirectly putting pressure on the memory model of a SE or DSE engine.

Finally, with the second P_3 variant we used ROP control transfer dynamics to introduce also fake control dependencies.

2.6 Related Work

Prior research explored ROP for software protection goals orthogonal to obfuscation: tamper checking of selected code regions through chains that use gadgets from such regions [11], covert watermark encoding [246], and steganography of short code [242]. Each of them could complement our design, especially [11] for checking code integrity of non-obfuscated parts.

ROPOB [267] is a lightweight obfuscation method to rewrite transfers between CFG basic blocks using ROP gadgets. It considers standard disassembly algorithms as adversary (a "lighter" A_1 case), and does not withstand static attacks like SE (A_1) or ROPDissector (A_2), nor dynamic ones like DSE or TDS (A_3). ROPOB leaves data manipulation instructions in plain sight, whose rewriting poses several challenges (Section 2.4.2).

VM deobfuscation attacks like Syntia [39] and VMHunt [390] intercept and simplify (A_3) dispatching and opcode handling sequences. They do not apply directly to ROP chains, and embody flavors of the agnostic and general approach of TDS.

movfuscator [72] is an extreme instance of the weird machine concept, rewriting programs using only the Turing-complete `mov` instruction. Kirsch et al. present [206] a custom linear-sweep algorithm (A_1) that recovers the CFG by targeting logic dispatching elements used for the very encoding.

2.7 Evaluation

We arrange our experimental analysis in three parts. We first study the efficacy of our techniques against prominent solutions for $\mathsf{A}_{1\text{-}2\text{-}3}$ (Section 2.7.1), confirming the theoretical expectations. We then study the resource usage of viable deobfuscation attacks using a methodology adopted in previous works [28, 278], and put such numbers into perspective with VM-obfuscated[5] counterparts (Section 2.7.2). Finally, we analyze the applicability of our method to real-world code (Section 2.7.3).

We ran the tests on a Debian 9.2 server with two Xeon E5-4610v2 and 256 GB of RAM. Appendix Appendix C contains the settings we used to generate our 72 test functions and the VM variants with Tigress, and more implementation details. The rewriter currently consists of ~3K Python LOC.

Table 2.1. Terminology for obfuscation configurations.

SETTING	DESCRIPTION
ROP_k	ROP obfuscation with $\mathsf{P_3}$ inserted at a fraction of program points $k \in \{0, 0.05, 0.25, 0.50, 0.75, 1.00\}$ and with $\mathsf{P_1}$ instantiated with n=4, s=n, p=32
nVM	n layers of VM obfuscation with $n \in \{1, 2, 3\}$
$n\text{VM-IMP}_x$	n layers of VM obfuscation with implicit flows used for every VPC load at layer(s) $x \in$ {first, last, all}

Table 2.1 details configuration naming for the main ROP and VM experiments. For the latter we try multiple layers of nested virtualization as this is known to slow down SE and DSE-based attacks [278, 309], and use a Tigress predicate that adds implicit flows to virtual program counter (VPC) loads: those frustrate taint analysis-based simplifications and also create many redundant states whenever VPC becomes symbolic.

2.7.1 Efficacy of ROP Strengthening Transformations

The techniques presented in Section 2.5 should intuitively raise the bar to existing automated attacks, and hinder symbiotic combinations between them. We now study how each automated approach feels the effects of each technique individually already on small program instances, discussing also design-aware enhancements we tried for ROP tools. In the end, DSE emerges as the one and only viable option for our attacker.

We leverage the Tigress framework [76] to generate functions appropriate as reverse engineering targets with a desired complexity and structure. Tigress will also annotate CFG split and join points with probes to help us measure code coverage.

2.7.1.1 General Attacks

In the context of general-purpose automated attacks, we consider angr [329] as SE engine, S2E [70] for DSE, and the TDS implementation released by its authors. Let us start with SE. For $\mathsf{P_1}$ we consider a function with control structure [76] `for (if (bb 4) (bb4))` having 4 mathematical computations per block, 15 loop iterations, and a single `int` as input. In a "ROP-P_1" version we encode in the array for $\mathsf{P_1}$ n=4 δ-offsets, with no garbage entries ($s = n$) and $p = 32$ repetitions, for a total of 128 cells populated statically.

To explore enough paths to hit all coverage points ($\mathsf{G_2}$), angr took a time in the order of seconds for the native function, and over 4500 seconds for ROP-P_1. The aliasing $\mathsf{P_1}$ induces on RSP updates for branching slows angr down significantly

[5] We do not consider commercial tools like VMProtect for two reasons: they offer little control over the transformations (but may rather combine many at once), and add tricks and bombs [392] to break deobfuscation solutions by targeting implementation gaps instead of their methodological shortcomings.

already for little code, as the SMT solver sees increasingly complex expressions over RSP. Aliasing reverberates on secret finding (G_1) too: with a simpler `for (for (bb 4))` code, angr cracked the secret in the order of seconds for the original code, and over 5 hours for ROP-P_1. Other configurations of variable complexity confirmed these trends. When we tested P_3 shielding a single program point per basic block, 24 hours were not sufficient for angr to crack the secret. These results suggest SE may not be readily suitable against our approach.

As for DSE, in the experiments P_1 impacted it slightly and only for G_2: the reason is that S2E benefits from concrete input values when picking the next path to execute. For P_3 we obtained two confirmations: its two variants bring similar time increases, and while higher k fractions of shielded program points inflate the state space possibly more, code with small input space may not always offer sufficient independent sources (i.e., symbolic registers) for optimal composition of P_3 instances. We postpone a detailed analysis of the induced overheads to Section 2.7.2 as we consider larger code instances.

P_1 and P_3 resist TDS by design. The tested output traces kept non-simplifiable (Section 2.5.3) implicit control dependencies from having a tainted input value determine a jump target: as those are pivotal to put pressure on DSE, combining DSE with TDS-simplified input traces [392] would not ease attacks.

Summarizing, P_1 and P_3 effectively raise the bar for A_1 and A_3 attacks, respectively: SE and TDS look no longer useful already for little code. P_2 and gadget confusion target syntactic approaches, unlike the semantics-aware attacks we considered above: we address them next in the ROP-aware domain.

2.7.1.2 ROP-Aware Attacks

To analyze ROP payloads we use and extend ROPDissector to start from a memory dump of the program taken when entering the chain of interest: in this configuration it operates as a hybrid static-dynamic analysis and surpasses ROPMEMU in branch analysis and flipping capabilities. With ROPDissector now embodying a full-fledged ROP-A_2 approach, we test if it can help with G_2, while G_1 is out of scope as A_2 recovers code but neglects data constraints.

Backing our expectations, shielding branches with P_2 in the rewriting makes ROPDissector fail in revealing any basic block other than those the input used for the test reveals. We tried to further extend ROPDissector by using its gadget guessing technique (a ROP-educated form of pattern matching [95]) to reveal new blocks by executing the chain at different start offsets. Our gadget confusion however makes such analysis explode, with many short and unaligned candidate blocks that are difficult to distinguish from P_2-protected true positives.

We conclude this part by stressing the importance of conceiving all of our protections. P_1 impacts ROPDissector only if no dump is supplied, and P_3 does not affect it directly. Hence, without P_2 an attacker could have used ROPDissector or a similar tool to aid semantic attacks in code coverage scenarios.

Table 2.2. Successful attacks in the 1h-budget per program.

Obfuscation Configuration	Secret Finding		Code Coverage
	Found	Avg Time	100% Points
Native	70/72	65.2s	72/72
$\text{ROP}_{0.05}$	19/72	907.9s	34/72
$\text{ROP}_{0.25}$	10/72	568.4s	11/72
$\text{ROP}_{0.50}$	9/72	884.0s	9/72
$\text{ROP}_{0.75}$	5/72	775.3s	7/72
$\text{ROP}_{1.00}$	1/72	3028.7s	6/72
1VM-IMP$_{all}$	61/72	85.8s	68/72
2VM	62/72	71.6s	67/72
2VM-IMP$_{first}$	62/72	100.4s	66/72
2VM-IMP$_{last}$	61/72	104.1s	65/72
2VM-IMP$_{all}$	62/72	160.6s	64/72
3VM	62/72	119.2s	69/72
3VM-IMP$_{first}$	54/72	899.2s	56/72
3VM-IMP$_{last}$	62/72	240.3s	61/72
3VM-IMP$_{all}$	0/72	-	0/72

2.7.2 Measuring Obfuscation Resilience

We now measure the amount of resources required to carry automated attacks for secret finding (G_1) and code coverage (G_2) over synthetic functions from an established methodology. We ask Tigress to generate 72 non-cryptographic hash functions with 6 control structures analogous to the most complex ones from an influential obfuscation work [27], input sizes of $\{1, 2, 4, 8\}$ bytes, and three seeds (details in Appendix C).

We exclude techniques that were ineffective on smaller inputs like SE, and restrict our focus to DSE (recently [278] makes a similar choice). DSE allows us to set up controlled and accurate experiments for measuring G_1 and G_2, as S2E typically succeeds in either goal in about one minute for each of the 72 functions. This makes measuring obfuscation overheads feasible, with a 1-hour budget per experiment sufficient to capture a slowdown of ∼50x or higher. With 15 configurations and 2 goals, the tests took > 2000 CPU hours.

In light of all the considerations made in Section 2.7.1, we use a ROP_k setup with P_1 and P_3 enabled (P_2 and gadget confusion are disabled as they do not affect DSE), with the same $\{s, n, p\}$ settings mentioned there for P_1, and with P_3 instantiated in its first variant and applied at different fractions k of program points (Table 2.1). As state exploration strategy for S2E we use class-uniform path analysis [57] as it consistently yielded the best results across all ROP and VM configurations: its state grouping seems to work effectively for reducing bias towards picking hot spots involved in path explosion, which could be the case with P_3 instances under other strategies.

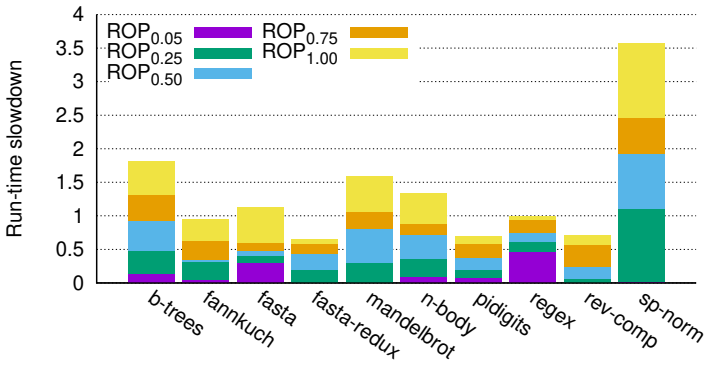


Figure 2.5. Run-time overhead for `clbg` benchmarks of different ROP_k settings with 2VM-IMP$_{last}$ used as baseline.

2.7.2.1 Secret Finding

Column two and three of Table 2.2 summarize the results for the differently obfuscated configurations: for each class we report for how many functions S2E found the secret, and the average time for successful attempts. For 2 of the 72 non-obfuscated functions S2E failed also with a 3-hour budget, likely due to excessively complex path constraints.

Coherently with insights from previous works [278, 309], applying one or two layers of VM obfuscation does not prevent S2E from solving the majority of the secrets (the same sets of 61-62 functions over 72) even when using implicit VPC loads[6], with average overheads as high as 1.6x when applied to either the inner or the outer VPC, and 2.46x when to both. For 3VM implicit VPC loads are significantly more effective in slowing down S2E when applied on the innermost VPC than on the outermost one, while when used at all the three layers S2E found zero secrets within the 1-hour budget.

The fraction of successful attacks to ROP_k is lower than for VM configurations already for $k = 0.05$, except for 3VM-IMP$_{all}$ that however, as we see in Section 2.7.3.2, may bring a destructive impact on program running time. The fraction of ROP_k-protected functions that S2E can crack decreases with k: while we cannot compare average times computed for different sets, individual figures reveal that S2E can crack only the simpler functions as k increases, with a higher processing time compared to when they were cracked for a smaller k.

2.7.2.2 Code Coverage

The last column of Table 2.2 lists for how many functions S2E covered all the CFG split and join points annotated by Tigress and reachable in the native counterparts (as the functions are relatively small, we consider coverage an "all or nothing" goal like in [28,278]). As seen in Section 2.3.1, we recall that secret finding may not require full coverage (neither achieving G_2 is sufficient for G_1). For most VM configurations,

[6]We do not report data for 1VM and $\text{ROP}_{k=0}$ programs since S2E breaks them with no appreciable slowdown w.r.t. their non-obfuscated counterparts.

the functions for which S2E fully explored the original CFG are slightly more than those for which it recovered the secret. ROP_k already for $k = 0.05$ impedes achieving G_2 for nearly half of the functions, and leaves only a handful (6-11) within the reach of S2E for higher k values.

2.7.3 Deployability

To conclude our evaluation, we investigate how our methods can cope with real-world code bases in three respects: efficacy of the rewriting, run-time overhead for CPU-intensive code, and an obfuscation case study on a popular encoding function.

2.7.3.1 Coverage

We start by assessing how our implementation can handle the code base of the `coreutils` (v8.28, compiled with gcc 6.3.0 `-O1`). Popular in software testing, this suite is a suitable benchmark thanks to its heterogeneous code patterns. Using symbol and size information, we identify 1354 unique functions across its corpus of 107 programs. We skip the 119 functions shorter than the 22 bytes the pivoting sequence requires[7] (Section 2.4.2.3). Our rewriter could transform 1175 over 1235 remaining functions (95.1%, or a 0.801 fraction if normalized by size). 40 failures happened during register allocation as one spilling slot was not enough to cope with high pressure (Section 2.4.3), 19 for code like `push rsp` and `push qword [rsp + imm]` that the translation step does not handle yet (Section 2.4.2.1), and 1 for failed CFG reconstruction.

As informal validation of functional correctness, we run the test suite of the `coreutils` over the obfuscated program instances, obtaining no mismatches in the output they compute.

2.7.3.2 Overhead

Albeit a common assumption is that heavy-duty obfuscation target one-off or infrequent computations, we also seek to study performance overhead aspects. We consider the `clbg` suite [146] used in compiler research to benchmark the effects of code transformations (e.g., [63, 96]). As a reference we consider 2VM-IMP_{last} as it was the fastest configuration for double virtualization with implicit VPC loads (1VM is too easy to circumvent, and 3VM brings prohibitive overheads, i.e., over 5-6 orders of magnitude in our tests).

Figure 2.5 uses a stacked barchart layout to present slowdowns for ROP_k, as its overhead can only grow with k. With the exception of `sp-norm` that sees repeated pivoting events from a ROP tight loop calling a short-lived ROP subroutine, ROP_k is consistently faster than 2VM-IMP_{last} for $k \leq 0.5$, and no slower than 1.81x (`b-trees` that repeatedly calls `malloc` and `free`) when in the most expensive setting $k = 1.00$.

2.7.3.3 Case Study

Finally, we study resilience and slowdowns of selected obfuscation configurations on the reference implementation of the popular base64 encoding algorithm [354]. base64 features byte manipulations and table lookups relevant for transformation code of

variable complexity that users may wish to obfuscate. An important consideration is that in the presence of table lookups, using concrete values for input-dependent pointers is no longer effective (but even counter-productive) for DSE to explore relevant states. We thus opt for the per-page theory-of-arrays ([26, 50]) memory model of S2E. This choice allows S2E to recover a 6-byte input in about 102 seconds for the original implementation, 180 for 2VM-IMP$_{last}$, 281 for 2VM-IMP$_{all}$, and 1622 for 3VM-IMP$_{last}$.

A budget of 8 hours was not sufficient for 3VM-IMP$_{all}$, as well as for ROP$_k$ already for $k=0$ (when only P_1 is enabled). As anticipated in Section 2.5.5, the aliasing from P_1 on RSP updates can impact the handling of memory in DSE executors in ways that the synthetic functions of Section 2.7.2 did not (as they do not use table lookups). As for code slowdowns, ROP$_k$ seems to bring rather tolerable execution times: for a rough comparison, execution takes 0.299ms for ROP$_{0.25}$ and 1.791ms for ROP$_{1.00}$, while for VM settings we measured 1.63ms for 2VM-IMP$_{last}$, 347ms for 2VM-IMP$_{all}$, 668ms for 3VM-IMP$_{last}$ and 2211s for the unpractical 3VM-IMP$_{all}$.

2.8 Conclusion

Adding to the appealing properties of ROP against reverse engineering that we discussed throughout the chapter, the experimental results lead us to believe that our approach can:

1. hinder many popular deobfuscation approaches, as well as symbiotic combinations aimed at ameliorating scalability;

2. significantly increase the resources needed by automated techniques that remain viable, with slowdowns $\geq$ 50x for the vast majority of the 72 targets for both end goals $\mathsf{G}_{1\text{-}2}$;

3. bring multiple configuration opportunities for resilience (and overhead) goals to the program protection landscape.

While obfuscation research is yet to declare a clear winner and automated attacks keep evolving, our technique is also orthogonal to most other code obfuscations, meaning it can be applied on top of already obfuscated code (Section 2.4.3). We have followed established practices [29] of analyzing our obfuscation individually and on function units, yet in future work we would like to expand both points: namely, studying mutually reinforcing combinations with other obfuscations, and applying ROP rewriting inter-procedurally, removing the stack-switching step during transfers between ROP functions, since our design allows that. Finally, to optimize composition of symbolic registers when instantiating P_3, we may look at def-use chains as suggested by [278] for FOR cases, exploring analyses like [25] necessary to obtain the required information.

[7] While we could have added a trampoline to some code cavity large enough to hold it, these functions appear to be stubs of unappealing complexity.

This chapter explored how to strengthen software obfuscation with low overhead. While this effectively slows down attackers with no access to source code, given enough time or access to source code, they may still be able to reverse engineer the software under attack. Reverse engineering or black-box testing may reveal bugs that an attacker can exploit to take control of a system considered secure. Automated testing is the most effective way to detect bugs during development, even before product release. In the following chapters, we explore and improve several automated techniques to find and mitigate several types of vulnerabilities at scale.

Chapter 3

Predictive Context-sensitive Fuzzing

3.1 Introduction

Fuzz testing (or *fuzzing*) techniques earned a prominent place in the software security research landscape over the last decade. Their efficacy in generating unexpected or invalid inputs that make a program crash helps developers catch bugs early, even before they turn into vulnerabilities [68]. As an example, their deployment at scale in the OSS-Fuzz [145] initiative has led so far to the discovery of over 30,000 bugs in the daily testing of hundreds of open-source projects.

The most popular and researched form of fuzzing is *Coverage-guided Fuzzing* (CGF), which uses code (or possibly other) coverage information extracted from the target execution to deem whether the current testing input led to *interesting* (e.g., previously unseen) portions of a program. The main intuition behind CGF research is that code coverage is strongly correlated with bug coverage [280] and no dynamic testing technique can detect a bug if execution does not reach the corresponding program point at least once.

Improving the effectiveness of the input generation process in CGF systems is a flourishing topic of research. For instance, various solutions focus on increasing the covered code [19, 303, 398], typically by guiding the fuzzer input mutations to meet complex control-flow conditions in the program. However, for software testing, coverage is only one part of the equation [173] and the ultimate metric for the effectiveness of fuzzing remains the ability to discover bugs.

Other efforts have focused on retaining for further mutations inputs that, while being equivalent to prior executions in terms of covered program points, exercise new valuable execution paths and internal states of the program [249]. To this aim, many state-of-the-art CGF systems track *edge coverage* information, so as to distinguish visits to the same basic block from different predecessor blocks [377].

Edge coverage and other *function-local* metrics track and summarize the effects of the entire execution on entities from individual functions. A limitation of this strategy is that they may lead a fuzzer to overlook relevant internal states of a program related to the different ways in which execution reaches an entity. In program analysis, this concept goes under the name of *context-sensitivity* and has seen many

applications (e.g., refining the precision of pointer analyses [387], developing compiler optimizations [159]).

ANGORA [68] has recently showcased the benefits of context-sensitivity for fuzzing by augmenting edge coverage with global *calling-context* information, i.e., the sequence of active function calls on the stack leading to each executing function [98]. Unlike context-insensitive fuzzing, such a *fully context-sensitive* approach can, in principle, differentiate the coverage of each testcase in a fine-grained manner and lead to the discovery of more bugs [68, 377].

As accurate call stack tracking and context encoding would be costly and degrade the fuzzer's throughput, ANGORA [68] and other fuzzers [131, 132] embody a *best-effort* strategy for full context-sensitivity: they model the calling context as a hash of the call stack and compute context-sensitive coverage identifiers by combining the hash for the current context with the function-local edge identifier upon entering a basic block.

This scheme is naturally prone to collisions, which are detrimental to fuzzing as they may lead to missing many relevant testcases [137]. Therefore, these fuzzers have to resort to larger coverage maps that can severely harm performance. More importantly, as we study in the chapter, fully context-sensitive approaches are prone to state explosion and tend to overly discriminate similar testcases, polluting the fuzzer's internal queue with a large number of redundant testcases that further reduce fuzzing efficiency [377].

In this chapter, we show that the current "*all-or-nothing*" approach to context-sensitive fuzzing is unnecessarily inefficient and a much more effective approach is possible. Our proposal is based on three important insights. First, we show that we can do away with run-time call stack tracking altogether by relying on a form of target code specialization, i.e., *function cloning*, which, for a given calling context, can create a clone of each callee and redirect the caller invocation to it. With this strategy in place, existing function-local coverage tracking techniques can naturally disambiguate calling contexts with no changes. As a result, edges from cloned functions can benefit from the collision-free tracking of modern fuzzers as their presence implicitly carries context-sensitivity information.

Second, we show that, while fully context-sensitive approaches are in general problematic due to an inherent state explosion problem, *selective* approaches can be a much better alternative. In particular, we show that, with techniques able to restrict cloning to program portions that are more likely to benefit from a contextual refinement of their edge profiles, we can bound our cloning efforts to trade a modest increase in program size with efficient context-sensitivity provided only for the callees that "*matter*". We term our approach *predictive context-sensitive fuzzing*.

Third, we show that the data flow of a program can be a strong predictor of those program portions. In particular, we analyze the flow of objects through function arguments at call sites and pick those call targets that see a highly diverse incoming data flow compared to other invocations of the function in the rest of the program. The intuition is that such differences reflect relevant variations in program behavior that we want to be able to capture by means of context-sensitive coverage tracking. Moreover, we show that such prediction strategy can be realized by analyzing only local data flows (i.e., at caller-callee pairs) rather than global ones (i.e., using full calling contexts). This results in a practical and scalable predictive context-sensitive

fuzzing solution.

On the popular FuzzBench suite [258], the best configuration of our approach can reveal more unique bugs than best-effort context-sensitivity (+19.05%) and an LTO-boosted collision-free edge coverage solution (+9.64%). The bugs we find across fuzzing sessions are different than with edge coverage alone by 18.04%. As we study in the chapter, this improvement mainly comes from our ability to explore more pervasively code regions already covered by context-insensitive solutions. Furthermore, unlike best-effort context-sensitive approaches that harm code coverage due to internal wastage, we can even slightly improve such metric while experiencing only a limited growth of retained testcases (+26% w.r.t. edge coverage) and a rather small impact on the execution throughput (−6.4%). Despite the subjects we studied are well-tested in prior efforts and daily in OSS-Fuzz, our tests revealed 8 enduring security issues in the latest code releases for 5 of them. Additionally, we observed and reported 10 bugs on 6 C++ case studies selected in order to put pressure on our predictive strategies. At the time of writing, 11 CVE identifiers have been issued in total.

Contributions. To summarize, this chapter proposes:

- A selective approach to context-sensitive fuzzing that augments only promising program portions with contextual information, using function cloning to enable a collision-free encoding with no run-time tracking;
- A data-flow analysis to predict program portions likely to benefit from such refinement when fuzzing, using a strong local signal given by call-argument value diversity among different callers for a given target function.
- An open-source implementation in LLVM that produces programs suitable for out-of-the-box fuzzing, available at https://github.com/pietroborrello/predictive-ctx-fuzzing.
- An evaluation of our techniques on top of AFL++ over the FuzzBench suite, where we consistently outrank state-of-the-art context sensitive and insensitive techniques without incurring internal wastage in the fuzzer, exposing several enduring security bugs. We pair it with 7 case-studies on other well-tested software, seeking new bugs.

3.2 Background

This section covers fundamental concepts of fuzzing and the pointer analysis primitives that back our predictive approach.

3.2.1 Coverage-guided Fuzzing

Fuzzing techniques have a prominent place in software security research due to their effectiveness in bug discovery [289]. In the most naive embodiment, a fuzzer is a system that attempts repeated executions of a target program over randomly generated testcases while monitoring it for crashes. Many techniques are nowadays available to optimize the testcase generation process, e.g., to discover more bugs within a given time budget [42] or to prioritize specific code regions for testing [43].

The amount of information that a modern fuzzer acquires during the executions of the program under test can vary, leading to a distinction between black-box [165,393], white-box [142,293], and grey-box [241,399] fuzzers. In particular, grey-box fuzzers use lightweight instrumentation to track coarse-grained state information such as the code coverage achieved by each testcase and are largely popular due to their effectiveness. As we anticipated in Section 3.1, tracking code coverage can also serve as a *feedback* for coverage-guided fuzzers, allowing them to distinguish the program behaviors distinctive of each testcase by profiling, e.g., the control-flow edges taken during the execution (edge coverage). Ultimately, this choice improves the ability of a fuzzer to find vulnerabilities [100].

Coverage-guided fuzzers instrument program code to update a *coverage map* (e.g., when the program takes a control-flow edge) that eventually serves as a profile of the testcase execution. Some also keep track of hit counts at coverage points. A relevant aspect of map updates involves *collisions*, which harm the effectiveness of fuzzing: a fuzzer may overlook program behaviors (and in turn bug discovery opportunities) if the encoding scheme for map updates treats two distinct coverage facts as if they were the same.

For instance, the popular AFL fuzzer [399] tracks edge coverage by combining, upon entering a basic block, the index of the current block with the one of its predecessors as $curr \oplus (prev >> 1)$. Despite a limited run-time overhead, this hashing scheme incurs frequent collisions [137]. Fuzzers such as AFL++ and LibFuzzer mitigate this problem by inserting dummy basic blocks to disambiguate *critical edges* [240] in the control-flow graph. Thanks to this transformation, they can track the original edges by using only the (unique) identifier of the currently executing basic block in the modified program, therefore achieving collision-free edge coverage.

3.2.2 Pointer Analysis

Pointer analysis is a static program analysis that identifies the possible targets of a pointer expression [164] by building the *points-to set* of abstract objects that each expression may reference. An *abstract object* corresponds to an allocation site and concisely represents all the concrete object instances that the program may create there. Points-to sets are always sound (i.e., they never miss feasible objects), while their accuracy (i.e., presence of objects that the program will never dereference) depends on the used pointer analysis.

The first element of differentiation for pointer analyses is the treatment of dependencies. Every time a location v results into a plausible target for a pointer p, an *inclusion-based* analysis [10] adds v to the points-to set for p and re-processes any constraints involving p at other program points, while an *unification-based* analysis [338] merges the points-to set for p with the one currently holding v. Inclusion-based analyses are more precise but face a worst-case complexity that is nearly cubic, whereas union-based analyses can typically run in linear time [337].

Further relevant properties are *flow-sensitivity*, i.e., when an analysis can compute a refined solution for a pointer expression at each program point, and *context-sensitivity*, i.e., when it uses contextual information to further differentiate allocation sites or the location of pointer expressions. Intuitively, an analysis featuring either property is more accurate but also more expensive to run. Pointer analyses are

used in several security scenarios (e.g., [67, 167, 286]), also thanks to recent technical advances and state-of-the-art implementations [220, 344] available for mainstream compilers.

3.3 Motivation and Open Problems

We use the code in Listing 3.1 to showcase how contextual information can help a fuzzer explore program locations or internal states that may trigger a bug only when the incriminated code is reached along certain execution paths.

Function `set_packet_data` contains a heap overflow bug at line 11. To trigger it, the program state needs to satisfy two conditions: (i) the allocation size of `packet` must be less than the size `len` passed as argument and (ii) the `level` field of `packet` must be 0 to skip the check on the size at line 8 and the subsequent error handling block. This can happen when the execution reaches `set_packet_data` from the `encapsulate` function. Condition (i) can be met thanks to an integer overflow at line 15, while encapsulating 256 packets would make the `level` field overflow and satisfy condition (ii).

Assuming that function `create` executes once and before `encapsulate`, when execution reaches the buggy line through the call from `encapsulate`, condition (ii) is met but a CGF system based on edge coverage cannot see the testcase as interesting (i.e., bringing new coverage) because the involved edge has been visited before[1]. Hence, it will not retain it for further mutations that may meet also condition (i).

ANGORA [68] extends edge coverage to distinguish executions of the same branch by different *calling contexts* (Section 3.1). To this end, it tracks the calling context dynamically as the hash of the current call stack, computed by XOR-ing at each call and return instruction the current hash value with the unique numeric identifier of the involved function. Then, it combines this hash with AFL's edge hash identifiers, obtaining a *best-effort* feedback where each map entry should ideally capture a distinct context-sensitive edge instance.

Challenges We studied the internal fuzzer wastage that comes with *best-effort* context-sensitivity approaches by analyzing popular fuzzing subjects with different configurations of the AFL++ fuzzer. We consider two standard configurations: 1) a context-insensitive AFL-style setup (EDGES) with a coverage map of 2^{16} entries indexed by edge hashes; and 2) the default AFL++ configuration (LTO) that uses collision-free edge coverage (state-of-the-art in the CGF practice, Section 3.2.1) with unique edge identifiers assigned during link-time optimization.

For fully context-sensitive fuzzing (CONTEXT), we reproduce the working of ANGORA by combining AFL's edge encoding with the XOR-based call-stack hash described above. We study two configurations, with coverage maps of 2^{16} and 2^{20} entries, respectively. As a reference, we also study our approach (PREDICTIVE) in the variant based on an inclusion-based pointer analysis (the variant using an union-based analysis—see Section 3.6—performed analogously here).

[1] Refining edge coverage with AFL-style 1-byte hit counts would not help here. This strategy effectively differentiates internal states only for small counts [68]: here, after 256 executions of the branch at line 8, the count overflows and the fuzzer loses sensitivity.

```
struct packet {
    u16 size;
    u8 level;
    u8 data[];
} __attribute__((packed));

void set_packet_data(struct packet* packet, u8* buf, size_t len) {
  if (packet->level && packet->size != len + sizeof(struct packet)) {
    error("Invalid encapsulation");
  }
  memcpy(packet->data, buf, len);
}

struct packet *encapsulate(struct packet* inner) {
  u16 size = inner->size + sizeof(struct packet);

  struct packet *outer = malloc(size);
  outer->size = size;
  outer->level = inner->level +1;
  set_packet_data(outer, (u8*)inner, inner->size);

  return outer;
}

struct packet *create(u8 *buf, size_t len) {
  if (len > 0x1000) return NULL;

  size_t size = len + sizeof(struct packet);
  struct packet *packet = malloc(size);
  packet->level = 0;
  packet->size = (u16)size;
  set_packet_data(packet, buf, len);

  return packet;
}
```

Listing 3.1. Motivating example for context-sensitive fuzzing.

Figure 3.1 plots statistics collected from a 24-h fuzzing on a subject, `libxml2`, particularly representative of the issues behind current approaches. We study the size of the queue, the throughput (completed executions), the number of distinct map entries covered by the testcases, and, where applicable, how many per-entry unique collisions we identified. A collision at a map entry implies that the fuzzer erroneously treated (at least) two distinct (context-sensitive) edge instances as if they were the same. The data highlight two internal wastage problems for current context-sensitive fuzzers, which we refer to as *coverage map explosion* and *queue explosion*.

To understand the first problem, we took a closer look at ANGORA [68]. As acknowledged by the authors, their encoding method for context-sensitive edge instances is prone to hash collisions (we identified them on 50.7% of the map entries for CONTEXT 2^{16}). Collisions are undesirable, since they lead to loss of context sensitivity[2] and ultimately increase the likelihood of discarding useful testcases [137]. Therefore, ANGORA uses a larger map with 2^{20} entries. While this choice can effectively mitigate collisions (1.2% for CONTEXT 2^{20}), it can hamper the throughput of the fuzzer because of higher map access latency (since the map would no longer fit common L2 cache sizes) and slower processing at the end of each execution. On standard hardware, we observed slowdowns of one order of magnitude. To partially mitigate this coverage map explosion problem, we collected our data on a high-end Intel Xeon Platinum 8160 with a 1-MB L2 cache. Even on such a high-end configuration, the number of completed executions dropped from ~45 millions to ~7 millions. Such low throughput ultimately resulted in much poorer edge coverage than any other configuration.

The second problem (queue explosion) is well-understood in literature [377]: many retained testcases with exceedingly high similarity. For the CONTEXT 2^{16} configuration, the queue size grows significantly (from 9,911 to 33,675 retained testcases), but the edge coverage achieved over time is appreciably lower than EDGES (where 9.8% of map entries see collisions) and much lower than the one obtainable with a collision-free LTO solution. The problem is less noticeable in the CONTEXT 2^{20} configuration (although the queue size still doubles to 21,157), but only because the much lower throughput (and edge coverage) masks the queue explosion problem.

Summarizing, our analysis shows that prior context-sensitive fuzzing strategies struggle to achieve good precision without introducing wastage due to explosion issues: allow more collisions and lose context sensitivity (at the cost of discarding important testcases), or reduce collisions and overly discriminate contexts (at the cost of retaining many redundant testcases and trashing throughput). The key reason this is essentially an impossible needle to thread is that prior strategies are entirely *blind* to which of the many distinct contexts (e.g., up to 16M from the `main` of `libxml2` and, in general, exponentially large w.r.t. the number of functions [387]) are important to capture in order to retain *interesting* testcases.

[2]And even worse weaker path sensitivity than a context-insensitive baseline, since a single hash is used for calling contexts and edges. Therefore, one may suggest combining a collision-free edge ID with a hash of the context. Unfortunately, this method would be much poorer than the one of ANGORA due to the limited entropy of edge identifiers, which would be completely marginal compared to the one of contexts.

Fuzzer configuration	Queue size	Executions / sec (large L2)	Map entries Used / Total	Map entries Colliding
EDGES (2^{16} map)	9,911	609.04	19.86% of 64 KB	9.8%
LTO (collision-free)	11,093	572.02	15.59% of 50 KB	-
CONTEXT (2^{16} map)	33,675	530.10	79.54% of 64 KB	50.7%
CONTEXT (2^{20} map)	21,157	84.38	7.21% of 1 MB	1.2%
PREDICTIVE	15,455	490.62	9.28% of 256 KB	-

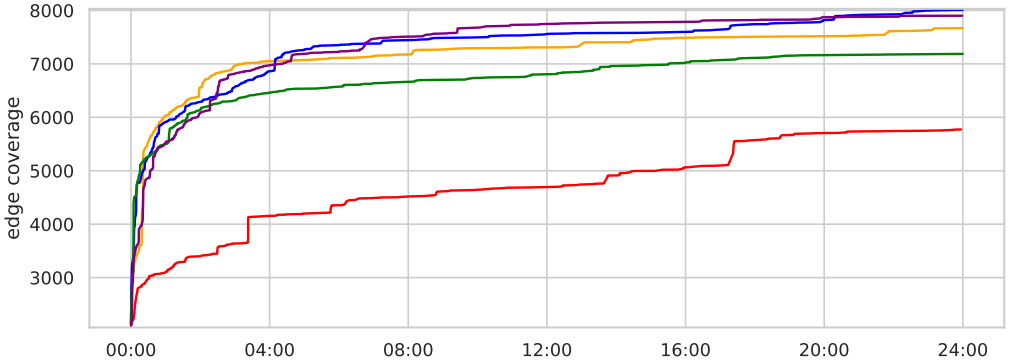


Figure 3.1. Fuzzer's internal wastage vs. edge coverage over 24 hours with best-effort context-sensitivity.

Our Approach We explore a *selective* angle to deploy context-sensitive fuzzing in a more effective way: we augment only certain program regions with contextual information, devising then a novel *predictive* solution to statically identify regions that are likely to benefit from context-sensitive edge profiles. As a concrete instance of this strategy, we favor call sites that see a higher diversity for the incoming data-flow at call arguments.

For our motivating example, such predictor would recognize that the `packet` object flowing into the buggy function comes from different allocation sites depending on the caller. Furthermore, as we study only local data-flows, instead of the full calling context we can rely on a much lighter context abstraction that discriminates only the identity of the caller function.

Ultimately, all these choices allow us to hit the "sweet spot" between insufficient and excessive context sensitivity, uncovering more bugs in well-known benchmarks with only a moderate impact on the fuzzer's internal wastage.

3.4 Predictive Context Sensitivity

This section presents the three main pillars of our approach: 1) a collision-free method to encode context-sensitivity, 2) a selective approach to restrict such sensitivity to program regions of interest for the sake of scalability, and 3) a data-flow analysis to predict regions likely to benefit from having been selected when a coverage-guided exploration reaches them. We produce a transformed program with context-sensitive

instances of control-flow edges, added according to a user-specified budget and in a cost-effective manner. Existing CGF systems can thus test such a program out of the box.

3.4.1 Function Cloning

A way to turn a context-insensitive program analysis into a context-sensitive one is to expose to the analysis a separate instance (*clone*) of the code unit of interest at each different encountered context. For instance, if contextual information is represented only by the caller of a function, the analysis may produce separate results for the unique clones of the callee devised for each possible caller.

Such an approach can accommodate different context-sensitivity definitions. Let us consider calling-context information, initially on recursion-free programs for simplicity. One may disambiguate the calling context for a specific function by taking the call graph of the program and, for each maximal acyclic path that reaches the function, introducing a clone at every caller-callee pair on the backward walk to its root node. In this way, whenever the analysis reaches a clone of the original function, the path from the root function to it is unique. Therefore, the identity of the clone determines the invocation context.

To handle recursion, we look for functions involved in direct and indirect recursion by analyzing the strongly connected components (SCCs) of the call graph [397]. During path analysis and cloning, we treat each SCC as a single node without a self-edge. This allows us to retain precise contextual information before and after entering recursive sequences (which in general may be unbounded in depth), treating only the recursive parts in a context-insensitive manner.

For a coverage-guided fuzzer, we need a way to discriminate different clones of the functions of interest that is both cheap to maintain or retrieve at run-time and composable with other encoding techniques in a space-efficient and collision-free way.

An elegant and effective way to maintain context-sensitivity for program points is to manipulate the code of the program and add concrete copies of the involved functions. This choice brings several advantages. By exposing contextual information through new code locations, we offload the collision problem to the feedback mechanism already in use by the coverage-guided fuzzer. In the case of edge coverage, collision-free encodings (Section 3.2.1) will just treat these new *context-sensitive* edges with unique identifiers. Furthermore, when deploying context-sensitivity in the *selective* flavor that we present in the next section, our scheme brings virtually no run-time overhead for tracking and retrieving the context, as we trade this efficiency for a modest increase in program size.

Let us use as running example our program from Listing 3.1. Its caller-callee pairs are (`create`, `set_packet_data`) and (`encap- sulate`, `set_packet_data`). For simplicity, we pick the second for specialization as we know that such path can expose the bug at line 11. Our cloning primitive adds to the program a duplicate of `set_packet_data`, which we call `__clone_spd`, and patches the call at line 20 to invoke it in lieu of the original function. When a coverage-guided fuzzer executes the augmented program, the branch originally at line 8 will benefit from separate coverage information when reaching `__clone_spd`, allowing the fuzzer to treat is

Table 3.1. Code features of FuzzBench subjects.

Benchmark	Type	Edges	Functions	Call sites	Calling contexts
`ffmpeg`	C, some C++	716 046	5 314	44 500	8 014 021
`file`	C, some C++	15 986	250	985	19 217
`grok`	C++	94 092	535	2 234	11 025
`libarchive`	C	67 096	866	4 377	27 984 301
`libgit2`	C	107 785	1 718	5 467	3 024 953
`libhevc`	C	119 646	197	853	125 907
`libhtp`	C++	11 203	181	706	6 718
`libxml2`	C	104 351	1 147	6 708	44 652 617 060
`matio`	C	24 112	300	1 795	2 793 663
`muparser`	C++	14 007	103	483	6 120
`ndpi`	C	49 216	355	1 991	10 507
`njs`	C	57 402	588	3 818	12 671 908
`openh264`	C++	78 819	384	1638	28 441
`stb`	C/C++	11 861	144	881	11 501
`usrsctp`	C	96 225	405	4 303	3 294 931 527
`zstd`	C/C++	38 863	848	5 027	140 141

as an interesting testcase (as execution hits a "new" branch) and to retain it for further mutations that eventually unveil the bug.

By choosing to work on call sites, we can virtually model any notion of context-sensitivity based on tracking portions of the call stack: a global policy will ensure that each cloning action draws out a piece of the desired portion. The call sites present within an added clone may be in turn disambiguated for context-sensitivity by applying cloning recursively.

3.4.2 The Need for Selective Sensitivity

While cloning can expose context-sensitivity information for program points in a "fuzzer-friendly" manner, it does not help us get around the path explosion problem that comes with calling contexts (Section 3.3). As evidence of this issue, Table 3.1 reports statistics collected for programs from the FuzzBench test suite that we later use for evaluation purposes (Section 3.6).

As a fuzzing harness often tests only a relevant subset of a code base, we collect the figures after removing all the functions unreachable according to LLVM's static analyses. In the *edges* column, we report the number of basic blocks that a collision-free edge coverage scheme instruments after breaking all the critical edges in program functions [241]. The last three columns represent, respectively, the number of nodes, edges, and acyclic paths in the call graph.

For many subjects, the number of contexts appears intractable for any practical collision-free attempt (we will return to this in Section 3.7), including cloning. Even when the contexts are not millions or more, the number of "context-sensitive" edges to disambiguate may still increase dramatically when the call sites are many, requiring in turn large coverage maps for their (collision-free) tracking.

However, we argue that a much more effective approach is possible: adding context-sensitivity only to *selected* program portions. Algorithm 1 presents the high-level workflow: we process the call graph at call-site granularity and follow a

prioritization policy to pick individual call sites for cloning.

We surveyed static analysis literature for contextual information representation in the PL community (e.g., [21,277,328]) and derived three policies that approximate their core ideas by performing a visit of the call graph and assigning priorities (captured by visit order) according to topological properties:

- *top*: assigns higher priority to call sites from nodes closer to the root(s) of the call graph, progressively exposing the context in a top-down fashion as in [277].
- *bottom*: assigns higher priority to call sites closer to leaves. This policy progressively exposes the last entries on the call stack as in *call strings* [328], which in some domains can effectively replace the full calling context.
- *uniform*: treats every call site with the same priority. It resembles [21] and mixes the effects of the other policies, exposing the top or bottom call-stack entries leading to a node depending on its proximity to a root node or a leaf.

During preliminary tests[3], these policies exposed a few additional bugs compared to standard edge coverage and best-effort context-sensitive solutions and did not experience any evident internal wastage. However, their apparent benefits were modest and also difficult to understand when compared to a *randomic* policy (which prioritizes call sites for selection uniformly at random) that we used as a baseline for selectivity and often had similar performance. The results for randomic, top (as 'bfs'), and uniform can be found at https://www.fuzzbench.com/reports/experimental/2021-05-25-cloning/index.html and for bottom at https://www.fuzzbench.com/reports/experimental/2021-07-09-cloning/index.html.

Eventually, we looked at these results retrospectively. Policies of this kind are well suited for static program analysis scenarios, where partial contextual information may still expose to an analysis sufficient information to reason on all the possible *refined* program states and, in turn, the user can measure the improvement (if any) in the precision of the returned answers. Instead, coverage-guided fuzzing is a dynamic analysis technique based on a lightweight abstraction of program state: no direct static measurement of the benefits of context-sensitivity seems possible. To effectively take advantage of any added context sensitivity, which can be available only in a limited quantity, we need a predictor for program portions that may practically benefit from it in a fuzzing sense.

3.4.3 Data Flow-based Prediction

A pivotal element of our proposal is a *prediction*-based policy that prioritizes for cloning those call sites exhibiting higher diversity in the incoming data-flow for the callee compared to other uses of such function in the rest of the program.

Our hypothesis is that data-flow diversity can be a strong indicator to identify execution contexts that are related to peculiar internal states of the program and are thus worth a pervasive exploration by the fuzzer. As we show in Section 3.6, the analysis that we describe next can be a good predictor for eliciting such states and uncovering new bugs.

We observe that function arguments are a natural way for programs to orchestrate data-flows through their code units. Therefore, we study the invocation of every

Algorithm 1: Priority-based Cloning for Partial Context-Sensitivity

```
function CloneByPriority(program, budget)
    callsites ← ⋃_{f∈program} GetAllCallsites(f)
    priorities ← GetPriorities(callsites)
    pqueue ← CreatePriorityQueue(callsites, priorities)
    while program.size < budget && callsite ← pqueue.pop() do
        target ← GetCallTarget(callsite)
        new_target ← CloneFunction(target)
        SetCallTarget(callsite, new_target)
        new_callsites ← GetAllCallsites(new_target)
        new_priorities ← GetPriorities(new_callsites)
        pqueue.push_all(new_callsites, new_priorities)
```

program function at the different call sites in the call graph and analyze what values are possible for each of its argument. In particular, we prioritize for cloning the call sites that pass, via arguments, objects that never or rarely appear at other call sites.

Put in other words, we find it reasonable to differentiate for the fuzzer those call sites that see peculiar objects, while we predict a lower benefit from doing so at call sites that see objects that recur elsewhere too. We remark that this choice does not relate to the complexity of the data-flow, but only to its diversity across distinct callers.

We use an off-the-shelf pointer analysis to build points-to information (Section 3.2.2) for pointer arguments, obtaining the possible abstract objects that an argument may reference when passed at a call site. We compute the prediction to use as priority value in Algorithm 1 as follows. Let the target function be in use at n call sites in the call graph and O be the set of all abstract objects that may be passed via its arguments at the current call site. The priority p of the call site is:

$$p = \frac{1}{n} \times \sum_{o \in O} (n - n_o)$$

where n_o is the number of call sites for f where object o may appear in any of its arguments. As we said ealier, we seek to favor the diversity of the incoming data-flow: an object o that does not appear at other call sites for the target will contribute with a $n - 1$ addend, whereas an object that may appear at all call sites will give a zero addend. Eventually, the edge coverage collected for the clones will expose the incoming data-flow diversity to the fuzzer, favoring a more pervasive exploration of the underlying program states.

3.4.4 Discussion

With predictive context-sensitive fuzzing, we propose to overcome the scalability and efficiency limitations of current context-sensitive fuzzing flavors by augmenting only selected program points with context information.

Our data flow-driven prioritization policy turns out to be effective in practice in discriminating program states that eventually lead to discovering more bugs through further input mutations. We focus on memory objects as fuzzers are notoriously effective in exposing memory handling errors [104], especially in combination with sanitizers [196]. We believe that other data flow-driven policies can help discover

further, different bugs. Furthermore, our approach may be adapted to work with other argument types, for instance by carrying value range analysis [154] on integer arguments.

Compiler-based instrumentation is a natural way to deploy our approach. For fuzzing programs available only as binaries, binary rewriting techniques or a modified JIT can intercept and divert call sites. However, analyzing pointer arguments may be challenging as, among others, it would need recovering object locations. We leave this investigation to future work.

3.5 Implementation

We implement our techniques as a set of analysis and transformation passes (2200 C++ LOC) for the intermediate representation (IR) of the LLVM compiler, which is a popular choice among CGF systems that instrument programs during compilation. We operate on a link time-ready whole-program IR file [254] for the uninstrumented program and produce a transformed IR file that we can feed to an off-the-shelf fuzzer.

As for evaluation purposes we work with AFL++ [132], which is currently one of the most performant CGF systems, we devise a small Python helper that automates the compilation process and also the insertion of sanitization machinery. Our cloning pass has provisions to correctly handle the instrumentation introduced by sanitizers such as ASan and UBSan, which insert tripwires that help fuzzers expose silent bugs [103].

For sizing purposes, we implement an analysis to estimate how many additional unique identifiers the collision-free edge coverage encoding of AFL++ (Section 3.2.1) would need after a cloning decision. This affects the number of elements that the coverage map of the fuzzer should account for and, therefore, its size. Good fuzzing practices [68] recommend map sizes no larger than (standard) L2 cache sizes, whereas overly large maps can be detrimental for performance even on favorable hardware as we saw in Section 3.3.

This means that, once we set a maximum desirable map size, we can use as residual budget for cloning the "free" map entries after we accounted for the edges currently in the program and, potentially, add clones up to its exhaustion.

For our evaluation, we opted for a general budget of 256 KB, which can host up to 2^{18} map entries. This general-purpose tuning choice allowed our fuzzers to discriminate and pervasively explore new program states without incurring internal wastage. Appendix D.3 details several preliminary experiments that we made with larger budget choices.

To analyze pointer arguments at call sites, we used the inclusion-based `Flow Sensitive` analysis of SVF [344] and the default unification-based analysis of SeaDsa [220].

As an implementation refinement, we heuristically attempt to lower the priority of a recurrent class of uninteresting targets for cloning: error-handling functions that lead to program termination. Such functions often see a high number of callers and an inherently diverse incoming data-flow: in our tests, we observed that targeting the functions that are called by no less than 25% of all the program functions ruled

out a great deal of error-handling functions with no false positives.

Our prototype can also reason on paths involving indirect-call sites by promoting each indirect call into a conditional selection of direct calls to plausible targets [8, 24, 116]. However, this is disabled by default since reasoning on indirect calls is notoriously hard. With a static approach, the precision of the pointer analysis for building call-target sets is crucial [37]: in most of the cases we analyzed, the size of the resulting sets led to path explosion. Nonetheless, as we will see throughout Section 3.6, the effects of our techniques allowed us to expose bugs and report security vulnerabilities in heterogeneous programs written in C++, mixed C/C++, and object oriented-style C. As future work, we plan to explore the potential benefits of profile-guided indirect call promotion [24] for these subjects, for instance using testcases from a short fuzzing session.

3.6 Evaluation

We study the performance of predictive context-sensitive fuzzing using the FuzzBench testing infrastructure. Popular in academia and industry since its release in 2020 and targeting real-world programs, FuzzBench has become de-facto a standard fuzzer benchmarking platform [258]. We study different dimensions of our approach for the following research questions:

RQ1 What burden do we place on the compilation pipeline?

RQ2 Can we outperform the state of the art in terms of bugs found? And which prioritization policy performs best?

RQ3 To what extent do we induce internal wastage, if any?

RQ4 Can we also find new bugs in well-tested software?

On top of the popular AFL++ [132] fuzzer, we test two `dataflow` variants of the *predictive* context-sensitive approach (one per selected pointer analysis), the `randomic` prioritization policy for uninformed *selective* context-sensitivity, and the `context` *best-effort* context-sensitivity of ANGORA [68].

To fully cover the context-sensitive fuzzing spectrum, we also test `lto` collision-free edge coverage boosted with link-time optimizations, which not only is the most effective coverage policy in current CGF systems [132] but, as we discuss later, *incidentally* introduces some context-sensitivity.

For `context`, we use a coverage map of 2^{18} entries to fill the L2 cache (256 KB) typical of the FuzzBench cloud infrastructure on which we ran the majority of our tests. We do not evaluate larger sizes as we experienced significant internal wastage for the reasons discussed in Section 3.3. For `lto`, the number of instrumented edges in each program (Table 3.1) determines the map size. For our `dataflow` and `randomic` fuzzers, we use the largest cloning budget value such that the resulting map still fits the L2 cache[4].

We could obtain a compilable whole-program IR file (Section 3.5) for 16 of the 22 benchmarks from FuzzBench. Bugs and missing features in the GLLVM [254] helper and other compilation errors unrelated to our techniques prevented us from testing

[4]For `ffmpeg` the number of unique edges already requires more than 2^{18} entries: we set the budget for it to the nearest feasible multiple of two (768 KB).

Table 3.2. Statistics from the pointer analyses (run-time costs and points-to set sizes) applied to the FuzzBench subjects.

	Time (sec)		Memory (MB)		|Set| (avg.)		|Set| (std.)	
Benchmark	**seadsa**	**svf**	**seadsa**	**svf**	**seadsa**	**svf**	**seadsa**	**svf**
`ffmpeg`	16.54	2193.75	3202	22553	215.53	4.38	694.64	68.31
`file`	0.12	25.07	94	522	4.86	2.79	9.36	6.26
`grok`	-	181.55	-	2797	-	1.74	-	2.92
`libarchive`	0.65	103.5	242	2073	31.98	2.1	50.99	15.21
`libgit2`	1.55	446.52	457	4711	92.15	2.41	99.49	9.91
`libhevc`	0.58	93.73	327	3143	173.45	1.15	71.12	0.52
`libhtp`	0.09	161.98	89	647	1.77	56.1	1.87	43.79
`libxml2`	6.4	648.34	870	4289	585.36	9.83	272.97	12.26
`matio`	0.32	111.34	138	1181	79.02	32.47	76.72	30.76
`muparser`	0.1	12.53	97	487	5.68	2.9	9.95	5.83
`ndpi`	0.62	265.14	309	3250	58.77	115.5	85.7	87.18
`njs`	2.21	185.14	320	2012	157.78	16.22	190.22	11.74
`openh264`	0.51	108.55	232	2269	46.32	2.11	61.09	13.39
`stb`	0.13	12.86	87	413	36.5	2.08	28.7	2.28
`usrsctp`	1.34	1095.52	330	5851	161.39	55.85	81.29	33.07
`zstd`	0.32	45.47	175	1206	7.72	13.85	8.67	5.25
Geo-mean	0.62	139.94	255.63	2007.8	44.24	7.09	48.344	11.074

the other programs. The link-time primitives that became available with LLVM 12 may help for them for future works.

For all the fuzzer configurations that we study, we instrument each whole-program IR file with the ASan and UBSan sanitizers [336] to expose common classes of silent bugs.

3.6.1 RQ1: Analysis and Compilation Costs

Our approach incurs direct and indirect preparation costs for the transformed program that we eventually feed to an off-the-shelf CGF system (AFL++ in all our tests).

Direct costs include generating clones (typically a very fast operation) and running the analyses behind our predictive policy. Table 3.2 reports the CPU time and memory usage from running points-to analysis on each whole-program IR file.

The inclusion-based analysis of SVF is more costly, as expected (Section 3.2.2): it takes on average 139.94 seconds and 1.96 GB of memory to analyze a program, peaking at 2193.75 seconds and 22 GB on a larger subject like `ffmpeg`.

The union-based analysis of SeaDsa is cheaper, completing in most cases in less than one second and using 7.9x less memory than SVF. We do not report numbers for `grok` as SeaDsa mishandles the instrumentation of UBSan for it.

Table 3.2 reports also statistics on the average number of possible pointed objects per call-site argument. SVF produces smaller (therefore, more accurate)

sets than SeaDsa, with on average 7.09 abstract objects per pointer (~6x smaller than the average value measured for SeaDsa). However, context-sensitivity and other enhancements make SeaDsa produce smaller sets on three subjects (`libhtp`, `ndpi`, `zstd`).

Indirect costs for IR preparation include the impact of cloning on the binary compilation process orchestrated by the off-the-shelf fuzzer. Among our `dataflow` and `randomic` configurations, we observe an average increase in compilation time of 153 seconds (peaking at 443 on `ffmpeg`).

As for the resulting binary size increase, which includes the instrumentation added by the fuzzer for context-sensitive edge instances, we observe a geometric mean of 3.6x and a peak value of 10.1x (`stb` grows from 1.45 MB to 14.8 MB), while `libarchive` sees the largest produced binary with its 46 MB (initial size: 34.1 MB). In the context of fuzzing, though, such increases hardly affect performance. This applies to both persistent fuzzing scenarios (as with FuzzBench), where a binary is (re)loaded in memory only sporadically, and fork-based settings, which benefit from copy-on-write OS mechanisms. This observation will be experimentally confirmed in Section 3.6.3: in our tests, trading such additional space for supporting selective collision-free context-sensitivity did not harm the execution speed and effectiveness of our fuzzers.

3.6.2 RQ2: Effectiveness in Bug Finding

To evaluate the bug finding capabilities of our five fuzzer configurations (hereafter *fuzzers* for brevity), we rely on the infrastructure of FuzzBench to count unique bugs via automatic crash deduplication. As we run the fuzzers on its cloud service, each configuration-benchmark pair sees 20 trials of 23 hours each.

Following standard practices [208], we reason on the median values over all trials to mitigate the well-known effects of randomness in fuzzing. Figure 3.2 reports the boxplots for each benchmark showing the number of bugs found by each fuzzer. For each benchmark, the fuzzers appear in the ranking order given by their median number of bugs found across the trials (and by their maximum number when breaking ties).

Trends. To compare the effectiveness of each fuzzer, we first consider the *average score* metric from FuzzBench. For each benchmark, the score of a fuzzer is given by expressing the median number of bugs it finds as the percentage of the median number of bugs from the fuzzer that performed best on that benchmark. The final cross-benchmark average score for a fuzzer, shown in Table 3.3, is the average of individual benchmark scores and mitigates distortion effects due to benchmarks having a different number of total bugs [258].

The best-performing fuzzers are those using predictive policies. In particular, the `svf` fuzzer obtains the highest score with an 11.84 net difference with `lto`, which in turn largely outperforms `context`; `svf` will similarly stand out also in the analysis of individual bugs provided next. The score of the `seadsa` fuzzer is harmed by inability to test `grok` (Section 3.6.1), yet its score is very close to the one of `svf`. The reason is that, as it will become apparent in the boxplots of Figure 3.2, `seadsa` consistently

Figure 3.2. Boxplots with mean value (△) and raw data points (·) for bugs uncovered in the FuzzBench programs across 20 trials. Fuzzers are ordered by ⟨median, maximum⟩ number of bugs found. `usrsctp` is omitted as no fuzzer found bugs for it.

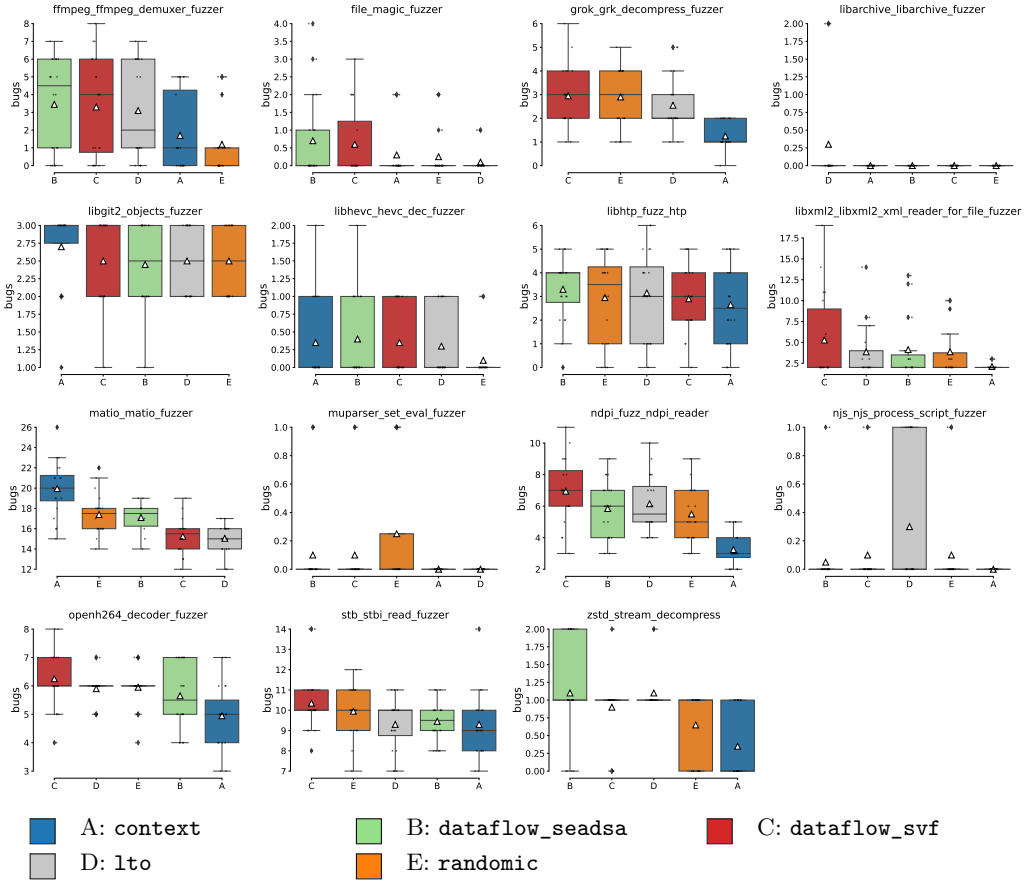


Table 3.3. Cross-benchmark average score from FuzzBench.

Fuzzer configuration	FuzzBench score
`dataflow_svf`	94.14
`dataflow_seadsa`	93.69
`randomic`	82.98
`lto`	82.30
`context`	63.42

finds comparable amounts of bugs among runs, resulting into competitive median values.

As we move to the other fuzzers, we remark how the `lto` state-of-the-art configuration is a *strong baseline*. In addition to collision-free encoding of edges, which outperforms classic (collision-prone) edge tracking and refinements [137], it benefits from link-time optimizations such as additional inlining. For instance, LLVM may inline a short-sized callee at a call site for performance, incidentally providing *some* context-sensitivity [380] as the inlined edge instances get new identifiers. However, an optimizing compiler follows performance-based (rather than context sensitivity-

based) inlining policies. When our data flow-based prediction mechanism drives the cloning decisions, we can observe a significantly larger number of bugs found for the subjects considered in this evaluation.

On the contrary, the best-effort context-sensitivity of `context` clearly suffers from a combination of the factors analyzed in Section 3.3. While we defer a detailed discussion of internal wastage effects to the next section, collisions hamper its ability to distinguish, and thus explore, useful program states that not only the `dataflow` fuzzers, but even `lto` can often retain in its queue. Combined with the time spent analyzing likely uninteresting testcases that pollute its queue and the lower end-to-end throughput (as discussed next), `context` ranks on average as the least effective fuzzer configuration in our tests.

Interestingly, though, `context` is the best performer on `matio`, which features a high number of potential calling contexts (Table 3.1). From analyzing its C source code, we noticed that `matio` follows an object-oriented paradigm that heavily relies on evolving the state of a single object. Therefore, we expect a limited diversity in the data-flow for the definition that we used for our prediction, which sees objects as a whole. Our approach, apparently, may lose efficacy whenever state differences involve portions of a single object: even the field-sensitive pointer analysis behind `svf` could not draw out complex variations of the data. On the contrary, `svf` is the most effective fuzzer on `libxml2`, another target written in C with an object-oriented paradigm and with a huge amount of potential calling contexts. Complex state variations as in programs like `matio` deserve further investigation, for example combining our approach with the data-oriented feedback of [130] from likely invariants for program variables.

In-depth Analysis. We can now qualitatively analyze the unique bugs identified by the fuzzers `svf` (125), `sea` (110 but one subject short), `lto` (112), and `context` (102). We omit `randomic` (110) in the following for brevity.

The left part of Figure 3.3 compares the unique bugs found by `svf` (our best performer) against the `lto` and `context` fuzzers that embody the state of the art. Due to internal wastage, `context` missed 27 of the unique bugs that both `svf` and `lto` could find. Of the 102 unique bugs `context` found, 74 were found by both the others, and 82 by `svf`. As for the 18 bugs that only `context` could find, 15 were from `matio` on which, as we discussed above, our predictive strategies appear less effective. On the other hand, `svf` revealed twice as many (43) unique bugs missed by `context`. As for our other predictive configuration, `seadsa` revealed 28 additional bugs, missed 23 bugs (2 from `grok` that was not tested), and shared 82 bugs with `context`.

The right part of Figure 3.3 compares the unique bugs found by `lto` against the two predictive fuzzers. Our fuzzers found several bugs that `lto` missed: 23 for `svf` (+20.2%) and 15 for `sea` (+13.1%). Of the 112 bugs found by `lto`, the bugs missed were 12 for `svf` (-10.7%) and 22 for `sea` (-19.6%). Here, `sea` is hampered by not being able to compile `grok`, losing any opportunity to find the 7 bugs from `svf` or the 6 from `lto` in it.

To study how refined contextual information is behind the bugs that only our fuzzers found, we analyze the characteristics of each crashing testcase. To this end,

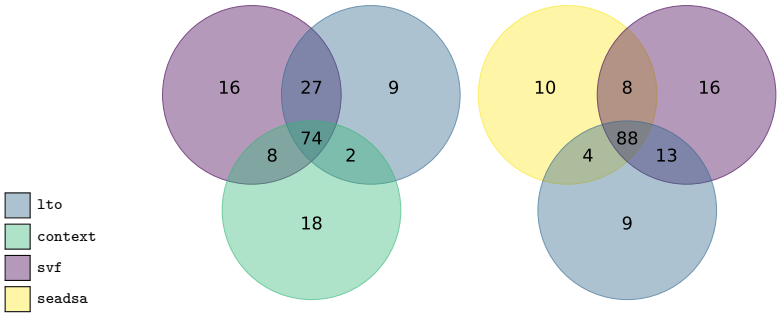


Figure 3.3. Venn diagrams for unique bugs found by the fuzzers.

Table 3.4. Impact of cloned functions on locally reproduced bugs.

	seadsa	svf
(Locally reproduced) bugs missed by `lto`	15	23
Code covered by `lto` w/o crashing	15	16
Execution of testcase invokes clones	14	21
Stack trace for crash contains clones	12	14

Table 3.5. Median queue size and executions/second ratio for each fuzzer across 20 trials of the FuzzBench programs.

	Queue size					Executions per second				
Benchmark	**context**	**seadsa**	**svf**	**lto**	**rand.**	**context**	**seadsa**	**svf**	**lto**	**rand.**
`ffmpeg`	11202	9536	9787	9713	8711	148	176	189	190	143
`file`	4046	2653	2681	1734	2645	425	438	463	501	425
`grok`	17651	-	4503	4093	4975	35	-	152	131	137
`libarchive`	8007	6608	6938	5526	5410	1659	1395	1421	1601	1500
`libgit2`	2443	1206	1190	1128	1271	899	896	826	868	931
`libhevc`	13229	9035	8727	7515	11161	166	138	153	122	172
`libhtp`	9243	13984	13878	6466	13990	2193	2964	2881	2274	2814
`libxml2`	41928	14106	15652	13977	15126	1352	1158	1090	1132	1249
`matio`	15040	10412	10374	9068	10935	298	389	415	492	334
`muparser`	1837	1456	1522	1184	1828	1215	2121	2183	3221	2108
`ndpi`	1623	1754	1673	1651	1783	38	42	39	42	42
`njs`	27660	5288	5083	4862	5236	498	620	570	650	623
`openh264`	6904	8625	8031	5239	7770	6	8	8	10	9
`stb`	3761	6292	6297	3228	6102	1414	1131	1384	1565	1330
`usrsctp`	2632	1635	1700	1631	1635	2351	2127	2164	2318	2136
`zstd`	26711	33782	25671	18464	28259	6516	4463	5149	6307	4841
Geo-mean	7784	5564	5416	4283	5528	431	535	506	541	502

we downloaded the queue from each trial from the FuzzBench infrastructure and ran each testcase on a locally compiled binary. Unfortunately, we could not reproduce some bugs counted by FuzzBench as the available enclosing zip files turned out corrupted.

To identify when, despite the context-sensitivity enrichments from LTO decisions, traditional edge coverage produced testcases that covered buggy code without inducing a crash, we check the bugs found only by our fuzzers against the cumulative coverage achieved by `lto` on each of its 20 runs.

Table 3.4 shows that all the code behind (locally reproducible) bugs additionally found by `seadsa` was always covered by `lto` without yielding a crash (15/15). For `svf`, 16 out of 23 additional bugs occurred in code that `lto` covered without yielding a crash. Interestingly, instead, nearly one third (7/23) of the bugs missed by `lto` come from new code coverage, which we believe the fuzzer could obtain by further mutating testcases that only context-sensitive edge counts made it retain.

More generally, for the majority of additional bugs from our fuzzers, we observe that one or more cloned functions are active on the call stack upon the crash, suggesting that cloning choices directly contributed to exposing the bug. In other cases, contextual information helped by retaining a testcase during previous executions (as it exercised edges from clones), allowing the fuzzer to further mutate the program state and data until exposing the bug. All these considerations back the intuition that data-flow diversity is a good predictor of regions that may remarkably benefit from context-sensitivity when fuzzing. We will resume this discussion in Section 3.6.4 by presenting a case study.

On a different note, by comparing the data points of Figure 3.2 with the code features of programs listed in Table 3.1 (we provide the reader with a simplified view in Table D.1 of Appendix D.1), the bug finding capabilities of our two fuzzers do not reflect a strong influence from the source language, even for C++ subjects. Section 3.6.4 will provide further thoughts on this point.

3.6.3 RQ3: Internal Wastage

As discussed in Section 3.3, internal wastage may hamper the effectiveness of a fuzzer by making it explore uninteresting program states and/or face higher latencies for completing the execution cycle of each testcase. We studied its impact by collecting statistics on the queue size and the execution throughput of each fuzzer at the end of the session in Table 3.5.

Our two predictive fuzzers yield a median queue size that, on average, is moderately larger than the value measured for `lto`: by 26.4% for `svf` and 29.9% for `sea`. As we discussed, some of the additionally retained testcases let them further explore states that led to additional bugs, with the data-flow being an effective predictor for eliciting such states. The queue median size growth for the all-or-nothing approach of `context` is on average 81.7% compared to `lto`, with peak values on `grok` (331.2%), `libxml2` (200%), and `njs` (468.9%).

To better put these numbers in perspective, we first study how many executions each fuzzer completed in a unit of time. Compared to the baseline `lto`, we observe for `context` a reduction of the fuzzing execution throughput by 20.3% on average, whereas our `svf` fuzzer (the best performer in RQ2) is slower than `lto` by only 6.5% on average. As anticipated in the discussion of RQ1, our compile-time cloning did not introduce any significant execution speed penalty (unlike, e.g., [224]).

Next, we also study how code coverage was affected. Context-sensitive approaches mean to favor more pervasive explorations of program states already reachable via function-local feedbacks (i.e., edge coverage), but internal wastage effects can hamper code coverage itself. Compared to `lto`, the best-effort context-sensitivity of `context` was detrimental for the code coverage obtained on several benchmarks (`libarchive`, `libhtp`, `libxml2`, `matio`, `njs`, `openh64`, `stb`). Both of our predictive fuzzers perform

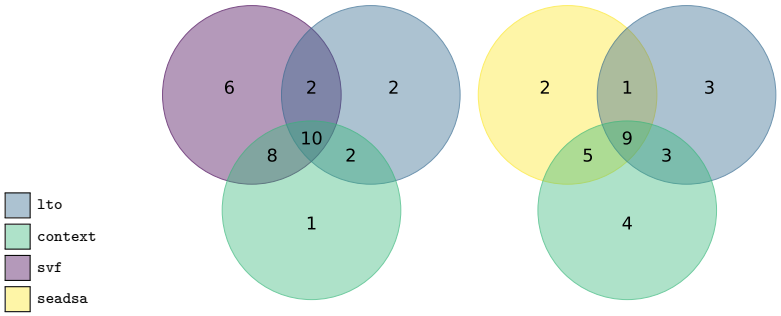


Figure 3.4. Breakdown of new bugs for FuzzBench programs.

well, obtaining coverage close to `lto` and higher in at least one variant (typically `svf`) on all subjects except `openh264` and `ztd`, showing no internal wastage. For this improvement, we note that the refined data-flow along cloned call sites may help the fuzzer retain and later mutate testcases that eventually lead to new code, as we observed in RQ2 for some of the additional bugs found by `svf`. Omitted here for brevity, Figure D.1 in Appendix D.1 provides complete charts for each fuzzer and program.

3.6.4 RQ4: New Bugs

As a last dimension to investigate, we conducted two sets of experiments on bugs that our approach can find in well-tested software, leaving out the less performant `randomic` for brevity.

FuzzBench. We first analyzed whether any bugs discussed in RQ2 would affect the latest program versions too (February 2022), which follow those used in FuzzBench by 9 months to over 4 years and are tested daily by the OSS-Fuzz initiative. As shown in Figure 3.4, 33 testcases (deduplicated by FuzzBench) could crash those versions too: in particular, `svf` (26 found out of 33) widely outperforms `context` (21), which in turn found only one bug that `svf` or even `lto` did not.

For the 33 testcases, we ruled out a few that matched issues in existing public bug reports and responsibly disclosed all the others to the respective developers. For bugs that hinted at ostensible security issues, we conducted further manual analysis to identify the logical root cause underlying each bug and cluster them accordingly (that is, we "conceptually" merged some). This analysis exposed 8 potential security issues in 5 programs: `ffmpeg` (1), `njs` (1), `stb` (4), `libhevc` (1), and `matio` (1). Six of them received a CVE ID (Appendix D.1), 1 was deemed a duplicate of one of our newly assigned CVE IDs (`stb`), and 1 was not considered a vulnerability by the vendor according to its criteria (we reported an undefined behavior from an invalid shift in `libhvec`). From commit dates, the issues were present in programs since at least 1.5-3 years.

In more detail, 5 issues derive from bugs found by our predictive fuzzers only: 3 for `stb` and 1 each for `ffmpeg` and `libhevc`; the `svf` variant exposed them all while `seadsa` missed 1. For these issues, `lto` typically covered the involved code without triggering a crash, with the exception of 1 issue (exposed by `svf`) as it involved new

Table 3.6. Bugs found in the additional studied C++ programs.

Benchmark	context	seadsa	svf	lto
exiv2	2	6	4	5
harfbuzz	1	1	1	1
httplib	1	1	1	1
lrzip	3	3	3	3
powerdns	0	0	0	0
protobuf	2	2	2	2
solidity	0	2	1	0

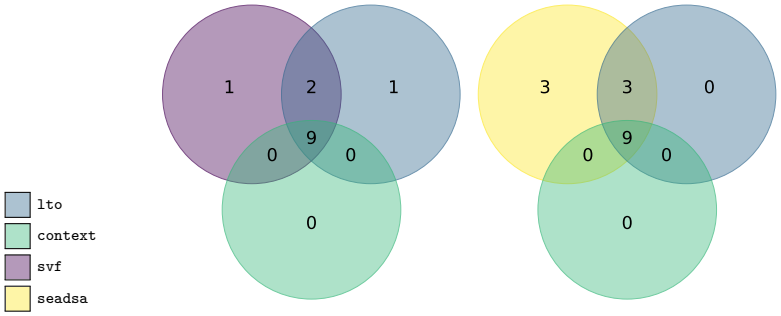


Figure 3.5. Breakdown of new bugs for programs of Table 3.6.

code coverage. The remaining 3 issues came from bugs spotted by both `svf` and `lto`[5].

CVE-2022-28048 As a case study, we discuss one of the CVEs assigned for `stb`, which is an image processing C library tested daily in OSS-Fuzz. The issue showcases how context-sensitivity is helpful to expose overlooked buggy code and how our predictive, data flow-based mechanism made effective cloning decisions for that end. It manifests as a heap use-after-free violation caused by an out-of-bound array write during JPEG decoding. The vulnerable function `stbi__process_marker` does JPEG segment processing and sees high call counts from two call sites: in the initial header parsing of `stbi__decode_jpeg_header` and in the subsequent image decoding of `stbi__decode_jpeg_image`.

For the first call site, the parser logic curtails the set of feasible program states by discarding early invalid header segments that would hit the bug, making it very hard for the fuzzer to expose it. This, however, does not occur at the other call site, which takes in non-header segments. Context-insensitive fuzzers and their enhancements like `lto` cover edges of the vulnerable function, but do not differentiate (and thus retain testcases for) the program internal states when invalid segments reach it from the second call site because they see no novel coverage. Our predictive approach introduces context-sensitive instances of these edges: the fuzzer sees them as distinct and will further mutate the associated testcases, eventually exposing the bug. Both `svf` (0.91) and `seadsa` (0.98) selected the call site for cloning with a high priority value p (Section 3.4.3).

More C++ Programs. As a second set of experiments, we studied 7 additional real-world subjects often used in prior works [130, 249] and/or tested daily in OSS-Fuzz. We specifically chose C++ programs to confirm that our approach can expose more and diverse bugs than `context` or an LTO-boosted edge coverage despite our prototype presently does not make predictions on the data-flow at indirect call sites. Table 3.6 shows the total unique bugs found in 5 fuzzing runs of 24h on the Intel Xeon Platinum 8160 machine used in Section 3.3.

In this experiment, the four fuzzers revealed 15 bugs in total: as shown in Figure 3.5, `seadsa` is the best performer and exposed them all. Both of our predictive fuzzers found all the bugs exposed by `context` (9) and additional ones in `exiv2` and `solidity`, which `context` likely missed due to internal wastage factors. `svf` yielded the same bug count (12) as `lto`: at a closer look, `svf` found 1 more bug in `solidity` but missed 1 of the 5 bugs that `lto` found in `exiv2`.

The increment and diversity of unique bugs found by our approach on these subjects appear consistent with the RQ2 experiments. Besides fuzzing randomness factors, a reason behind the excellent performance of `seadsa` here may be the recent refinements to the pointer analysis of SeaDsa to remove oversharing effects [220], which in the experiments of the authors led to higher precision than SVF on different C++ subjects.

For the 15 bugs, we conducted a manual analysis alike to the one described for the FuzzBench new bugs and responsibly disclosed all the issues to the involved parties. As of now, `lrzip`, `harfbuzz`, `solidity` and `protobuf-c` saw 5 CVE IDs assigned while we are awaiting bug evaluation for the others.

3.6.5 Discussion

In our tests, predictive context-sensitive fuzzing significantly outperforms the all-or-nothing, best-effort approach pioneered in Angora. The internal wastage factors induced by the latter make it fall behind even the `randomic` fuzzer configuration in several tested dimensions (e.g., 110 vs 102 unique bugs in RQ2). Our data-flow based predictive policy largely outperforms `randomic` as well as the three topological policies (top, bottom, uniform) that we evaluated in preliminary tests and discussed in Section 3.4.2. The results back our expectation that providing efficient and collision-free context-sensitivity only for the callees that matter—heuristically identified with a predictor based on diverse incoming data-flows at call sites—offers a practical, cost-effective, and scalable fuzzing solution.

Our techniques come with tenable compilation and analysis overheads, very limited run-time overhead, and do not cause queue explosion. The moderate number of additional testcases in the queue was instead instrumental for discovering more bugs in the tested benchmarks. Also, retaining a testcase characterized by a context-refined data flow occasionally helped a fuzzer reach new code through subsequent mutations.

The variant based on SVF generally resulted as the most effective configuration.

[5] Our readers may wonder why `lto` would still find bugs in well-tested software. While OSS-Fuzz conducts daily 5h tests on them, the collision-free configuration that we use is more performant than its settings thanks to LTO effects (including amounts of context-sensitivity from extensive inlining, Section 3.6.2).

A thorough study of the divergences in the cloning choices from different pointer analyses may be a promising direction for follow-up work. Appendix D.2 discusses a very fast intra-procedural analysis that we designed for minimal preparation costs in continuous integration pipelines.

Detailed analyses of the identified bugs revealed that not only our fuzzers found more and different bugs than `context` and `lto`, but also uncovered enduring bugs and security issues in subjects well-tested by the community. Even on programs that make a heavy use of functions pointers (i.e., the 7 case studies), our approach largely outperformed best-effort context-sensitivity [68] and could reveal more bugs than an LTO-boosted edge coverage; a profile-guided extension (Section 3.5) could be a worthy direction to improve our results.

3.7 Related Work

Local Feedbacks. A few function-local feedbacks have been proposed as a replacement or extension of code coverage. Padhye et al. [283] analyze alternatives such as the number of bits matched between operands of integer comparisons (for input-dependent conditions that are difficult to satisfy) or the size of allocation operations (for memory corruption-related bugs). Wang et al. [377] study, among others, extensions for the edge-coverage feedback currently in use by most CGF systems. For instance, the authors evaluate *n-gram* feedback to track bounded-length sequences of consecutively traversed edges as a better approximation of the program behaviors. Other efforts investigate auxiliary feedbacks involving data profiles [130, 162, 251]. As local feedbacks can naturally be augmented with our cloning-based context-sensitivity, future research may involve identifying profitable combinations.

Directed Fuzzing. While directed fuzzing is a long-studied subject [138, 141], its combination with grey-box fuzzing was only recently introduced with AFLGo [43]. This flavor of grey-box fuzzing can guide the exploration towards specific program points deemed interesting: for instance, a vulnerable code location. AFLGo builds a whole-program inter-procedural control flow graph (CFG) and assigns weights to basic blocks to define a distance function from the entry point to the target locations. The fuzzer uses this information to assign more energy to testcases that can potentially generate (directly or indirectly) a testcase triggering the target location. Hawkeye [67] improves the underlying CFG construction adding indirect calls targets, relying on the accuracy of the underlying pointer analysis used to resolve the indirect calls.

While directed fuzzing focuses on reaching predetermined program points based on user-specified criteria, our approach automatically selects interesting program points for context-sensitive coverage tracking. Nonetheless, our approach can potentially enhance directed fuzzing in two ways: (i) context-sensitivity may improve CFG construction (refining pointer analysis results for indirect calls) and (ii) given a stacktrace, we may clone only the specific context that leads to the target program state and assign ad-hoc weights to clones.

Software Hardening. A few hardening solutions resort to cloning techniques, often in combination with pointer analyses. Constantine [48] uses function cloning to improve the accuracy of pointer analysis by adding context-sensitivity. The authors apply the method to the cryptographic functions in a library that are secret-sensitive, which are typically in limited number that somewhat bounds explosion issues. The prioritization based on data-flow diversity that we propose may potentially help Constantine scale to bigger programs.

ProbeGuard [34] clones functions to provide hardened versions that can be hotpatched to protect programs from probing attacks. Control-flow integrity solutions leverage type or pointer analyses to enumerate the possible targets of a indirect branches, and restrict the code to follow one of them [73, 353, 364]: also in this setting, cloning functions may potentially improve the precision of the underlying analysis.

FIRestarter [33] provides an efficient crash recovery solution based on Software (STM) and Hardware (HTM) Transactional Memory, by cloning program parts to provide both the STM and HTM version. DynPTA [286] enhances a unification-based pointer analysis with context-sensitive heap modeling using function summaries to distinguish different allocation sites, ultimately treating them as virtual clones of the original function.

Calling Contexts. Programming language literature largely studied calling contexts and their portions (e.g., [9,21,328,345,387]). Due to their sheer number, a static enumeration of calling contexts is often unfeasible [345], and even space-efficient dynamic methods need wide identifiers to keep collisions low [45]. Furthermore, for complex programs, short executions often result in dozens of million distinct contexts [98, 99]. Unlike cloning, these techniques incur non-negligible temporal or spatial overheads, hindering an effective composition with local feedbacks used by fuzzers. Also, we have shown that full context-sensitivity can be unnecessarily inefficient when fuzzing, while selectivity can be much more effective.

3.8 Conclusion

In this chapter, we presented a novel approach to context-sensitivity in fuzzing which we term predictive context-sensitive fuzzing. Our proposal stems from the analysis of existing context-sensitive approaches, which track full calling contexts and allow context/edge hash collisions for the sake of a practical implementation. Such approaches face an impossible trade-off: allow too many collisions and lose context (but also path) sensitivity, allow too few and incur trashing behavior due to queue/map explosion.

With predictive context-sensitive fuzzing we show that it is possible to find the sweet spot by proactively selecting (and cloning) only the contexts that look more promising (as predicted by program analysis), forbidding unpredictable collisions and eliminating internal wastage. Our tests show that data-flow diversity can serve as an excellent predictor for such contexts, with significant coverage and bug-finding improvements compared to the most performant state-of-the-art solution (e.g., +9.8% total bugs, also different by 18.04%).

Fuzzing is a highly effective way to a wide range of security-relevant bugs. However, fuzzing can only find bugs that violate properties that the fuzzer checks, possibly through the use of sanitizers (e.g., out-of-bound accesses for ASan [320]). Thus, fuzzing will be able to find a precise subset of bugs, mainly defined by the sanitizers it uses. Sanitizers usually try to detect subtle memory corruptions or other common types of undefined behavior. In the next chapter, we introduce a sanitizer and a static analysis framework to detect subtle type confusion bugs that would elude the detection of state-of-the-art sanitizers.

Chapter 4

Uncovering Container Confusion Bugs in the Linux Kernel

4.1 Introduction

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

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

In more detail, an object downcast subtracts the offset of the parent field in the child object from the address available for the parent, yielding the address of its *container* structure (i.e., the child). The term container follows from the popular `container_of` macro pioneered by the Linux kernel. Issuing a downcast is not only always unsafe, but even not conforming to any C language standard [279]. Thus, the correctness and safety burden is on the shoulder of the developers, who have to guarantee through program semantics that the requested child type is correct. Failing to meet this requirement would cause a type confusion, which may have possibly disastrous consequences, such as a memory corruption vulnerability [229].

For object-oriented languages, runtime type information (RTTI) enables straightforward validation of downcasting operations. For example, current solutions that look for type confusion in C++ code rely on forms of RTTI tracking [113,121,152,200].

Solutions with provisions for C code can detect (some) cases of type confusion by intercepting heap allocations of objects and binding them with their top-level allocation type [113, 200] in userland code, whereas for kernels current mitigation proposals require costly manual rewriting to encode type information at every allocation site [121].

In this chapter, we take a systematic approach to discover type confusion vulnerabilities resulting from incorrect downcasting on structure embeddings, which we call *container confusion*. We design a new sanitizer that does away with runtime type tracking of objects and uses instead information on object allocation boundaries, which we obtain using an off-the-shelf solution. In more detail, we rely on redzones from memory sanitization literature [320] to augment allocation sites for out-of-bound access detection. Our sanitizer checks type compatibility for a downcasting operation by checking the relative position of the embedded parent structure, the outer child structure, and the redzones. This scheme transforms a type check in multiple straightforward structure bound checks, with low runtime overhead and no manual code changes.

We apply our sanitizer to the Linux kernel, one of the most complex and security-sensitive program instances. An initial study of its code base, which we conducted to gauge the potential bug surface, reveals more than 50,000 occurrences of `container_of` involving nearly 4,000 structure types. The type graph is also highly connected, with extreme cases such as `list_head` used as parent for over 1800 child types.

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

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

We responsibly reported our findings to the Linux kernel maintainers, who acknowledged them, and proposed patches for all the bugs we found. At the time of writing, 94 patches have been merged in the kernel. Our reports sparked valuable discussions which, among others, resulted in upgrading the C standard (to mitigate recurrent issues that we found) and in an attempt to change the list iterator integral to the kernel.

In sum, this chapter proposes the following contributions:

- We systematize a class of type confusion bugs, showing how C programs are affected by incorrect downcasting on structure embeddings. We dub it *container confusion*.
- We design a sanitizer for them that does away with type tracking and show its applicability to the Linux kernel.

- We derive 5 general patterns of container confusion from bugs we found in the kernel and design a static analyzer around them to make our approach scale in coverage.
- We evaluate our approach on a recent Linux kernel version, identifying 11 bugs with dynamic analysis (e.g., fuzzing) and another 78 bugs through our static analyzer.

Our sanitizer and static analyzer form a framework, UNCONTAINED, open-sourced at https://github.com/vusec/uncontained-sanitizer and https://github.com/vusec/uncontained-dataflow.

4.2 Background

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

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

Casting an object to an incompatible type violating casting rules (*i.e.*, bad-casting) causes *type confusion*. For instance, a static downcast in C++ checks only if the source and destination types are in the same type hierarchy, but not if the runtime destination type is the expected one. As a result, large C++ projects, such as the major browsers, parts of Windows, and the Oracle JVM [152], are rife with type confusion bugs.

Downcasting in C. The problem is not limited to object-oriented languages such as C++ but also extends to large programs written in C. Since C is not an object-oriented programming language, it does not support classes like C++. However, developers use *structure embedding* to benefit from an approximation of classes and inheritance. In particular, properties shared by multiple types are defined as a struct *embedded* in all the relevant types. In such a way, all the child types inherit the struct members declared in the parent type that is embedded. We show a simplified example of such use in Listing 4.1. Since the child type includes the parent type in this design, it is called a *container*.

Analogous to C++, we require primitives to go from the child type to its parent ("upcasting") and from the parent to its child type ("downcasting"). Upcasting is implemented by obtaining a pointer to the embedded parent structure from the child structure and is guaranteed safe. Downcasting is not defined in the C standard since it would require using a pointer to the parent structure to obtain a pointer outside of the memory defined by the type of the parent structure itself [279]. Still, many projects, including the Linux kernel, do exactly that. Given a pointer to the parent in a type hierarchy based on structure embedding, they implement their own version of downcasting, often in the form of a macro, that uses pointer arithmetic to calculate a pointer to the child type.

Such a macro is often named `container_of`. The reference implementation in the Linux kernel is shown in Listing 4.2. The `container_of` macro is not exclusive

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

Listing 4.1. Structure embedding example, where `gr_request` and `goku_request` "inherit" from `usb_request`.

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

Listing 4.2. `container_of` implementation in the Linux kernel.

to the Linux kernel but present in many large C projects such as Qemu, Nodejs, Xorg, the Windows kernel, git, FreeBSD, and XNU.

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

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

To iterate over a list, the kernel uses macros such as `list_for_each_entry`. It repeatedly follows the `next` pointer to find the next `list_head` and then uses `container_of` to set the iterator to the base of the entry that embeds it. For

instance, we can iterate over all inodes of a superblock as follows:

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

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

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

4.2.2 Sanitizers

Sanitizers are runtime tools to detect undefined behavior in programs, typically through compiler-based instrumentation that checks undefined behavior. The best-known example is AddressSanitizer (ASan) [320], which detects memory errors such as buffer overflows and use-after-frees. ASan instruments every memory access with a check that consults a shadow memory to see if the memory access is valid. In particular, to detect buffer overflows, ASan pads memory allocations with *redzones* and poisons the memory in the shadow memory (setting it to a nonzero value) so that any future access results in an ASan error. In this chapter, we will repurpose ASan redzones to detect object boundaries.

4.3 Container Confusion in the Linux Kernel

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

4.3.1 Security Implications

Like C++'s `static_cast`, the `container_of` macro does not perform runtime checks to verify whether the structure is actually contained within the expected outer structure. When this is not the case, container confusion leads the program to access memory under wrong assumptions on its layout. Two base scenarios are possible: a) the structure is embedded in a different container, leading to member access over memory contents typed for another layout; or b) the structure is not embedded in a container, leading to a pointer that is out-of-bounds by the relative offset assumed within the container.

The security implications of bad casting have been well-researched for C++ (e.g., in the CaVeR paper [229]) and similarly apply here, being `container_of`

```
1  static int gr_dequeue(struct usb_ep *_ep,
2                        struct usb_request *_req) {
3    struct gr_request *gr_req; // renamed: was `req`
4    ...
5    struct gr_ep *ep = ...; // derived from `_ep`
6    list_for_each_entry(gr_req, &ep->queue, queue) {
7      if (&gr_req->req == _req)
8        break;
9    }
10   if (&gr_req->req != _req) {
11     ret = -EINVAL;
12     goto out;
13   }
14   ...
15 }
```

Listing 4.3. Using the list iterator `gr_req` past its validity causes container confusion.

equivalent to C++'s static downcasting. Such effects can range from subtle state corruptions to controlled out-of-bounds accesses that attackers can evolve for exploit construction. The security risk is mainly dependent on structure layouts, for example when memory containing function pointers can be overwritten. To probabilistically mitigate these and other issues, the Linux kernel can randomize the layout of some structures at compile time [172]. While this can make exploitation less reliable, in some cases it may also turn an unexploitable bug into a security vulnerability. At the time of writing, only a few structure types (65 in the entire kernel) can undergo randomization: enabling it globally can be difficult as code may assume a specific layout for some structures, while others have layouts that are tuned for better performance [93].

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

4.3.2 Running Example

We discuss next our running example (Listing 4.3) involving the kernel USB stack to better illustrate container confusion.

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

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

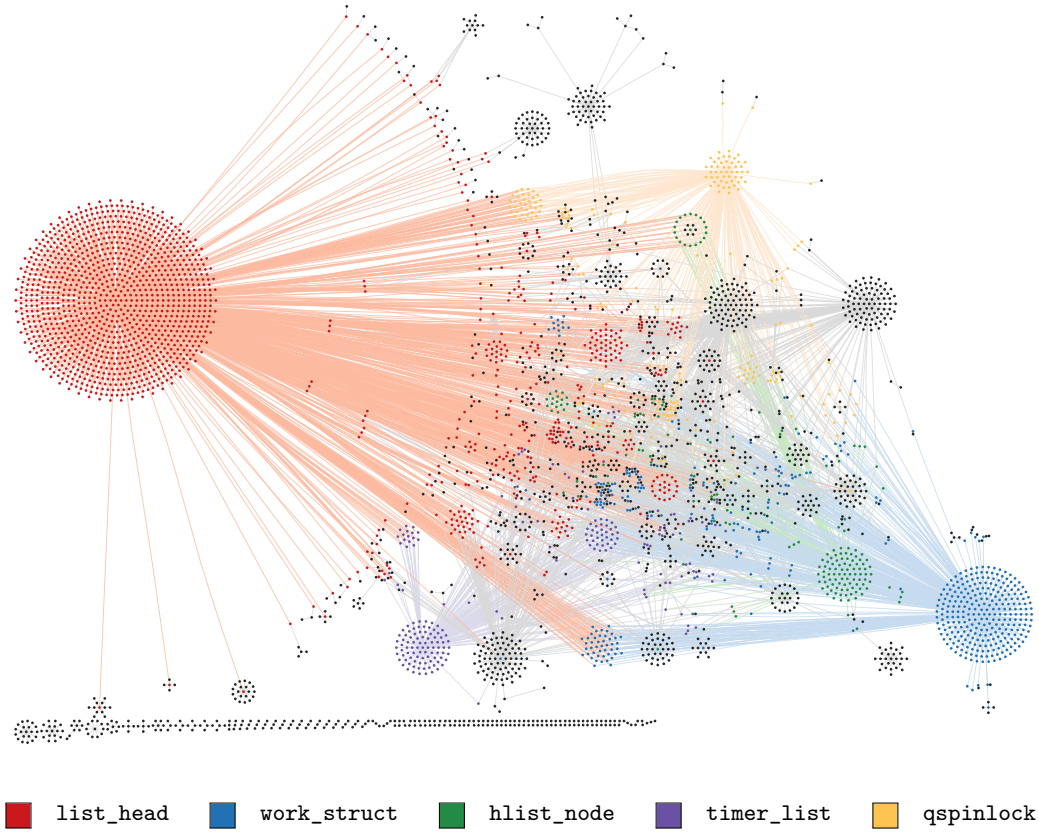


Figure 4.1. Type graph for `container_of` (and alike) instances.

within structure `*ep`.

The exploitability of the bug depends on the position of field `req`, used at line 10, within `gr_request` structures. Listing 4.1 shows the partial structure layout. Had the position been "deeper", the resulting pointer could have reached and surpassed the outer `gr_ep` structure, referencing the adjacent heap storage. Were `_req` to match such an out-of-bound pointer, the code attempts to remove a list entry that is not present, possibly causing further memory corruption.

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

4.3.3 Type Graph Complexity

To examine the use of structure embedding in the Linux kernel, we analyze the prevalence of `container_of` and its derivatives, as `container_of` takes part in several macros and inline functions. Depending on the selected kernel configuration, we note that the build system of the kernel can choose between different function implementations and even type definitions. Hence, we study the Linux kernel v.5.17 with the configuration in use to Google's syzbot [144] for continuous fuzzing.

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

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

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

For example, the `usb_request` structure shown in Listing 4.1 can be embedded in 17 different child structures in use to different USB drivers. Generally speaking, a variety of destination types may favor cases of invalid runtime downcasts.

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

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

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

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

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

4.4 UNCONTAINED Overview

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

In Section 4.5, we present a novel container confusion sanitizer that uses object boundaries to detect invalid downcasts during dynamic analysis. After describing the design and implementation, we evaluate effectiveness and performance of the sanitizer by combining it with the well-known syzkaller [373] kernel fuzzer and other

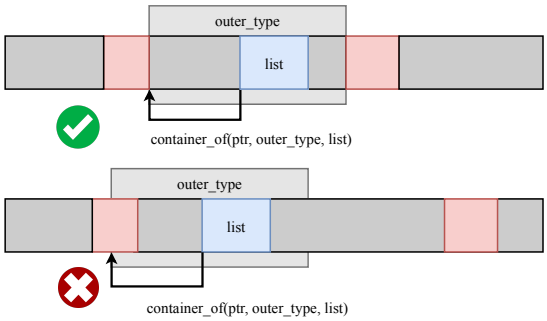


Figure 4.2. Redzone layout for a valid downcast (top) and for an invalid one (bottom). Here, *list* is the member field name.

benchmarks. Finally, we use the sanitizer to analyze the occurrence of container confusion in the Linux kernel.

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

4.5 Container Confusion Sanitizer

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

4.5.1 Design

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

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

```
1  static int gr_dequeue(struct usb_ep *_ep,
2                        struct usb_request *_req) {
3    struct gr_request *gr_req; // renamed: was `req`
4    ...
5    struct gr_ep *ep = ...; // derived from `_ep`
6    list_for_each_entry(gr_req, &ep->queue, queue) {
7      if (&gr_req->req == _req)
8        break;
9    }
10   if (!check_redzone(gr_req, sizeof(struct gr_request))) {
11     uncontained_report(gr_req);
12   }
13   if (&gr_req->req != _req) {
14     ret = -EINVAL;
15     goto out;
16   }
17   ...
18 }
```

Listing 4.4. Running example with our bound checks added.

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

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

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

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

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

Container Nesting. The bounds-checking policy we just presented may mishandle containers that are embedded in another container. For those cases, we cannot expect the presence of redzones for the inner container, being it a structure field. However, we can still do our validation through the outer container. In the Linux kernel, only 547 of its 4033 container types may incur such a scenario, whereas for 3486

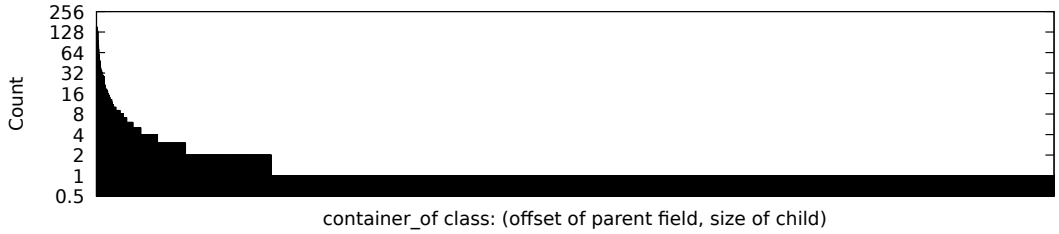


Figure 4.3. Distribution of `container_of` invocations according to offset of parent field and container size. Logarithmic scale.

no nesting is possible. Therefore, when the desired child type of a `container_of` instance is one of those 547, we apply the following scheme if the normal bound checks fail.

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

Time-of-use Checking. In the Linux kernel code, we found several cases where a `container_of` instance sees at runtime also objects of an incompatible type but the following code is never affected by the confusion. For example, with list iterators, the obtained child pointer was used only to access the parent again through the child field corresponding to it. These *anti-patterns* in the programming practice are not strictly bugs. Therefore, in our design, we opted to validate a `container_of` instance at the time of use for its output pointer rather than immediately when downcasting. Listing 4.4 shows our running example augmented with bound checks around redzones.

To identify uses of the output pointer, we run a standard intra-procedural def-use [155] analysis. As the program may modify it before dereferencing it (e.g., to access a child field), we analyze pointer arithmetic operations and, when the modification can be determined statically, we forward the check to the next use of the pointer. When the program dereferences it or we can no longer follow it statically, we emit bound checks and have them account for the modified offset, if any.

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

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

To look into this dimension, we identify a domain and a codomain for it. As

domain, we study how many unique `container_of` instances are present in the Linux kernel as we consider the pair (parent field, child type) for a downcast operation. We include the field as one child may embed multiple parents. As codomain, we identify pairs of the form (offset of parent field, size of child) for such operations, since these are the two quantities that we use—independently from one another—for bound checking. We count 6526 unique instances mapping to unique 3262 pairs. A collision occurs when two distinct instances map to the same pair.

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

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

4.5.2 Implementation

The sanitizer of UNCONTAINED consists of two components. The first one is a coccinelle [284] script to intercept occurrences of `container_of` at the source level, which the C preprocessor would otherwise expand before we may instrument them.

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

We also develop a framework[1] of potentially independent interest to apply custom LLVM passes during kernel compilation and run VMs for testing (e.g., with syzkaller) and debugging, automatically spawning one with a breakpoint attached to the found crash site for manual inspection in gdb.

To have full visibility on type information, we run our pass as a link-time optimization. We then leverage the existing redzone insertion and shadow memory mechanisms of Kernel Address Sanitizer (KASAN) [214] to support object boundary identification for stack, global, and heap-allocated variables. While our sanitizer can coexist with KASAN's machinery to sanitize memory accesses for safety violations, we disable its generation as these checks are unnecessary for our purposes.

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

We find two object allocation schemes that require special handling. One involves

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

a known limitation of redzones with arrays: in these cases, redzones cannot be inserted around their individual elements, unless one modifies the type definition. With a coccinelle script, we identify in the code base all the types that take part in array allocations and disable the validation of `container_of` instances using them as a child type. For future work, we are considering the addition of machinery to test all possible array cells when their number is known statically, whereas for dynamic sizes the recent proposal of bounded flexible C arrays [81] may be of help.

The second scheme involves the allocation of multiple, differently typed structures (e.g., `kalloc (sizeof(A) + sizeof(B), ...)`) followed by pointer extraction for each structure. While we should consider this an anti-pattern, it occurs frequently and we therefor devise a coccinelle script to disable the involved types from validation. However, for a few recurring cases and if code semantics allowed doing so safely, we manually split allocations and enable container confusion detection for types like `io_buffer` used in `io_uring` code or `net_device` private data in networking code.

Overall, for the two schemes, we disable validation for 13926 out of 56468 downcasts. We also highlight that the shadow memory and redzones of KASAN operate only after the early boot phase of the kernel. Heap objects allocated by the boot memory allocator `memblock` have no redzones: we identify and skip them using address range checks at runtime.

While we test and evaluate our sanitizer around the Linux kernel, the adaptations needed for other subjects would be limited. Redzone management for userland software is available in LLVM with AddressSanitizer [320], while kernels like FreeBSD and XNU have their own KASAN implementation.

4.5.3 Evaluation

We run our sanitizer on the Linux kernel v.5.17 (commit `c269497d248e`). For the fuzzing experiments, we use syzkaller (commit `9e8eaa75a18a`) and build two images compiled, respectively, with the default kernel configuration and the one in use to Google's syzbot [144], as it enables additional features. The choice is an attempt to slightly balance the exploration of code between pervasiveness and breadth.

To stress specific/additional components, we also run typical userland workloads such as installing programs with the `aptitude` package manager, executing `binutils` utilities, code for SGX enclaves, and the Linux Test Project [227].

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

4.5.3.1 Discovered Cases of Container Confusion

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

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

The 10 anti-patterns relate to places where a container confusion occurred but developers manage it explicitly later. As examples, we briefly describe two of the anti-patterns that our sanitizer found. The first involves the function `crypto_alg_lookup()` of the Kernel Crypto API. The function can return a pointer to a synchronous-hash structure (`shash_alg`) confused as if it were an asynchronous (`ahash_alg`) one. However, all the users of the function eventually check the requested instance type through additional fields to differentiate them and correctly cast the confused pointer before use. The second involves the `inet_lookup_established()` networking function, which can return a pointer to a `struct inet_timewait_sock` confused as a `struct sock`. Similar to above, all the users of the function check the socket state to differentiate them.

4.5.3.2 Runtime Overhead

We conduct two sets of experiments to measure the overhead introduced by the sanitizer component of UNCONTAINED: the bare sanitization costs with LMbench [257] and their impact on the end-to-end throughput when fuzzing with syzkaller.

We run the LMbench programs on a single QEMU-KVM instance with 8 GB of RAM executing on an i7-10700K CPU host machine with minimal background activity and identical software to the previous experiments. We repeat each experiment 10 times, taking the median value for every program. Our sanitizer introduces a geomean overhead of 74%. As a reference, KASAN introduces a 126% overhead (with 33% coming from redzone management, which we use too). We list figures for the individual programs in Appendix E.1.

For fuzzing throughput, we measure how many test cases one syzkaller VM executes within the first hour of fuzzing. We take the median value of 10 experiment repetitions, starting from an empty fuzzing corpus. The syzkaller baseline with no sanitizers enabled executed 80348 test cases, whereas with UNCONTAINED 69734 with a net reduction of the fuzzing throughput of around 13%. As a reference, KASAN introduced a 55% net reduction of the throughput. We find our approach to induce an overhead[2] acceptable for fuzzing.

4.6 Retrospective Analysis and Bug Patterns

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

[2]One opportunity to reduce it would be to follow [326] by disabling stack walking upon memory (de)allocation events, as it helps only for crash debugging/deduplication but is expensively frequent. Each crash may be analyzed offline by re-running the test case in an unmodified KASAN.

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

We find this to stem directly from the inherent coverage problem of dynamic tools. Much code may be locked under specific kernel states [153, 405], require emulation for crossing the hardware/software barrier with device drivers [291], or need complex input generation logic (e.g., with protocols). Special-purpose fuzzers [85, 285, 291, 311, 326, 334, 335, 349], which one may run naturally on our instrumented kernels, currently exist only for a fraction of such components.

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

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

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

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

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

Listing 4.5 reports an exemplary bug found when fuzzing in the `sock_init_data()` function while manipulating a `socket` struct. The function assumes that its `struct socket* sock` parameter is embedded in a `socket_alloc` container. This assumption is correct for most sockets in the kernel, except for TUN and TAP ones. Hence, when a program path from function `tun_chr_open()` reaches the buggy function, its argument is embedded in a `tun_file` container instead.

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

```
1  static int tun_chr_open(struct inode *inode, struct file *file) {
2    struct tun_file *tfile;
3    ...
4    sock_init_data(&tfile->socket, &tfile->sk);
5    ...
6  }
7
8  struct inode *SOCK_INODE(struct socket *socket) {
9    return &container_of(socket,
10     struct socket_alloc, socket)->vfs_inode;
11 }
12
13 void sock_init_data(struct socket *sock, struct sock *sk) {
14   if (sock) {
15     ...
16     sk->sk_uid = SOCK_INODE(sock)->i_uid;
17   } else {
18     ...
19   }
20   ...
21 }
```

Listing 4.5. The first argument to `sock_init_data()` is contained within `tfile` when called from `tun_chr_open()`. `SOCK_INODE()` incorrectly assumes `sock` to be contained within a `socket_alloc` struct.

```
1  static void inet_diag_msg_sctpasoc_fill(
2          struct inet_diag_msg *r,
3          struct sock *sk,
4          struct sctp_association *asoc) {
5    union sctp_addr laddr;
6    ...
7    laddr = list_entry(asoc->base.bind_addr.address_list.next,
8      struct sctp_sockaddr_entry, list)->a;
9    ...
10   if (sk->sk_family == AF_INET6) {
11     *(struct in6_addr *)r->id.idiag_src = laddr.v6.sin6_addr;
12     ...
13   }
14   ...
15 }
```

Listing 4.6. `list_entry()` assumes the presence of at least one entry within `asoc->base.bind_addr.address_list`, causing a container confusion in `inet_diag_msg_sctpasoc_fill` due to the missing check for whether the list is empty.

Pattern ❷: Empty-list Confusion. As we anticipated in Section 4.2.1, a confusion can originate when issuing a `container_of` operation on the owning structure of a circular list. When such a list is empty, the owning structure sees the `next` and `prev` fields of its embedded `list_head` point to itself. Accessing list members in a `list_entry`[3], `list_first_entry`, or `list_last_entry` operation causes container confusion.

Listing 4.6 reports an exemplary bug found in the kernel networking stack when fuzzing. Since the `inet_diag_msg_sctpasoc_fill()` function assumes that the `asoc->base.bind_addr.address_list` list is populated without checking for it,

```
1  void inet_bind_hash(struct sock *sk,
2          struct inet_bind_bucket *tb,
3    const unsigned short snum) {
4    ...
5    hlist_add_head(&sk->sk_bind_node, &tb->owners);
6    ...
7  }
8
9  int __inet_hash_connect(..., struct sock *sk, ...) {
10   ...
11   struct inet_bind_bucket *tb;
12   ...
13   if (port) {
14     ...
15     tb = inet_csk(sk)->icsk_bind_hash;
16     ...
17     if (hlist_entry((&tb->owners)->first,
18         struct sock, sk_node) == sk &&
19         !sk->sk_bind_node.next) {
20       inet_ehash_nolisten(sk, NULL, NULL);
21       spin_unlock_bh(&head->lock);
22       return 0;
23     }
24     ...
25   }
26   ...
27 }
```

Listing 4.7. `inet_bind_hash()` inserts list elements using the `sk_bind_node` member, whereas `__inet_hash_connect()` accesses them incorrectly using the `sk_node` member.

`laddr` points to a container-confused object when the `list_entry()` operates on an empty list. The code at line 11 copies some of its fields into memory provided to userspace. As these confused fields contain kernel heap pointers, this results in a KASLR leak that deterministically breaks the address randomization of the kernel, which often represents one of the first steps in kernel exploitation [150, 170, 194, 217]. At the time of disclosure, the bug had been present in the Linux kernel for almost 7 years.

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

Listing 4.7 shows an exemplary bug found when fuzzing involving the `sock` structure. A `struct sock` can be inserted in multiple lists by embedding multiple list structures. Among others, two single-linked lists use the fields `sk_bind_node` and `sk_node`. The socket code manages the `&tb->owners` list, which holds sockets using their `sk_bind_node` member. But `__inet_hash_connect()` accesses the same objects using the `sk_node` member. As the two members are located at different offsets, the downcast adjusts the pointer incorrectly, causing container confusion. As a result, the condition at line 17, which controls a fast path for the function, never evaluates to true. At the time of disclosure, the bug had been present in the Linux

```
1  void sgx_mmu_notifier_release(struct mmu_notifier *mn,
2                                struct mm_struct *mm) {
3    struct sgx_encl_mm *encl_mm = ...;
4    struct sgx_encl_mm *tmp = NULL;
5    ...
6    list_for_each_entry(tmp, &encl_mm->encl->mm_list, list) {
7      if (tmp == encl_mm) {
8        list_del_rcu(&encl_mm->list);
9        break;
10     }
11   }
12   ...
13   if (tmp == encl_mm) {
14     synchronize_srcu(&encl_mm->encl->srcu);
15     mmu_notifier_put(mn);
16   }
17 }
```

Listing 4.8. Incorrect use of the list iterator variable `tmp` after the loop in `sgx_mmu_notifier_release()`.

```
1  ...
2      ret = kobject_init_and_add(&f2fs_feat,
3        f2fs_feat_ktype,
4        NULL, "features");
5  ...
6  ssize_t f2fs_attr_show(struct kobject *kobj,
7                         struct attribute *attr, char *buf) {
8    struct f2fs_sb_info *sbi = container_of(kobj,
9              struct f2fs_sb_info,
10             s_kobj);
11   struct f2fs_attr *a = ...;
12   return a->show ? a->show(a, sbi, buf) : 0;
13 }
```

Listing 4.9. Invalid `container_of` on `kobj` (originating from `&f2fs_feat`) in `f2fs_attr_show()`.

kernel for 18+ years (i.e., the extent of its git history).

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

This container confusion characterized our running example (cf. Section 4.3.2). Listing 4.8 shows another exemplary bug that we found in SGX code when running an enclave in our instrumented kernel build using `qemu-sgx`. As the function processes an empty `&encl_mm->encl->mm_list` list, the `tmp` iterator is never assigned a valid entry, holding a confused pointer after the loop. At the time of disclosure, the bug had been present in the Linux kernel for more than 2 years.

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

Pattern ❺: Containers with Contracts. An object embedded in a data structure may come with additional metadata (e.g., custom RTTIs [229]) that program semantics uses as an implicit *contract* to control what operations can be done on it.

This is the case with the *sysfs* subsystem of the kernel, which lets userspace programs inspect and control several kernel features. Listing 4.9 shows a container confusion that we found in an inspection function when fuzzing. Here, the `kobject` that `kobject_init_and_add()` registers is not embedded in another structure, but the buggy `f2fs_attr_show()` function treats it as if embedded in a `f2fs_sb_info` structure.

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

Bug Counts. With our sanitizer (Section 4.5.3.1), we discovered 6 mismatches on data structure operators, 2 cases of empty-list confusion, and 1 case for each of the other patterns.

4.7 Static Analyzer

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

4.7.1 Design

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

For our analysis to scale to a code base as huge as the Linux kernel while maintaining satisfying accuracy, we make the following design choices. We cast bug pattern search to a static *information flow analysis* problem, relying on def-use information to track value propagation. The five bug patterns become rules for an on-demand backward or forward analysis where `container_of` instances act as sources or sinks depending on the pattern. We extend def-use chains through procedure boundaries (as a simplified form of [155]) and model memory as a single, coarse-grained symbolic location for scalability. We use semantic knowledge of common data structure manipulations (e.g., list iterators) to model several flows that involve indirection, enabling static reasoning.

We provide descriptions below for how we encode the five bug patterns as rules for the information flow analysis. Appendix E.2 contains more rigorous definitions of what we use as (and do at) sources, sinks, and path-discarding filters.

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

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

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

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

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

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

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

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

Pattern ❺. Confusion cases on containers with contracts are hard to spot in terms of code manipulations alone. We find it reasonable to assume that, for a given

code base, the identity of such container types is known. For the Linux kernel, we devise an analysis for `kobject` containers that one may in principle adapt to other types from other code bases. The analysis comes with a forward and a backward component.

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

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

4.7.2 Implementation

We implement the general forward and backward information flow analyses and the rules for patterns ❶, ❷, ❸, and ❺ as a pass for LLVM IR in 1286 lines of C++ code. We run it a link time so we can effectively extend def-use chains across procedure boundaries. In this scenario, though, LLVM would normally merge type definitions having an identical memory layout: to keep our analyses accurate, we disabled this behavior by changing ~25 lines of code in the compiler.

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

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

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

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

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

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

4.7.3 Evaluation

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

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

We run our static analyzer on the same kernel code base studied in Section 4.5.3. Table 4.1 summarizes the findings from a manual analysis of the reported cases of potential container confusion: we identified 80 bugs, 179 anti-patterns, and 107 false positives. We responsibly disclosed and proposed patches (138 in total with 91 already merged at the time of writing) for all the bugs as well as for the anti-patterns that can be removed without intrusive program semantics changes.

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

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

Pattern ❶. Reports about *Statically Incompatible Containers* cases include 3 bugs, 27 anti-patterns, and 72 false positives. This pattern is prone to false positives (67.3% of the total among all five patterns) due to imprecision of the static analysis: we found most of them to occur when some backward control flows are unfeasible as they are guarded by checks on fields carrying explicit type tags[4]. A similar semantics is also behind most of the anti-patterns we found. As for the bugs, static checking identifies the TUN bug from fuzzing that we discussed when presenting the pattern in Section 4.6, but also a similar variant for TAP socket interfaces.

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

For example, we found a container confusion in code that incorrectly checks HID device drivers reports, affecting all the 9 kernel drivers that rely on it. The bug had

[4]It could be a one-time effort to add such domain knowledge to the checker. However, we found 72 still feasible here for our manual analysis.

been present in the kernel for almost 9 years. In other HID driver code, we found 2 use-after-free and 1 NULL pointer dereference bugs. We also found a bug in the RT scheduler for an incorrect check on the task queue that had been present for 15 years.

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

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

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

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

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

The most immediate effect of our reporting and patching activity was upgrading the C standard for the Linux kernel to C11 [83]: this makes it possible to declare iterators valid only within loops, forcing developers to use (valid) retrieved values in a safer way. Shortly after, maintainers followed up with a proposal under adoption for a safer design of list iterators [84].

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

4.8 Discussion

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

The sanitizer component, thanks to precise runtime information, offers high accuracy by incurring only few false positives in our tests. This wealth of information also allows it to detect bugs that are out of reach of the static analyzer due to the

latter's inherent under-approximation (e.g., for cases of memory indirections that we cannot recover via domain knowledge). This can be seen in the limited overlap in the bugs found: only 2 of the 11 bugs found dynamically occur in the reports of the static analyzer.

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

To improve the reach of the static analyzer, precise modeling of memory may be an area worth examining. We opted not to use pointer analyses as accurate ones are expensive on large programs [344] and features desirable in this context like flow- and context-sensitivity would increase their costs considerably. Also, they would be unaware of the many indirect control transfers to functions caused by userland activities. We leave this investigation to future work.

Similarly, it would be interesting to explore directed fuzzing [43] and/or fuzzers specialized for certain kernel areas (Section 4.6) to reach functions/regions where static analyses report potential container confusion cases. Doing so may enable both their in-depth exploration and input generation for some reports, but we are not aware of any fuzzing system that one may use at the present time to explore this angle.

4.9 Related Work

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

Type Confusion Detection. Most existing type confusion detectors are limited to C++. UBSan [239], for instance, replaces static casts with dynamic casts in C++ to expose bugs. CaVeR [229], TypeSan [152], HexType [195], and Bitype [287] are specialized to find type confusion for C++ classes by managing runtime type metadata and performing checks on cast operations. CASTSan [269] efficiently detects type confusion leveraging C++ virtual tables, but is limited to polymorphic classes only. While all other existing approaches rely on dynamic analysis, TCD [407] uses a field-, context- and flow sensitive pointer analysis to detect type-confused C++ code.

libcrunch [200] and EffectiveSan [113] support C programs. However, both approaches rely on intercepting object allocations and binding them with their top-level allocation type. In practice, this would be hard, if not impossible, to collect in projects with the complexity of a kernel. For this reason, the typed allocator mitigation in XNU resorted to manual annotations in allocations [121]. Our approach overcomes the need of both allocation-time type inference and manual annotations.

Speculative Type Confusion. Previous work has explored speculative type confusion while dealing with objects of multiple types. Confusion in the speculative domain fundamentally differ from non-speculative one for observability and/or explainability. Kasper [197] scans the Linux kernel for arbitrary speculative gadgets. It shows how the current list iterator implementation is subject to speculative container confusion when dealing with the list heads if the terminating condition is mispredicted. Kirzner et al. [207] focus on speculative type confusion in the Linux kernel. The chapter highlights possible type confusion originating from eBPF code, compiler-introduced vulnerabilities, and polymorphic types. BHI [30] leverages a speculative type confusion in eBPF code in their exploit. FPVI [298] and Spook.js [2] exploit speculative type confusion in JavaScript engines.

Other Sanitizers. Similarly to ASan [320], several sanitizers rely on redzones: Purify [157], Memcheck [322], Dr. Memory [56] and LPC [156] leverage them to detect memory corruptions in the form of spatial and temporal safety violations.

MSan [340] targets reads from uninitialized memory using a shadow map mechanism. Other sanitizers, such as Undangle [58], FreeSentry [396], DangNull [228], and DangSan [363] detect dangling pointers that cause use-after-free errors.

For boundary identification, other techniques encode tracking metadata within pointers, as with low-fat pointers [114, 223] and delta pointers [219]. For example, our approach could replace redzones with low-fat pointers on supported systems.

Static Analyzers. We conclude by mentioning a few popular static analysis tools for the Linux kernel. Coccinelle [284] is pervasively used as a program matching and transformation tool. In addition to its use for refactoring and code hardening, it also has provisions to find intra-procedural bugs. Sparse [55] uses Linux kernel-specific annotations to perform few specialized checks. Smatch [54] followed in its footsteps to build a generic static analysis framework for several kernel bug types; it can only conduct intra-procedural dataflow analyses.

4.10 Conclusion

We presented a sanitization scheme for container confusion designed as a compiler-based runtime checker. For demonstration, we implemented the sanitizer for the Linux kernel, finding 11 bugs, which were undetected by previous work. Those bugs have often existed in the kernel for several years. Based on our results, we identified common bug patterns and used those categories to build a tailored static analyzer to discover bugs in code often unreachable by dynamic analysis. With our static analyzer, we unveiled 78 additional, previously undiscovered bugs. We conclude that bad downcasting is not only problematic in object-oriented programming languages but also occurs in large C projects, with serious security impact.

We have disclosed and proposed possible fixes for all found bugs and relevant anti-patterns to the Linux kernel mailing list, with a total of 143 patches and 94 already merged.

By extending the classes of bugs static and dynamic analysis may find, we focused on improving software against malicious inputs that may cause memory corruption. However, the absence of vulnerabilities does not guarantee the security of a software program. A prototypical example is cryptographic software, which needs stronger guarantees against attackers that may infer secret information the software is computing on. Side-channel attacks may leverage noticeable differences in how software executes on a given hardware to disclose secret information. In the next chapter, we design and develop a framework to automatically mitigate side-channel vulnerabilities during compilation.

Chapter 5

Automatic Side-Channel Resistance

5.1 Introduction

Protecting the confidentiality of security-sensitive information is a key requirement of modern computer systems. Yet, despite advances in software security engineering, this requirement is more and more challenging to satisfy in face of increasingly sophisticated microarchitectural side-channel attacks. Such attacks allow adversaries to leak information from victim execution by observing changes in the microarchitectural state (e.g., cache eviction), typically via timing measurements (e.g., memory access latency).

Such attacks have been shown practical in the real world with or without the assistance of CPU bugs. Examples in the former category are transient execution attacks such as Spectre [210], Meltdown [236], L1TF [358], and MDS [59,314,368]. Examples in the latter category are traditional cache attacks (e.g., FLUSH+RELOAD [394] and PRIME+PROBE [281]) against security-sensitive software victims such as crypto libraries. While the former are the focus of many mitigation efforts by vendors, for the latter the burden of mitigation lies entirely on the shoulders of software developers [185].

In theory, this is feasible, as side-channel attacks leak secrets (e.g., crypto keys) by observing victim secret-dependent computations (e.g., branch taken or array indexed based on a crypto key bit) via microarchitectural measurements. Hence, eliminating explicit secret-dependent code/data accesses from software—a practice generally known as *constant-time programming* [31]—is a viable avenue for mitigation. In practice, removing side-channel vulnerabilities from software is a daunting and error-prone task even for skilled developers. Not surprisingly, even production side channel-resistant implementations are riddled with flaws [90, 333].

To address this problem, much prior work has proposed solutions to automatically transform programs into their constant-time equivalents or variations [65, 134, 135, 171, 203, 237, 238, 247, 253, 265, 327, 339, 343, 370, 381, 382, 401–404]. Unfortunately, even the most recent solutions [301, 332, 389] offer limited security or compatibility guarantees, hindering their applicability to real-world programs.

In this chapter, we introduce Constantine, a compiler-based system for the

automatic elimination of side-channel vulnerabilities from programs. The key idea is to explore a radical design point based on *control and data flow linearization* (or CFL and DFL), where all the possible secret-dependent code/data memory accesses are always executed regardless of the particular secret value encountered. The advantage of this strategy is to provide strong security and compatibility guarantees by construction. The nontrivial challenge is to develop this strategy in a practical way, since a straightforward implementation would lead to *program state explosion*. For instance, naively linearizing secret-dependent branches that guard loop exits would lead to unbounded loop execution. Similarly, naively linearizing secret-dependent data accesses by touching all the possible memory locations would lead to an unscalable solution.

Our design is indeed inspired by radical and impractical-by-design obfuscation techniques such as the M/o/Vfuscator [72], which linearizes the control flow to collapse the program's control-flow graph into a single branchless code block with only data movement (i.e., x86 `mov`) instructions [206]. Each `mov` instruction uses an extra level of indirection to operate on real or dummy data depending on whether the code is running the intended side of a branch or not.

Revisiting such design point for side-channel protection faces several challenges. First, linearizing all the branches with `mov`-only code hinders efficient code generation in modern compilers and leads to enormous overheads. To address this challenge, Constantine only linearizes secret-dependent branches pinpointed by profiling information, allows arbitrary branchless instructions besides `mov`, and uses efficient indirect memory addressing to allow the compiler to generate efficient code. Second, the M/o/Vfuscator only linearizes the control flow and encodes branch decisions in new data flows, a strategy which would only multiply the number of secret-dependent data accesses. To address this challenge, Constantine couples CFL with DFL to also linearize all the secret-dependent data flows (generated by CFL or part of the original program).

Finally and above all, M/o/Vfuscator does not address state explosion. For example, it linearizes loop exits by means of invalid `mov` instructions, which generate exceptions and restart the program in dummy mode until the original loop code is reached. Other than being inefficient, this strategy originates new side channels (e.g., exception handling) that leak the number of loop iterations. To address state explosion, Constantine relies on carefully designed optimizations such as *just-in-time loop linearization* and *aggressive function cloning*. The former linearizes loops in the same way as regular branches, but adaptively bounds the number of iterations based on the original program behavior. The latter enables precise, context-sensitive points-to analysis which can strictly bound the number of possible targets at secret-dependent data accesses.

Collectively, our optimizations produce a scalable CFL and DFL strategy, while supporting all the common programming constructs in real-world software such as nested loops, indirect function calls, pointer-based accesses, etc. Our design not only addresses the state explosion problem, but also leads to a system that outperforms prior comprehensive solutions in terms of both performance and compatibility, while also providing stronger security guarantees. For example, we show Constantine yields overheads as low as 16% for cache-line attacks on standard benchmarks. Moreover, to show Constantine provides the first practical solution for automatic

side-channel resistance for real-world software, we present a case study on the wolfSSL embedded TLS library. We show CONSTANTINE-protected wolfSSL can complete a modular multiplication of a ECDSA signature in 8 ms, which demonstrates CONSTANTINE's automated approach can effectively handle a fully-fledged real-word crypto library component for the very first time.

Contributions

To summarize, this chapter proposes the following contributions:

- We introduce CONSTANTINE, a system for the protection of software from side channels.
- We show how CONSTANTINE can automatically analyze and transform a target program by efficiently applying control and data flow linearization techniques.
- We implement CONSTANTINE as a set of compiler transformations for the LLVM toolchain. CONSTANTINE is open source (available at https://github.com/pietroborrello/constantine).
- We evaluate CONSTANTINE on several standard benchmarks, evidencing its performance advantage against prior solutions. We also present a case study on the wolfSSL library to show its practical applicability on real-world software.

5.2 Background

Microarchitectural side channels generally allow an adversary to infer *when* and *where* in memory a victim program performs specific code/data accesses. And by targeting secret-dependent accesses originating from secret-dependent control and data flows in the original program, an adversary can ultimately leak secret data. Constant-time programming is a promising solution to eliminate such explicit secret-dependent accesses from programs, but state-of-the-art automated solutions are severely limited in terms of security, performance, and/or compatibility.

Control Flow Secret-dependent control flows (e.g., code branching on a crypto key bit) induce code accesses that microarchitectural attacks can observe to leak secrets. Early constant-time programming solutions only attempted to balance out secret-dependent branches with dummy instructions (e.g., with cross-copying [3]) to mitigate only simple execution time side-channel attacks [250]. Molnar et al. [265] made a leap forward with the *program counter security model* (PC-security), where the trace of secret-dependent executed instructions is the same for any secret value.

Prior work has explored two main avenues to PC-security. The first avenue is a form of *transactional execution* [301], which always executes both sides of every secret-dependent branch—hence a *real* and a *decoy* path—as-is, but uses a transaction-like mechanism to buffer and later discard changes to the program state from decoy paths. This approach provides limited security guarantees, as it introduces new side channels to observe decoy path execution and thus the secret. Indeed, one needs to at least mask exceptions from rogue operands of read/write

instructions on decoy paths, introducing secret-dependent timing differences due to exception handling. Even when normalizing such differences, decoy paths may perform read/write accesses that real paths would not make, introducing new decoy data flows. An attacker can easily learn data-flow invariants on real paths (e.g., an array always accessed at the same offset range) and detect decoy path execution when the observed accesses reveal invariant violations. Also, this approach alone struggles with real-world software compatibility. For instance, it requires loops to be completely unrolled, which leads to code size explosion for nested loops and for those with large trip count.

Another avenue to PC-security is *predicated execution* [82], which similarly executes both real and decoy paths, but only allows the instructions from the real path to update the program state. Updates are controlled by a predicate that reflects the original program branch condition and take the form of a constant-time conditional assignment instruction (e.g., `cmov` on x86) [82]. When on a decoy path, read/write operations get rewired to a single (conditionally assigned) shadow address. However, such decoy (shadow) data flows can again introduce new side channels to leak the decoy nature of a path [74, 82]. Moreover, this form of predication hampers the optimization process of the compiler, forcing the use of pervasive `cmov` instructions and constraining code transformation and generation. Some more recent solutions attempt to generate more optimized code by allowing some [332] or all [389] accesses on unmodified addresses on decoy paths. However, this hybrid strategy mimics transactional execution behavior and is similarly vulnerable to side-channel attacks that detect data-flow invariant violations on decoy paths. In addition, existing solutions face the same compatibility issues of transactional solutions with real-world code.

Unlike prior solutions, Constantine's control-flow linearization (CFL) executes both real and decoy paths using an indirect memory addressing scheme to transparently target a shadow address along decoy paths. This strategy does not force the compiler to use `cmov` instructions and yields more efficient generated code. For instance, a Constantine-instrumented wolfSSL binary contains only 39% of `cmov` instructions (automatically emitted by the code generator, as appropriate) compared to predicated execution, resulting in a net CFL speedup of 32.9%. As shown later, the addition of data-flow linearization (DFL) allows Constantine to operate further optimizations, eliminating shadow address accesses altogether as well as the corresponding decoy data flows. Constantine is also compatible with all the common features in real-world programs, including variable-length loops bound by means of *just-in-time linearization* and indirect calls handled in tandem with DFL.

Data Flow Data-dependent side channels have two leading causes on modern microarchitectures. Some originate from instructions that exhibit data operand-dependent latencies, e.g., integer division [82] and some floating-point instructions [12] on x86. Simple mitigations suffice here, including software emulation [13] (also adopted by Constantine), compensation code insertion [74], and leveraging hardware features to control latencies [82].

The other more fundamental cause stems from secret-dependent data flows (e.g., an array accessed at an offset based on a crypto key bit), which induce data

accesses that microarchitectural attacks can observe to leak secrets. Hardware-based mitigations [381] do not readily apply to code running on commodity processors. To cope with such leaks, existing compiler-based solutions have explored code transformations [389] and software-based ORAM [301].

SC-Eliminator [389] transforms code to preload cache lines of security-sensitive lookup tables, so that subsequent lookup operations can result in always-hit cache accesses, and no secret-dependent time variance occurs. Unfortunately, since the preloading and the secret-dependent accesses are not atomic, a non-passive adversary may evict victim cache lines right after preloading and later observe the secret-dependent accesses with a cache attack. Cloak [149] adopts a similar mitigation approach, but enforces atomicity by means of Intel TSX transactions. Nonetheless, it requires manual code annotations and can only support short-lived computations. Moreover, these strategies are limited to standard cache attacks and do not consider other microarchitectural attacks, including those that operate at the subcacheline granularity [262, 395].

Raccoon [301] uses Path ORAM (Oblivious RAM) [339] as a shortcut to protect data flows from attacks. ORAMs let code conceal its data access patterns by reshuffling contents and accessing multiple cells at each retrieval [143]. Unfortunately, this strategy introduces substantial run-time overhead as each security-sensitive data access results in numerous ORAM-induced memory accesses.

Unlike prior solutions, Constantine's data-flow linearization (DFL) eliminates all the explicit secret-dependent data flows (generated by CFL or part of the original program) by forcing the corresponding read/write operations to touch all their target memory locations as computed by static points-to analysis. While such analyses are known to largely overapproximate target sets on real-world programs, Constantine relies on an *aggressive function cloning strategy* to enable precise, context-sensitive points-to analysis and strictly bound the number of possible targets. For instance, a Constantine-instrumented wolfSSL binary using state-of-the-art points-to analysis [344] yields an average number of target objects at secret-dependent data accesses of 6.29 and 1.08 before and after aggressive cloning (respectively), a net reduction of 83% resulting in precise and efficient DFL. Unlike prior solutions that are limited to array accesses, Constantine is also compatible with arbitrary pointer usage in real-world programs.

Decoy-path side channels We use Listing 5.1 to show how existing constant-time protection solutions struggle to maintain both memory safety and real execution invariants along decoy paths, ultimately introducing new side channels for attackers to detect decoy paths.

The `if` condition at line 6 guards the statement at lines 7-9 (two read operations followed by one write operation). Let us consider the case $4096 <= \mathsf{secret} < 8192$. All the state-of-the-art solutions [82,301,332,389] would also run the corresponding decoy path (statements inside the condition, normally executed only when $\mathsf{secret} < 4096$), but with different code transformations. The approach of Coppens et al. [82] rewires the memory accesses at lines 7-9 to touch a shadow address, therefore allowing an attacker to detect decoy path execution by observing (three) accesses to the shadow address. SC-Eliminator [389] preloads both tables before executing the branch, but

```
1  char last_result;
2  char tableA[8192];
3  char tableB[4096];
4
5  char secret_hash(unsigned int secret) {
6    if (secret < 4096) {
7      register char tmp = tableB[secret];
8      tmp ^= tableA[secret];
9      last_result = tmp;
10   }
11   return last_result;
12 }
```

Listing 5.1. An example function vulnerable to control and data flow side channels.

executes the read/write operations at lines 7-9 with unmodified addresses, introducing a decoy out-of-bounds read at line 7. Such memory safety violation might cause an exception if the memory after tableB is unmapped, which, since the exception is left unmasked, would terminate the program and introduce a termination-based decoy-path side channel. Raccoon [301] closes such termination side channels by masking the exception, but this strategy also introduces an exception handling-based decoy-path side channel. The approach of Soares et al. [332], on the other hand, replaces such an unsafe read access with an access to a shadow address, which however introduces the same decoy-path side channel discussed for Coppens et al. [82].

Finally, even assuming no exception is caused by the out-of-bounds read at line 7 and that we can even eliminate the out-of-bounds behavior altogether without introducing other side channels, an attacker can still trivially detect decoy-path execution by side channeling the read at line 8. The shadow access of Coppens et al. [82] would leak decoy-path execution as discussed, but so will all the other solutions [301, 332, 389], which would allow an in-bound access at offset $4096 <= \text{secret} < 8192$ to tableA. Such access would never happen during real execution, breaking a program invariant on a decoy path and introducing a decoy data-flow side channel an attacker can use to again detect decoy-path execution.

In contrast, Constantine's combined CFL/DFL strategy would instead ensure the very same data accesses during real or decoy execution, preserving program invariants and eliminating decoy-path side channels by construction. Table 5.1 provides a detailed comparison between Constantine and prior solutions.

5.3 Threat Model

We assume a strong adversary able to run arbitrary code on the target machine alongside the victim program, including on the same physical or logical core. The adversary has access to the source/binary of the program and seeks to leak secret-dependent computations via microarchitectural side channels. Other classes of side channels (e.g., power [235]), software (e.g., external libraries/OS [53]) or hardware (e.g., transient execution [64,318]) victims, and vulnerabilities (e.g., memory errors or other undefined behaviors [380]) are beyond the scope of constant-time programming and subject of orthogonal mitigations. We further make no restrictions on the

Table 5.1. Technical, security, and compatibility features from state-of-the-art solutions vs. Constantine.

Feature	**Coppens et al.**	**Raccoon**	**SC-Eliminator**	**Soares et al.**	Constantine
control flows	predicated	transactional	hybrid	hybrid	linearization
data flows	-	Path ORAM	preloading	-	linearization
loop handling strategy	unroll	unroll	unroll	unroll	just-in-time
integration with compiler	backend	IR level	IR level	IR level	IR level
sensitive region identification	user annotations	annot. + static analysis	annot. + static analysis	user annotations	profiling (taint)
decoy-path side channels	shadow accesses	read/write accesses	read/write accesses	shadow, safe read/write accesses	no
fix variable-latency instructions (e.g., `div`)	no	sw emulation	no	no	sw emulation
threat model	code	code+data	code+data*	code	code+data
variable-length loops	no	no	decoy paths till bound	no	yes
indirect calls	no	-	-	-	yes
recursion	fixed-depth**	fixed-depth	-	-	yes
spatial safety preserved	yes	no	no	yes	yes
supported data pointers	-	arrays	arrays	arrays	no restrictions

** unimplemented * cache line with preloading

microarchitectural side-channel attacks attempted by the adversary, ranging from classic cache attacks [281, 394] to recent contention-based attacks [6, 147]. With such attacks, we assume the adversary can observe the timing of arbitrary victim code/data accesses and their location at the cache line or even lower granularity.

5.4 Constantine

This section details the design and implementation of Constantine. We first outline its high-level workflow and building blocks.

5.4.1 Overview

Constantine is a compiler-based system to automatically harden programs against microarchitectural side channels. Our linearization design pushes constant-time programming to the extreme and embodies it in two flavors:

1. **Control Flow Linearization** (CFL): we transform program regions influenced by secret data to yield secret-invariant instruction traces, with real and decoy parts controlled by a *dummy execution* abstraction opaque to the attacker;

2. **Data Flow Linearization** (DFL): we transform every secret-dependent data access (including those performed by dummy execution) into an oblivious operation that touches all the locations such program point can possibly reference, leaving the attacker unable to guess the intended target.

CFL and DFL add a level of *indirection* around value computations. The CFL dummy execution abstraction uses it to implicitly nullify the effects of instructions that presently execute as decoy paths. DFL instead wraps load and store operations to induce memory accesses for the program that are secret-invariant, also ensuring real and decoy paths access the same collections of objects.

Linearizing control and data flows represents a radical design point with obvious scalability challenges. To address them, Constantine relies on carefully designed optimizations. For control flows, we rely on a M/o/Vfuscator-inspired *indirect*

memory addressing scheme to legalize decoy paths while allowing the optimizer to see through our construction and generate efficient code. We also propose *just-in-time loop linearization* to efficiently support arbitrary loops in real-world programs and automatically bound their execution based on the behavior of the original program (i.e., automatically padding the number of iterations based on the maximum value observed on real paths).

For data flows, we devise *aggressive function cloning* to substantially boost the precision of static memory access analysis and minimize the extra accesses required by DFL. To further optimize DFL, we rely on an efficient object metadata management scheme and on hardware-optimized code sequences (e.g., AVX-512) to efficiently touch all the necessary memory locations at each secret-dependent data access. We also exploit synergies between control-flow and data-flow handling to (i) eliminate the need for shadow accesses on decoy paths (boosting performance and eradicating problematic decoy data flows altogether); (ii) handle challenging indirect control flows such as indirect function calls in real-world programs.

To automatically identify secret-dependent code and data accesses, we rely on *profiling* information obtained via dynamic information flow tracking and propagate the dependencies along the call graph. To analyze memory accesses, we consider a state-of-the-art Andersen-style points-to analysis implementation [344] and show how aggressive function cloning can greatly boost its precision thanks to newly added full context sensitivity.

From a security perspective, CFL ensures PC-security for all the instructions that operate on secret data or whose execution depends on it; in the process it also replaces variable-latency instructions with safe software implementations. DFL provides analogous guarantees for data: at each secret-dependent load or store operation, the transformed program obliviously accesses every potentially referenced location in the execution for that program point and is no longer susceptible to microarchitectural leaks by design.

Figure 5.1 provides a high-level view of the CFL, DFL, and support program analysis components behind Constantine. Our techniques are general and we implement them as analyses and transformation passes for the *intermediate representation* (IR) of LLVM.

5.4.2 Control Flow Linearization

With control flow linearization (CFL) we turn secret-dependent control flows into straight-line regions that meet PC-security requirements by construction [265], proposing just-in-time linearization for looping sequences. We also make provisions for instructions that may throw an exception because of rogue values along decoy paths, or yield variable latencies because of operand values.

> ***CFL**: The sequence of secret-dependent instructions that the CPU executes is constant for any initial input (PC-security) and data values do not affect the latency of each such instruction.*

With this invariant, only data access patterns can then influence execution time, and DFL will make them insensitive to secret input values. We assume that an oracle

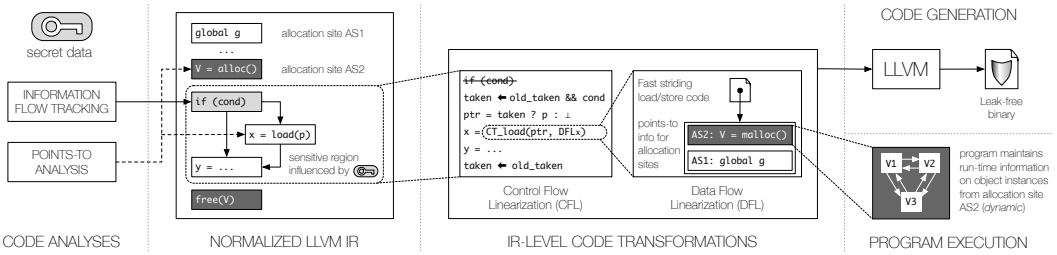


Figure 5.1. Architecture of CONSTANTINE: code analyses, CFL & DFL transformations, and run-time object metadata.

(the taint analysis of Section 5.4.4.1) enucleates which control-flow transfer decisions depend on secret data. Such information comprises if-else and loop constructs and indirect-call targets. For each involved code region, we push the linearization process in a recursive fashion to any nested control flows (i.e., if-else branches, loops, and function calls), visiting control-flow graphs (CFGs) and call graph edges in a post-order depth-first fashion. By doing so we avoid leaks from decoy paths when executing secret-independent inner branches in a protected region.

5.4.2.1 Dummy Execution

Each linearized region holds a *"taken"* predicate instance that determines if the original program would execute it (*real path*) or not (*decoy path*) under the current program state. We incrementally update the predicate with a new instance at every control-flow decision that guards the region in the original program, and let the compiler use the previous incoming instance upon leaving the region. The predicate backs a *dummy execution* indirection abstraction where we let decoy paths execute together with real paths, and use the *taken* predicate to prevent that visible effects from decoy paths may pollute the program state.

The key to correctness is that we can safely allow decoy paths to make local computations (i.e., assign to virtual registers in the IR), as long as their values do not flow into memory. For memory operations, each pointer expression computation selects an artificial $\perp$ value when in dummy execution. DFL primitives wrap every load and store instruction and make both real and decoy paths stride the same objects thanks to points-to metadata associated with the memory operation. Upon leaving a region, local values that the program may use later (i.e. *live* virtual registers) undergo a selection step to pick values from real paths at merge points.

The key to efficiency is using a selection primitive that is transparent for the optimizer thanks to indirection. As we observed in Section 5.2, the `cmov` selector typical of predicated execution constrains the behavior of the optimizer during code generation. We leverage the indirection on *taken* to design selection primitives based on arithmetic and logic operations that can instead favor optimizations.

Let us consider the pointer assignment $ptr = taken\ ?\ p : \perp$ of Figure 5.1. By modeling *taken* as an integer being 1 on real paths and 0 on decoy ones, and by using NULL to represent $\perp$ for DFL, the selection becomes $ptr = taken * p$. DFL helpers will prevent NULL accesses and deem them as from decoy paths: those cannot happen on real paths since, like prior literature [301], we work on error-

free programs. This constant-time multiplication-based scheme unleashes many arithmetic optimizations (e.g., global value numbering [308], peephole [4]) at the IR and backend level, bringing a net CFL speedup of 32.9% in wolfSSL over using the `cmov` approach. Appendix F.1 details other primitives that we evaluated.

Selection may be needed for (ϕ) compiler temporaries too, as we will detail in Section 5.4.2.3. Unlike memory addresses, both incoming values may be arbitrary, allowing for more limited optimization: for them we use the `select` IR instruction and let LLVM lower it branchlessly as it sees fit (including an x86 `cmov`).

Hereafter, we use `ct_select` to refer to a constant-time selection of any values, but we inline the logic in the IR in the implementation.

5.4.2.2 Compiler IR Normalization

Constantine takes as input the intermediate representation (IR) produced for the program by the language-specific compiler frontend. We assume that the IR comes in static single assignment (SSA) form [308] and that the CFG of every function containing regions to transform is reducible. The code can come in already-optimized form (e.g., `-O2`, `-O3` settings).

We apply a number of *normalization* passes that simplify later transformations with the ultimate goal of having *single-entry, single-exit regions* as unit of transformation, similarly to [389].

We use existing LLVM passes to lower switch constructs into if-else sequences, and to unify multiple function exit points into a single one (for abort-like sequences that do not fall through, we add artificial CFG edges to the exit node). As we work on error-free programs, we replace exception-aware `invoke` statements with normal calls. We also turn indirect calls into if-else sequences of direct calls using points-to information (Section 5.4.4.2), guarding each direct call with a pointer comparison on the target.

We then massage the CFG using standard compiler techniques [4] so that it results into a graph composed only of single-entry, single-exit regions: this will hold for all branches and loop constructs in the IR. This normalized IR is the input for the taint oracle of Section 5.4.4.1.

5.4.2.3 Branch Linearization

We can now detail how branch linearization operates and its orchestration with dummy execution. Under the single-entry, single-exit structural assumption from IR normalization, for a conditional construct of the likes *if (cond) then {A} else {B}*, we note that its exit CFG node post-dominates both the "then" and "else" regions of the branch, and is dominated by the entry node by construction. In SSA form, ϕ-nodes select incoming path-sensitive values. To linearize a conditional construct we:

1. remove the conditional branch, unlinking blocks A and B;
2. replace in A every pointer expression computation with a conditional assignment `ct_select`($cond, ptr, \perp$);
3. replace similarly in B, using the condition negated (!*cond*);

```
if (c_outer) {
  b1 = v[2]
} else {
  if (c_inner) {
    b2 = v[0]
  } else {
    b3 = 0
  }
  b_inner = φ(b2,b3)
}
b4 = φ(b1,b_inner)
v[1] = b4
```

(a) Original code

```
t0 = <incoming 'taken' predicate>
t1 = c_outer && t0
  ptr1 = ct_select(t1, &v[2], ⊥)
  b1 = ct_load(ptr1, DFL_b1)
t1−else = !c_outer && t0
  t2 = c_inner && t1−else
    ptr2 = ct_select(t2, &v[0], ⊥)
    b2 = ct_load(ptr2, DFL_b2)
  t2−else = !c_inner && t1−else // unused
    b3 = 0
  b_inner = ct_select(c_inner, b2, b3)
b4 = ct_select(c_outer, b1, b_inner)
ptr3 = ct_select(t0, &v[1], ⊥)
ct_store(ptr3, b4, DFL_store1)
```

(b) After linearization

Figure 5.2. Linearization and dummy execution.

4. wrap memory accesses with DFL `ct_{load, store}` primitives, supplying the DFL metadata for the operation (Section 5.4.3);

5. replace each ϕ-node $v_0 = \phi(v_A, v_B)$ in the exit block (which assigns virtual register v_0 according to whether A or B executed) with a conditional assignment `ct_select`$(cond, v_A, v_B)$;

6. merge ⟨entry, A, B, exit⟩ to form a single block, in this order.

We thus "sink" *cond* to conditionally assign pointers ($\perp$ for decoy paths) and virtual registers that outlive the region. Our transformation preserves the SSA form and can always be applied locally.

We can now add the dummy execution idea to the picture. Without loss of generality, let us consider two nested if-else statements that possibly take part in a larger linearized region as in Figure 5.2. When reaching the outer if construct, the program sees a *taken* predicate instance $\mathtt{t}_0$ that determines whether the execution reached the construct as part of a real ($taken = true$) or decoy computation.

Inside a region, IR instructions that assign virtual registers do not need to know $\mathtt{t}_0$. Path-sensitive assignments of live-out values from a region, such as $\mathtt{b}_{\mathtt{inner}}$, check the linearized conditions ($\mathtt{c}_{\mathtt{inner}}$ in this case). Memory-related instructions see instead their pointer expressions conditionally assigned according to some $\mathtt{t}_i$ *taken* instance. Those instances are updated upon entering the enclosing code block in the (original) program to reflect the combination of control-flow conditions with the incoming *taken* predicate.

5.4.2.4 Loop Linearization

To cope with the practical requirements of real-world code, with Constantine we explore a *just-in-time* approach for the linearization of loops. Let us consider the following secret-sensitive fragment, taken from a wolfSSL function that computes $x/R == x \ (mod\ N)$ using a Montgomery reduction:

```
_c   = c + pa;
tmpm = a->dp;
for (x = 0; x < pa+1; x++)
  *tmpm++ = *_c++;
for (; x < oldused; x++)   // zero any excess digits on
  *tmpm++ = 0;             // destination that we didn't write to
```

The induction variable x depends on secret data *pa*, outlives the first loop, and dictates the trip count of the second loop. Prior solutions struggle with each of these aspects, as well as with continue/break statements we found in wolfSSL. For the secret-dependent trip count issue, some [389] try to infer a bound and pad the loop with decoy iterations, then unroll the loop completely. However, high trip counts seen at run time or inaccurate bound predictions make unrolling immediately impractical due to code bloat.

In Constantine we design a new approach to handle loops that avoids unrolling and supports full expressivity for the construct. The key idea is to flank the normal trip count of a loop with an own CFL induction variable—dubbed *c_idx* next—and let such variable dictate just-in-time how many times that loop should execute.

```
base:
  i_base = 0
body:
  i_cur = φ(base: i_base,
            body: i_body)
  [...]
  i_body= i_cur+1
  [...]
  cond = ... // exit loop?
  br cond, out, body
out:
  x = i_body
```

(a) Original code

```
base:
  i_base = 0
body:
  i_cur = φ(base: i_base, body: i_body)
  i_real = φ(base: undef, body: i_out)
  i_body = i_cur+1
  i_out= ct_select(taken, i_body, i_real)
  [...]
  cond = ... // exit loop?
  cfl_cond = ... // CFL override
  br cfl_cond, out, body
out:
  x = i_out
```

(b) After linearization

Figure 5.3. Linearization with local variables outliving loops.

After IR normalization, a loop is a single-entry, single-exit region: its exit block checks some condition *cond* for whether the program should leave the loop or take the back-edge to the loop body. Note that break/continue statements are just branches to the exit node and we linearize them as in Section 5.4.2.3. Before entering the loop we set $c\_idx := 0$, and modify the exit block in such a way that the program still makes the original *cond* computation, but uses instead the current *c_idx* value to decide whether to leave the loop.

Say that we expect the program to execute the loop no more than k times (we address loop profiling in Section 5.4.4.1). At every iteration our exiting decision procedure increments *c_idx* by 1 and faces:

1. $taken = true \wedge cond = false$. The program is on a real path and wishes to take the back-edge to the body: we allow it;

2. *taken* = *true* ∧ *cond* = *true*. The program is on a real path and wishes to exit the loop: if $c\_idx = k$ we allow it, otherwise we enter dummy mode (*taken* := *false*) and the program will perform next $k - c\_idx$ dummy iterations for PC-security;

3. *taken* = *false*. We make the program leave the loop when $c\_idx = k$, and take the back-edge otherwise.

Note that for (3) we do not use the value of *cond*, as it can go rogue along decoy paths, but we still read it for the sake of linearization. Additionally, during (1) we validate the prediction of the oracle: whenever real program paths wish to iterate more than k times, we adaptively update k allowing the loop to continue, and use the k' seen on loop exit as the new bound when the program reaches the loop again. The handling of this comparison is also linearized.

Nested loops or linearized branches in loop bodies pose no challenge: we incrementally update the taken predicate and restore it across regions as we did for nested branches in Section 5.4.2.3 and Figure 5.2.

Let us resume the discussion of the code fragment. As variable x outlives the first loop, we should prevent decoy paths from updating it for the sake of correctness. If the compiler places x in memory, the IR will manipulate it using load and store instructions, and the dummy execution abstraction guarantees that only real paths can modify it. If instead it uses a virtual register v for performance, we flank it with another register v' conditionally assigned according to *taken*, and replace all the uses of v as operand in the remainder of the CFG with v'. Figure 5.3 shows this transformation with $\texttt{i}_{\texttt{body}}$ and $\texttt{i}_{\texttt{out}}$: decoy paths keep modifying $i_{\texttt{body}}$ for the sake of PC-security, but do not pollute the program state. Thanks to this design, we do not demote v to memory storage, which could harm performance especially for tight loops, nor we constrain the optimizer.

5.4.2.5 Operand Sanitization

As last step, we safeguard computations that could cause termination leaks from rogue values along decoy paths. In our design, this may happen only with divisions instructions receiving zero as divisor value. In Section 5.2 we noted that x86 integer division is also subject to variable latencies from operand values. We address both issues via software emulation, replacing `*div` and `*rem` LLVM instructions with subroutines that execute in constant-time, and for `*div` are also insensitive to rogue values.

5.4.2.6 Code Generation

Our CFL design poses no restrictions on code optimization as well as code generation operated in the backend. The optimizer can transform CFL-generated indirect memory references by means of optimizations such as common subexpression elimination and the code generator can lower such references using the most efficient patterns for the target architecture (including `cmov` instructions on occasion). However, we need to prevent the code generation process from inadvertently adding branches in branchless IR-level code. Indeed, this is not uncommon [90, 265]: luckily, modern compilers offer explicit support to preserve our constant-time invariants. In more

detail, we use LLVM backend options (e.g., `-x86-cmov-converter=0` for branchless lowering on x86) to control this behavior. As discussed later, we have also experimentally validated CONSTANTINE-instrumented binaries preserve our security invariants by means of a dedicated verifier.

5.4.3 Data Flow Linearization

With data flow linearization (DFL), we devise a new abstraction for controlling the data access patterns influenced by secret data, so that arbitrarily different (secret) inputs will lead to the same observable program behavior for an attacker. As we discuss in our security evaluation of Section 5.5, this design hardens against side-channel attacks that prior solutions cannot handle and it does not suffer from leaks through data-flow invariants and memory safety violations as we saw for such solutions in Section 5.2. Furthermore, thanks to its combination with points-to analysis, DFL is the first solution that does not place restrictions on pointer and object types, supporting for instance pointer-to-pointer casts that occur in real-world crypto code.

> ***DFL**: For every program point that performs a memory load or store operation, DFL obliviously accesses all the locations that the original program can possibly reference for any initial input.*

To support this invariant we conduct a context-sensitive, field-sensitive points-to analysis (described in Section 5.4.4.2) to build DFL metadata for each use of a pointer expression in a sensitive load or store instruction. Such metadata describes the portions of the object(s) that the expression may reference each time the program evaluates it. We assume that only program-allocated memory can hold secret-dependent data (external library calls cannot leak from Section 5.3).

For dynamic storage, that is stack- and heap-allocated objects, we instrument the involved allocation sites in the program to keep track at run time of the object instances currently stemming from an allocation site of interest to DFL (rightmost part of Figure 5.1).

DFL uses indirection around incoming pointer values: it obliviously accesses all the candidate object portions identified by the points-to analysis, and retrieves or modifies the memory value only within the object instance (if any) corresponding to the incoming pointer value. We apply DFL to every memory load or store made in a code region linearized by CFL (where the operation will see an incoming $\perp$ value when on a decoy path), and to memory operations that are outside input-dependent control flows but still secret-sensitive (e.g., array accesses with input-dependent index).

Unlike prior solutions, we do not need a shadow location for decoy paths (accessing it would leak the nature of such paths, Section 5.2), nor we let rogue pointers concur to memory accesses. Our design makes data accesses oblivious to secret dependencies and to the nature of control paths, and preserves memory safety in the process.

```
typedef struct dfl_obj_list {
    struct dfl_obj_list* next;
    struct dfl_obj_list* prev;
    struct dfl_obj_list** head_ptr;  // for fast removal from list
    unsigned long magic;   // to distinguish DFL heap objects
    unsigned char data[];  // contents of program object
} dfl_obj_list_t;
```

Figure 5.4. In-band metadata for data flow linearization.

5.4.3.1 Load and Store Wrappers

For the linearization of the data flow of accessed locations, we use `ct_load` and `ct_store` primitives for DFL indirection and resort to different implementations optimized for the storage type and the size of the object instances to stride obliviously. As we discussed when presenting the CFL stage, we accompany each use of a pointer expression in a load or store with DFL metadata specific to the program point.

DFL metadata capture at compilation time the *points-to information* for all the allocation sites of possibly referenced objects. The analysis comprises stack allocations, objects in global memory, and heap allocation operations. For each site, we use field-accurate information to limit striding only to portions of an object, which as a whole may hold thousands of bytes in real-world code.

Depending on the scenario, the user can choose the granularity λ at which memory accesses should become oblivious to an adversary. One may only worry about cache attacks (λ=64) if, say, only cross-core (cache) attacks are to be mitigated (e.g., with cloud vendors preventing core co-location across security domains by construction [292]). Or one may worry about arbitrary attacks if, say, core colocation across security domains is possible and attack vectors like MemJam ($\lambda = 4$) are at reach of the attacker.

Our wrapper implementations stride an object portion with a pointer expression incremented by λ bytes every time and which may match the incoming p input pointer from the program at most once. Depending on the object portion size, DFL picks between standard AVX instructions for striding, AVX2/AVX512 gather-scatter sequences to load many cache lines at once followed by custom selection masks, and a `cmov`-based sequence that we devise to avoid the AVX setup latency for small objects (details in Appendix F.2).

The DFL load and store wrappers inspect all the allocation sites from the metadata. For global storage only a single object instance exists; for stack and heap objects the instances may change during the execution, and the wrappers inspect the run-time metadata that the transformed program maintains (using doubly linked lists and optimizations that we describe in the next sections).

For a load operation, DFL strides all the object instances that the program point may reference and conditionally selects the value from the object portion matching the desired address. For decoy paths, no match is found and `ct_load` returns a default value.

For a store operation, DFL breaks it into a load followed by a store. The rationale is to write to every plausible program point's target, or the adversary may discover a secret-dependent write destination. For every object portion identified by DFL

store metadata, we read the current value and replace it with the contents for the store only when the location matches its target, otherwise we write the current value back to memory. Decoy paths "refresh" the contents of each object; real paths do the same for all but the one they modify.

5.4.3.2 Object Lifetime

DFL metadata supplied at memory operations identify objects based on their allocation site and characteristics. While global storage is visible for the entire execution, stack and heap locations have a variable lifetime, and we need to maintain run-time metadata for their allocation sites.

We observe that real-world crypto code frequently allocates large structures on the stack and pointers seen at memory operations may reference more than one such structure. At the LLVM IR level, stack-allocated variables take the form of `alloca` instructions that return a pointer to one or more elements of the desired type. The compiler automatically releases such storage on function return.

We interpose on `alloca` to wrap the object with *in-band metadata* information depicted in Figure 5.4. Essentially, we prepend the originally allocated element with fields that optimize DFL operations and preserve stack alignment: the program element becomes the last field of a variable-sized `dfl_obj_list_t` structure. Then, we assign the virtual register meant to contain the v pointer from `alloca` with the address of $v.data$ (32-byte offset on x64).

This transformation is simple when operating at the compiler IR level: unlike binary rewriting scenarios [103], the compiler is free to modify the stack layout while preserving program semantics, including well-behaved pointer arithmetics. Upon `alloca` interposition, we make the program update the run-time allocation site information and a symmetric operation happens on function exit.

Heap variables see a similar treatment. We interpose on allocation operations to widen and prepend the desired object with in-band metadata, with the address of $v.data$ returned to the program instead of the allocation base v. The $v.magic$ field is pivotal for handling `free()` operations efficiently: when interposing on them, we may witness a `dfl_obj_list_t` structure or a "standard" object from other program parts. We needed an efficient means to distinguish the two cases, as `free()` operations take the allocation base as input: for DFL objects we have to subtract 32 from the input pointer argument. We leverage the fact that allocators like the standard libc allocator `ptmalloc` prepend objects with at least one pointer-sized field. Hence accessing a heap pointer p as $p - 8$ is valid: for DFL objects it would be the address of the *magic* field and we check its peculiar value to identify them.

5.4.3.3 Optimizations

One advantage of performing DFL at compiler IR level is that we can further optimize both the data layout to ease metadata retrieval and the insertion of our DFL wrappers.

We identify functions that do not take part in recursive patterns and promote to global variables their stack allocations that sensitive accesses may reference. The promotion is sound as such a function can see only one active stack frame instance

at a time. The promotion saves DFL the overhead of run-time bookkeeping, with faster metadata retrieval for memory operations as we discuss next. To identify functions apt for promotion, we analyze the call graph of the program (made only of direct calls after IR normalization) to identify strongly connected components from recursion patterns [397] and exclude functions taking part in them.

We also partially inline DFL handlers, as object allocation sites are statically known from points-to analysis. For global storage, we also hard-code the involved address and striding information. For instance, a load operation from address *ptr* becomes in pseudo-code:

```
res = 0
res |= dfl_glob_load(ptr, glob1, stride_offset_g1, stride_size_g1)
res |= dfl_glob_load(ptr, glob2, stride_offset_g2, stride_size_g2)
res |= dfl_load(ptr, objs_as1, stride_offset_as1, stride_size_as2)
res |= dfl_load(ptr, objs_as2, stride_offset_as2, stride_size_as2)
res |= dfl_load(ptr, objs_as3, stride_offset_as2, stride_size_as2).
```

This is because the oracle determined that *ptr* may reference (portions of) global storage *glob1*, *glob2* or objects from allocation sites *as1*, *as2*, *as3*, where $objs\_as_i$ is the pointer to the data structure (a doubly linked list of objects, as with AS2 in Figure 5.1) maintained at run time for the allocation site (Section 5.4.3.2). With the OR operations we perform value selection, as each `dfl_` helper returns 0 unless the intended location *ptr* is met during striding. In other words, instead of maintaining points-to sets for memory operations as data, we inline their contents for performance (saving on retrieval time) and leave the LLVM optimizer free to perform further inlining of `dfl_` helpers code. The treatment of store operations is analogous.

Finally, we devise an effective (Section 5.7) striding optimization for loops. We encountered several loops where the induction variable flows in a pointer expression used to access an object, and from an analysis of its value (based on LLVM's *scalar evolution*) we could determine an invariant: the loop would be touching all the portions that require DFL striding and a distinct portion at each iteration. In other words, the code is "naturally" striding the object: we can avoid adding DFL striding and thus save on $n(n-1)$ unnecessary accesses.

5.4.4 Support Analyses

The compatibility of Constantine with real-world code stems also from two "oracles" as we tailor robust implementations of mainstream program analysis techniques to our context: an *information flow tracking* technique to identify program portions affected by a secret and a *points-to analysis* that we enhance with context sensitivity to obtain points-to sets as accurate as possible.

5.4.4.1 Identifying Sensitive Program Portions

Control and data flow linearization need to be applied only to regions affected by secret data, as protecting non-leaky code only hampers performance.

We assume the user has at their disposal a profiling suite to exercise the alternative control and data flow paths of the crypto functionality they seek to protect. Developers can resort to existing test suites for libraries, actual workloads, or program testing tools (e.g. generic [132] or specialized [160] fuzzers) to build one.

We then use DataFlowSanitizer (DFSan), a dynamic information flow tracking solution for LLVM, to profile the normalized IR of Section 5.4.2.2 over the profiling suite. DFSan comes with taint propagation rules for virtual registers and program memory and with APIs to define taint source and sink points. We write taint source configurations to automatically taint data that a program reads via I/O functions (e.g., a key file) and use as sink points conditionals, memory load/store operations, and `div`/`rem` instructions in the normalized IR. In the DFSan-transformed IR we then encode rules in the spirit of FlowTracker [306] to handle implicit flows among virtual registers, leaving those possibly taking place through memory to complementary tools like FTI [136].

We aggregate DFSan outputs to build a set of branches and memory accesses that are secret-dependent, feeding it to CFL and DFL. As we mentioned in Section 5.4.2, CFL will then push the hardening process to nested flows, linearizing their control and data flows. During the execution of the profiling suite we also profile loop trip counts that we later use as initial predictions for CFL (Section 5.4.2.4).

5.4.4.2 Points-to Analysis

Points-to analyses [330] determine the potential targets of pointers in a program. Nowadays they are available off-the-shelf in many compilation systems, with inclusion-based approaches in Andersen style [10] typically giving the most accurate results. In Constantine, we extend the Andersen-style analysis of the popular SVF library for LLVM [344]. For each pointer usage in the program, we use this analysis to build the *points-to set* of objects that it may reference at run time. Typically, points-to analyses collapse object instances from a dynamic allocation site into an abstract single object. Hence, points-to sets contain information on object allocation sites and static storage locations.

Points-to analyses are sound. However, they may overapproximate sets by including objects that the program would never access at run time. In a lively area of research, many solutions feature inclusion-based analyses as the approach is more accurate than the alternative, faster unification-based one [338]. Inclusion-based analyses could give even more accurate results if they were to scale to *context sensitivity*, i.e., they do not distinguish the uses of pointer expressions (and thus potentially involved objects) from different execution contexts. The context is typically intended as call-site sensitivity, while for object-oriented managed languages other definitions exist [259, 331]. To optimize the performance of DFL, we need as accurate points-to sets as possible, so in Constantine we try to restore context sensitivity in an effective manner for a sufficiently large codebase such as the one of a real-world crypto library.

Aggressive Cloning We use function cloning to turn a context-insensitive analysis in a context-sensitive one. A calling context [98] can be modeled as an acyclic path on the call graph and one can create a function clone for every distinct calling context encountered during a graph walk. This approach can immediately spin out of control, as the number of acyclic paths is often intractable [99, 387].

Our scenario however is special, as we may clone only the functions identified as secret-dependent by the other oracle, along with their callees, recursively. We thus

explore *aggressive cloning* along the maximal subtrees of the call graph having a sensitive function as root. The rationale is that we need maximum precision along the program regions that are secret-dependent, while we can settle for context-insensitive results for the remainder of the program, which normally dominates the codebase size.

Aggressive cloning turns out to be a key performance enabler, making DFL practical and saving on important overheads. As we discuss in Section 5.7, for wolfSSL we obtain points-to sets that are ~6x smaller than the default ones of SVF and very close to the run-time optimum. The price that we trade for such performance is an increase in code size: this choice is common in much compiler research, both in static [271] and dynamic [69] compilation settings, for lucrative optimizations such as value and type-based specialization.

Refined Field Sensitivity A field-sensitive analysis can distinguish which portions of an object a pointer may reference. Real-world crypto code uses many-field, nested data structures of hundreds or thousands of bytes, and a load/store operation in the program typically references only a limited portion from them. Field-accurate information can make DFL striding cheaper: this factor motivated our practical enhancements to the field-sensitive part of SVF.

The reference implementation fails to recover field-precise information for about nine-tenths of the wolfSSL accesses that undergo DFL, especially when pointer arithmetics and optimizations are involved. We delay the moment when SVF falls back to field-insensitive abstract objects and try to reverse-engineer the structure of the addressing so to fit it into static type declarations of portions of the whole object. Our techniques are inspired by *duck typing* from compiler research; we cover them in Appendix F.3. Thanks to these refinements, we could recover field-sensitive information for pointers for 90% of the sensitive accesses in our case study.

Indirect Calls Points-to analysis also reveals possible targets for indirect calls [344]. We use this information during IR normalization when promoting them to if-series of guarded direct calls (Section 5.4.2.2), so to remove leaks from variable targets. We refine the candidates found by SVF at call sites by matching function prototype information and eliminating unfeasible targets. Indirect call target information is also necessary for the aggressive cloning strategy.

5.4.5 Discussion

Constantine implements a compiler-based solution for eliminating microarchitectural side channels while coping with the needs of real-word code. We chose LLVM for its popularity and the availability of mature information-flow and points-to analyses. Nonetheless, our transformations are general and could be applied to other compilation toolchains. Similarly, we focus on x86/x64 architectures, but multiplexing conditional-assignment and SIMD striding instructions exist for others as well (e.g. ARM SVE [341] , RISC-V "V" [305]).

Moreover, operating at the compiler IR level allows us to efficiently add a level of indirection, with *taken* unleashing the optimizer and with DFL making memory accesses oblivious to incoming pointers. In addition, aggressive function cloning

allows us to transform the codebase and unveil a significantly more accurate number of objects to stride. The IR also retains type information that we can leverage to support field sensitivity and refine striding.

The just-in-time strategy to linearize secret-dependent unbounded control flows (loops) allows us to dodge intractability with high bounds and code bloat with tractable instances [332]. For points-to set identification and indirect call promotion, our analyses yield very accurate results (i.e., closely matching the run-time accesses) on the programs we consider. We leave the exploration of a just-in-time flavor for them to future work, which may be helpful in non-cryptographic applications.

The main shortcoming of operating at the IR level is the inability to handle inline assembly sequences found in some crypto libraries. While snippets that break constant-time invariants are uncommon, they still need special handling, for instance with annotations or lifting. Verification-oriented lifting [304], in particular, seems a promising avenue as it can provide formally verified C equivalent representations that we could use during IR normalization.

As the programs we study do not exercise them, for space limitations we omit the treatment of recursion and multithreading. Appendix F.4 details the required implementation extensions.

5.5 Security Analysis

This section presents a security analysis of our transformations. We start by arguing that instrumented programs are semantically correct and induce secret-oblivious code and data access traces. We then discuss how our design emerges unscathed by traditional passive and active attacks and examine the residual attack surface.

Correctness and Obliviousness Correctness follows directly from our design, as all our transformations are semantics-preserving. In short, for control flows, real paths perform all and only the computations the original program would make. For data flows, values from decoy paths cannot flow into real paths and correctness properties such as memory safety are preserved. Appendix F.5 provides informal proofs for these claims.

We now discuss how our linearization design yields *oblivious* code and data access traces. For code accesses, PC-security follows by CFL construction, as we removed conditional branches, loops see a fixed number of iterations (we discuss wrong trip count predictions later), and IR normalization handles abort-like sequences. For data accesses, we wrapped load and store operations with DFL machinery that strides portions of every abstract (i.e., by allocation site) object that the operation may access, *independently of the incoming pointer value*. For dynamic storage, for any two secrets, the program will see identical object collections to maintain at run-time: the composition of the lists can vary during the execution, but identically so for both secrets. Finally, for virtual registers that are spilled to memory by the backend, the CPU reads and writes them with the same instructions regardless of the current *taken* predicate value, so those accesses are also oblivious.

Security Properties We build on the obliviousness claims above to show that both passive attacks (attackers only monitoring microarchitectural events) and active attacks (attackers also arbitrarily tampering with the microarchitectural state) are unsuccessful.

No instruction latency variance from secret-dependent operand values is possible, since we replace and sanitize instructions such as division (Section 5.4.2.5). Memory accesses may have variable latencies, but, thanks to the DFL indirection, those will only depend on non-secret code/data and external factors. Moreover, DFL wrappers do not leak secrets and do not introduce decoy paths side channels in terms of decoy data flows or exceptions. In Section 5.4.3.1, we explained how load and store helpers stride objects using safe [82, 301] `cmov` or SIMD instructions. As for decoy paths, *taken* can conditionally assign an incoming pointer with $\perp$: the adversary would need access to CPU registers or memory contents to leak the nature of a path (outside the threat model). Finally, helpers are memory-safe as points-to analysis is sound and we track object lifetimes.

Finally, an active attacker may perturb the execution to attempt Flush+Reload, Prime+Probe, and other microarchitectural attacks to observe cache line-sized or even word-sized victim accesses. With vulnerable code, they could alter for instance the access timing for a specific portion of memory, and observe timing differences to detect matching victim accesses. However, thanks to the obliviousness property of our approach, leaking victim accesses will have no value for the attacker, because we access all the possible secret-dependent code/data locations every time.

Residual Attack Surface We now discuss the residual attack surface for Constantine. Design considerations aside, the correctness and obliviousness of the final instrumented binary are also subject to the correctness of our Constantine implementation. Any implementation bug may introduce an attack surface. To mitigate this concern, we have validated our correctness claims experimentally by running extensive benchmarks and test suites for the programs we considered in our evaluation. We have also validated our obliviousness claims experimentally by means of a verifier, as detailed later. Overall, our implementation has a relatively small trusted computing base (TCB) of around 11 KLOC (631 LOC for our profiler, 955 LOC for CFL, 2561 for DFL, and 7259 LOC for normalization and optimization passes), which provides confidence it is possible to attain correctness and obliviousness in practice.

Constantine's residual attack surface is also subject to the correctness of the required oracle information. The static points-to analysis we build on [344] is sound by design and our refinements preserve this property—barring again implementation bugs. Our information-flow tracking profiler, on the other hand, relies on the completeness of the original profiling suite to eliminate any attack surface. While this is a fundamental limitation of dynamic analysis, we found straightforward to obtain the required coverage for a target secret-dependent computation, especially in cryptographic software. Implementation bugs or limitations such as implicit flows (Section 5.4.4.1) also apply here. A way to produce a more sound-by-design oracle is to adopt static information-flow tracking, but this also introduces overtainting and hence higher overheads [301].

An incomplete suite might also underestimate a secret-dependent loop bound. Thanks to just-in-time linearization correctness is not affected, but every time the trip count is mispredicted (i.e., real-path loop execution yields a higher count than the oracle), the adversary may observe a one-off perturbation (given that the instrumentation quickly adapts the padding). This is insufficient to leak any kind of high-entropy secret, but one can always envision pathological cases. Similar considerations can be applied to recursive functions.

In principle, other than statically unbound secret-dependent control flows, one can also envision statically unbound secret-dependent data flows such as a secret-dependent heap-allocated object size. We have not encountered such cases in practice, but they can also be handled using just-in-time (data-flow) linearization—i.e., padding to the maximum allocation size encountered thus far during profiling/production runs with similar characteristics.

Part of the residual attack surface are all the code/data accesses fundamentally incompatible with linearization and constant-time programming in general. For instance, on the CFL front, one cannot linearize imbalanced if-else constructs that invoke system calls, or more generally secret-dependent code paths executing arbitrary library/system calls. Their execution must remain conditional. A way to reduce the attack surface is to allow linearization of idempotent library/system calls or even to include some external library/system code in the instrumentation. On the DFL front, one cannot similarly linearize secret-dependent data accesses with external side effects, for instance those to a volatile data structure backed by a memory mapped I/O region (e.g., a user-level ION region [366]). Again, we have not encountered any of these pathological cases in practice.

Similarly, Constantine shares the general limitations of constant-time programming on the compiler and microarchitectural optimization front. Specifically, without specific provisions, a compiler backend may operate optimizations that inadvertently break constant-time invariants at the source (classic constant-time programming) or IR (automated solutions like Constantine) level. Analogously, advanced microarchitectural optimizations may inadvertently re-introduce leaky patterns that break constant-time semantics. Some (e.g., hardware store elimination [111]) may originate new instructions with secret-dependent latencies and require additional wrappers (and overhead). Others (e.g., speculative execution [210]) are more fundamental and require orthogonal mitigations.

5.6 Performance Evaluation

This section evaluates Constantine with classic benchmarks from prior work to answer the following questions:

1. What is the impact of our techniques on compilation time?
2. How is binary size affected by linearization?
3. What are the run-time overheads induced by CFL and DFL?

Methodology We implemented Constantine on top of LLVM 9.0 and SVF 1.9 and tested it on a machine with an Intel i7-7800X CPU (Skylake X) and 16 GB of RAM running Ubuntu 18.04. We discuss two striding configurations to conceal

memory access patterns with DFL: word size ($\lambda = 4$), reflecting (core colocation) scenarios where recent intra cache level attacks like MemJam [262] are possible, and cache line size ($\lambda = 64$), reflecting the common (cache attack) threat model of real-wold constant-time crypto implementations and also CONSTANTINE's default configuration. We use AVX512 instructions to stride over large objects. Complete experimental results when using AVX2 and the $\lambda = 1$ configuration (presently out of reach for attackers) are further detailed in Appendix F.6. We study:

- 23 realistic crypto modules manually extracted by the authors of SC-Eliminator [389] from a 19-KLOC codebase (SCE suite), used also in the evaluation of Soares et al. [332];
- 6 microbenchmarks used in the evaluation of Raccoon [301] (Raccoon suite)—all we could recover from the source code of prior efforts [237, 238]—using the same input sizes;
- 8 targets used in constant-time verification works: 5 modules of the pycrypto suite analyzed in [385] and 3 leaky functions of BearSSL and OpenSSL studied in Binsec/Rel [90].

For profiling, we divide an input space of 32K elements in 128 equal partitions and pick a random instance from each, producing a profiling input set of 256 inputs. We build both the baseline and the instrumented version of each program at (`-O3`). Table 5.2 presents our full experimental datasets with the SCE suite (first five blocks) and the Raccoon, pycrypto, and Binsec/Rel suites (one block each).

Validation We validated the implementation for PC-security and memory access obliviousness with two verifiers. For code accesses, we use hardware counters for their total number and a cycle-accurate software simulation in GEM5. For data accesses, we use `cachegrind` for cache line accesses and write a DBI [94] tool to track what locations an instruction accesses, including predicated `cmov` ones visible at the microarchitectural level. We repeatedly tested the instrumented programs in our datasets with random variations of the profiling input set and random samples of the remaining inputs. We found no visible variations.

Compilation Time To measure CONSTANTINE-induced compilation time, we applied our instrumentation to all the programs in our datasets and report statistics in Table 5.2. The first four data columns report the sensitive program points identified with taint-based profiling over the randomly generated profiling input set. For the SCE programs, we protect the key scheduling and encryption stages. For brevity, we report figures after cloning and after secret-dependent pushing to nested flows (Section 5.4.2): the former affected `des3` and `loki91`, while the latter affected `applied-crypto/des`, `dijkstra`, `rsort`, and `tls-rempad-luk13`. Interestingly, for `3way`, the LLVM optimizer already transformed out a secret-sensitive branch that would be visible at the source level, while no leaky data flows are present in it (consistently with [389]).

Across all 37 programs, the average dynamic analysis time for taint tracking and loop profiling was 4s, with a peak of 31.6s on `libgcrypt/twofish` (~1 C KLOC). For static analysis (i.e., points-to), CFL/DFL transformations, and binary

Table 5.2. Benchmark characteristics and overheads.

		IR constructs (sensitive/total)				performance		binary size	
	program	branches	loops	reads	writes	$\lambda = 4$	$\lambda = 64$	$\lambda = 4$	$\lambda = 64$
CHRONOS	aes	0/1	0/1	224/235	0/68	1.13x	1.08x	1.16x	1.16x
	des	0/1	0/1	318/362	0/36	1.19x	1.14x	1.37x	1.73x
	des3	0/3	0/3	861/1005	0/89	1.49x	1.36x	1.92x	2.84x
	anubis	0/1	0/1	776/1240	0/87	1.29x	1.12x	1.27x	1.27x
	cast5	0/1	0/1	333/372	0/36	1.13x	1.06x	1.16x	1.16x
	cast6	-	-	192/204	0/4	1.13x	1.08x	1.01x	1.01x
	fcrypt	-	-	64/74	0/18	1.04x	1.03x	1.01x	1.01x
	khazad	-	-	136/141	0/1	1.13x	1.09x	1.15x	1.15x
S-CP	aes_core	-	-	160/192	0/16	1.12x	1.06x	1.22x	1.22x
	cast-ssl	0/1	0/1	333/355	0/54	1.23x	1.10x	1.24x	1.24x
BOTAN	aes	0/12	0/6	452/525	0/153	1.05x	1.03x	1.36x	1.72x
	cast128	0/2	0/2	333/374	0/52	1.02x	1.01x	1.16x	1.16x
	des	0/1	0/1	136/185	0/24	1.01x	1.01x	1.16x	1.16x
	kasumi	0/7	0/7	96/174	0/18	1.01x	1.01x	1.29x	1.57x
	seed	0/6	0/6	320/360	0/41	1.02x	1.01x	1.18x	1.18x
	twofish	1/8	0/6	2402/2450	4/1084	1.14x	1.12x	1.45x	2.42x
APP-CR	3way	0/4	0/4	0/8	0/14	1.00x	1.00x	1.00x	1.00x
	des	2/10	0/6	134/182	2/17	1.24x	1.09x	1.23x	1.45x
	loki91	16/76	24/28	16/24	0/6	1.51x	1.43x	1.02x	1.02x
LIBGCRYPT	camellia	-	-	32/48	0/48	1.02x	1.01x	1.01x	1.01x
	des	0/2	0/2	144/195	0/12	1.06x	1.06x	1.29x	1.85x
	seed	0/4	0/1	200/265	0/18	1.18x	1.10x	1.22x	1.22x
	twofish	-	-	2574/2662	0/1080	1.97x	1.92x	1.43x	2.24x
RACCOON	binsearch	1/4	1/2	1/3	0/2	1.33x	1.18x	1.01x	1.01x
	dijkstra	3/15	0/5	5/10	3/7	3.45x	1.51x	1.01x	1.01x
	findmax	0/2	0/2	0/1	0/1	1.00x	1.00x	1.00x	1.00x
	histogram	0/2	0/2	1/2	1/1	2.66x	1.68x	1.01x	1.01x
	matmul	0/5	0/5	0/2	0/2	1.00x	1.00x	1.00x	1.00x
	rsort	0/9	4/6	6/8	4/4	1.87x	1.84x	1.30x	1.30x
PYCRYPTO	aes	0/11	0/5	96/223	0/59	1.13x	1.06x	1.19x	1.19x
	arc4	0/3	0/3	3/30	2/10	1.07x	1.03x	1.01x	1.01x
	blowfish	0/16	0/12	24/77	0/39	5.07x	3.17x	1.01x	1.01x
	cast	0/29	0/2	284/321	0/57	1.09x	1.04x	1.37x	1.37x
	des3	0/5	0/1	32/40	0/7	1.06x	1.04x	1.01x	1.01x
B/REL	tls-rempad-luk13	4/17	1/1	6/14	4/17	1.01x	1.01x	1.02x	1.02x
	aes_big	0/45	0/8	32/141	0/40	1.01x	1.01x	1.29x	1.29x
	des_tab	0/50	0/28	8/164	0/97	1.04x	1.02x	1.29x	1.29x
AVG (GEO)	SCE suite	-	-	-	-	1.16x	1.11x	1.22x	1.35x
	Raccoon suite	-	-	-	-	1.68x	1.33x	1.05x	1.05x
	pycrypto suite	-	-	-	-	1.48x	1.30x	1.10x	1.10x
	Binsec/Rel suite	-	-	-	-	1.02x	1.01x	1.19x	1.19x
	all programs	-	-	-	-	1.26x	1.16x	1.17x	1.25x

generation, the end-to-end average time per benchmark was 1.4s, with a peak of 23s on `botan/twofish` (567 C++ LOC). Our results confirm CONSTANTINE's instrumentation yields realistic compilation times.

Binary Size Next, we study how our instrumentation impacts the final binary size. Two design factors are at play: cloning for the sake of accurate points-to information

and DFL metadata inlining to avoid run-time lookups for static storage. Compared to prior solutions, however, we save instructions by avoiding loop unrolling.

During code generation, we leave the choice of inlining AVX striding sequences to the compiler, suggesting it for single accesses and for small stride sizes with the `cmov`-based method of Appendix F.2—we observed lower run-time overhead from such choice. When we use word-level striding ($\lambda = 4$), the binary size is typically smaller than for cache line-level striding ($\lambda = 64$), as the AVX helpers for fast cache line accesses feature more complex logics.

As shown in Table 5.2, the average binary size increment on the SCE suite is around 1.35x in our default configuration ($\lambda = 64$) and 1.22x for $\lambda = 4$. For `des3`, we observe 1.92-2.84x increases mainly due to cloning combined with inlining. Smaller increases can be noted for the two `twofish` variants, due to DFL helpers inlined in the many sensitive read operations. The binary size increase for all the other programs is below 2x. The Raccoon programs see hardly noticeable differences with the exception of `rsort`, for which we observe a 1.3x increase. We note similar peak values in the two other suites, with a 1.37x increase for `cast` in pycrypto and 1.29x for `aes_big` and `des_tab` in Binsec/Rel. Our results confirm Constantine's instrumentation yields realistic binary sizes.

Run-time Performance Finally, we study Constantine's run-time performance. To measure the slowdown induced by Constantine on our benchmarks, we measured the time to run each instrumented program by means of CPU cycles with thread-accurate CPU hardware counters (akin [389]). We repeated the experiments 1,000 times and report the mean normalized execution time compared against the baseline. Table 5.2 presents our results.

Constantine's default configuration produces realistic overheads across all our benchmarks, for instance with a geomean overhead of 11% on the SCE suite and 33% on the Raccoon programs. These numbers only increase to 16% and 68% for word-level protection. Our SCE suite numbers are comparable to those of SC-Eliminator [389] and Soares et al. [332] (which we confirmed using the artifacts publicly released with both papers, Appendix F.6), despite Constantine offering much stronger compatibility (i.e., real-world program support) and security (i.e., generic data-flow protection and no decoy path side channels) guarantees. On the Raccoon test suite, on the other hand, Raccoon reported two orders-of-magnitude slowdowns (up to 432x) on a number of benchmarks, while Constantine's worst-case slowdown in its default configuration is only 1.84x, despite Constantine again providing stronger compatibility and security guarantees (i.e., no decoy path side channels). Overall, Constantine significantly outperforms state-of-the-art solutions in the performance/security dimension on their own datasets, while providing much better compatibility with real-world programs. For the two other suites, we observe modest overheads with the exception of `blowfish`: its 3.17-5.07x slowdown originates in a hot tight loop making four secret-dependent accesses on four very large tables, a pathological case of leaky design for automatic repair.

5.7 Case Study

The wolfSSL library is a portable SSL/TLS implementation written in C and compliant with the FIPS 140-2 criteria from the U.S. government. It makes for a compelling case study for several reasons.

From a technical perspective, it is representative of the common programming idioms in real-world programs and is a complex, stress test for any constant-time programming solution (which, in fact, none of the existing solutions can even partially support). As a by-product, it also allows us to showcase the benefits of our design.

The library supports Elliptic Curve (EC) cryptography, which is appealing as it allows smaller keys for equivalent guarantees of non-EC designs (e.g. RSA) [357]. EC Digital Signature Algorithms (ECDSA) are among the most popular DSA schemes today, yet their implementations face pitfalls and vulnerabilities that threaten their security, as shown by recent attacks such as LadderLeak [17] (targeting the Montgomery ladder behind the EC scalar multiplication in ECDSA) and CopyCat [263] (targeting the vulnerable hand-crafted constant-time (CT) wolfSSL ECDSA code).

In this section, we harden with CONSTANTINE the `mulmod` modular multiplication procedure in ECDSA from the non-CT wolfSSL implementation. This procedure calculates a curve point $k \times G$, where k is a crypto-secure nonce and G is the EC base point. Leaks involving k bits have historically been abused in the wild for, e.g., stealing Bitcoin wallets [38] and hacking consoles [128].

Code Features and Analysis The region to protect comprises 84 functions from the maximal tree that `mulmod` spans in the call graph. We generate a profiling set of 1024 random inputs with 256-bit key length and identify sensitive branches, loops, and memory accesses (Table 5.3). The analysis of loops is a good example of how unrolling is unpractical. We found an outer loop iterating over the key bits, then 1 inner loop at depth 1, 4 at depth 2, and 3 at depth 3 (all within the same outer loop). Every inner loop iterates up to 4 times, resulting in a nested structure—and potential unroll factor—of 61,440. And this calulation is entirely based on profiling information, the inner loops are actually unbounded from static analysis.

Similarly, cloning is crucial for the accuracy of DFL. We profiled the object sets accessed at each program point with our DBI tool (Section 5.6). With cloning, on average, a protected access over-strides (i.e., striding bytes that the original program would not touch) by as little as 8% of the intended storage. Without cloning, on the other hand, points-to sets are imprecise enough that DFL needs to make as many as 6.29x more accesses than strictly needed.

Overheads Table 5.3 presents our run-time performance overhead results, measured and reported in the same way as our earlier benchmark experiments. As shown in the table, the slowdown compared to the original non-CT baseline of wolfSSL (using the compilation parameter W=1) is 12.7x, which allows the CONSTANTINE-instrumented version to complete a full run in 8 ms. The compilation parameter W allows the non-CT version to use different double-and-add interleavings over the key bits as part of its sliding window-based double-and-add approach to implement ECC multiplication. In brief, a higher W value trades run-time storage (growing exponentially with W) with steady-state throughput (increasing linearly

Table 5.3. Characteristics and overheads for wolfSSL.

	baseline	w/o cloning	w/ cloning
functions	84	84	864
binary size (KB)	39	135 (3.5x)	638 (16.35x)
exec cycles (M)	2.6	200 (77x)	33 (12.7x)
accessed objs/point	1	6.29	1.08

	tainted	w/o cloning - nested flows	w/ cloning - nested (tainted)
branches	13	39	1046 (118)
loops	12	31	863 (139)
reads	33	138	2898 (52)
writes	1	91	1892 (2)

	time (ms)	cycles (M)	binary size (KB)
wolfSSL (W=4)	0.35	1.6	39
wolfSSL (W=1)	0.57	2.6	39
wolfSSL (const. time)	0.7	2.9	47
CONSTANTINE (W=1)	8	33	638

with W), but also alters the code generated, due to snowball optimization from inlined constants. This choice turns out to be cost-effective in the non-CT world, but not for linearization.

For completeness, we also show results for the best configuration of the non-CT version (which we profiled to be W=4) and the hand-written CT version of wolfSSL. The non-CT code completes an ECC multiplication in 0.35 ms in its best-performing scenario, while the hand-written CT version completes in 0.7 ms. Our automatically hardened code completes in 8ms, that is within a 11.42x factor of the hand-written CT version, using 11.38x more CPU cycles, yet with strong security guarantees for both control and data flows from the articulate computation (i.e., 84 functions) involved.

In terms of binary size increase, with cloning we trade space usage for DFL performance. We obtain a 16.36x increase compared to the reference non-CT implementation, and 13.57x higher size than the CT version. The performance benefits from cloning are obvious (77x/12.7x=6.06x end-to-end speedup) and the size of the binary we produce is 638 KB, which, in absolute terms is acceptable, but amenable to further reduction. In particular, the nature of wolfSSL code is tortured from a cloning perspective: it comprises 36 arithmetic helper functions that we clone at multiple usage sites. We measured, however, that in several cases they are invoked in function instances (which now represents distinct calling contexts for the original program) that see the same points-to information. Hence, after cloning, one may attempt merging back functions from calling contexts that see the same points-to set, saving a relevant fraction of code boat without hampering DFL performance.

Other optimizations, such as our DFL loop optimization also yields important benefits, removing unnecessary striding in some loops—without it, the slowdown would more than double (27.1x). We conclude by reporting a few statistics on analysis and compilation time. The profiling stage took 10m34s, the points-to analysis 20s (~2s w/o cloning), and the end-to-end code transformation and compilation process

1m51s (31s for the non-CT reference).

Overall, our results confirm that Constantine can effectively handle a real-world crypto library for the first time, with no annotations to aid compatibility and with realistic compilation times, binary sizes, and run-time overheads. Constantine's end-to-end run-time overhead, in particular, is significantly (i.e., up to two orders of magnitude) lower than what prior comprehensive solutions like Raccoon [301] have reported on much simpler benchmarks.

During our analysis of the hand-written CT version of wolfSSL for ECDSA modular multiplication we found a side-channel that exposed secret information in during the computation. We reported the vulnerability, which was assigned CVE-2020-11713. Our analysis uses the fixed version of wolfSSL.

5.8 Conclusion

We have presented Constantine, an automatic constant-time system to harden programs against microarchitectural side channels. Thanks to carefully designed compiler transformations and optimizations, we devised a radical design point—complete linearization of control and data flows—as an efficient and compatible solution that brings security by construction, and can handle for the very first time a production-ready crypto library component.

Our framework mitigates side-channel attacks at the software level by controlling the compilation pipeline to ensure no side-channel is present while running on modern hardware. However, software mitigations cannot address CPU vulnerabilities that may cause information leaks. Several microarchitectural vulnerabilities have been discovered in recent years [210, 236, 368], heavily undermining the isolation guarantees provided by current CPUs. In the next chapter, we explore the relationship between software and hardware bugs, and architectural and microarchitectural vulnerabilities, focusing on their root cause. We highlight how they often share similar root causes and investigate empty spots. By investigating them, we discover, exploit, and propose mitigations for the first architectural CPU vulnerability that leaks data without side channels.

Chapter 6

Architectural CPU Vulnerabilities

6.1 Introduction

In recent years, a lot of research has been conducted to improve software security, both on the application layer as well as on the operating-system (OS) layer [193, 348]. The types of software vulnerabilities are well known and, e.g., categorized with the Common Weakness Enumeration (CWE) [351]. In addition to manual security analysis, there are several techniques to discover software vulnerabilities in automated and semi-automated ways, e.g., fuzzing [132, 399], or static and dynamic analysis [88, 274]. However, more recent works have shown that next to software vulnerabilities, there are software-exploitable hardware vulnerabilities, such as Meltdown [236] or Spectre [210]. These vulnerabilities can undermine software security which always assumes bug-free and secure hardware. The discovery of transient-execution attacks [61, 210, 236] showed that CPUs by virtually all vendors, including Intel, AMD, and ARM, are affected by these software-exploitable hardware vulnerabilities. However, as these vulnerabilities are architecturally not visible, transient-execution attacks use side channels to observe and exploit them architecturally.

Following Meltdown and Spectre, a multitude of transient-execution attacks has been discovered in this class of vulnerabilities [61, 205, 216, 314, 358, 368, 386]. All of these attacks leak data, Meltdown-type attacks even across security boundaries, including trusted execution environments (TEEs). Hence, they pose a severe threat to the system security and resulted in numerous ad-hoc mitigations on the operating-system and firmware level [60, 178]. Despite the significant amount of research on transient-execution attacks, they are not the only CPU vulnerabilities. Architectural bugs have been known for much longer, with infamous examples such as the Pentium FDIV bug [75] or the Pentium F00F bug [79]. These vulnerabilities are intuitively easier to observe as they do not require additional side channels. However, recent work has highlighted the difficulty of adequately testing CPU design for such vulnerabilities [101].

In this chapter, we systematically analyze both architectural and transient-execution vulnerabilities, showing that the underlying type of vulnerability is often the same. While the CWE recently introduced categories for such hardware vulnera-

bilities, we show that the root cause of hardware vulnerabilities can also be classified using the existing vulnerability types for software. As CPUs are also written in (hardware) programming languages, it is indeed not surprising that vulnerabilities known from software are also present in hardware. However, we mainly see complex vulnerabilities in the hardware, such as race conditions or use after free.

Based on our systematic analysis, we investigate categories in which transient-execution attacks are known, but no architectural equivalent is known. Specifically, we systematically inspect CPUs for improperly initialized storage locations that return (parts of) stale data. We focus on data loads where the structure that holds the data is larger than the effective loaded data. Not initializing the whole structure may leave stale data in the region not overridden by the effective data. This is, e.g., the case in the I/O address space, where memory-mapped devices often have strict limitations, such as only allowing aligned 32-bit loads to specific addresses [182].

Discovering architectural leaks. The scan of the I/O address space on Intel CPUs based on the Sunny Cove microarchitecture revealed that the memory-mapped registers of the local Advanced Programmable Interrupt Controller (APIC) are not properly initialized. As a result, architecturally reading these registers returns stale data from the microarchitecture. Any data transferred between the L2 and the last-level cache can be read via these registers. This vulnerability, named ÆPIC Leak, affects the 10^{th} generation mobile Ice Lake CPUs, the newest, 12^{th} generation, Alder Lake CPUs, and the current 3^{rd} generation of Xeon scalable server CPUs (Ice Lake SP).

As the I/O address space is only accessible to privileged software, ÆPIC Leak targets Intel's TEE, SGX. ÆPIC Leak can leak data from SGX enclaves that run on the same physical core. While ÆPIC Leak would represent an immense threat in virtualized environments, hypervisors typically do not expose the local APIC registers to virtual machines, eliminating the threat in cloud-based scenarios. Similar to previous transient-execution attacks targeting SGX [314,358,367–369], ÆPIC Leak is most effective when running in parallel to the enclave on the sibling hyperthread. However, ÆPIC Leak does not require hyperthreading and can also leak enclave data if hyperthreading is unavailable or disabled.

We present two new techniques to leak *data in use*, *i.e.*, values from enclave registers, and *data at rest*, *i.e.*, data stored in enclave memory. With Cache Line Freezing, we introduce a technique putting targeted pressure on the cache hierarchy without overwriting stale data. Cache Line Freezing exploits the observation that Sunny Cove implements an optimization for zero cache lines, *i.e.*, cache lines filled only with '0's. These cache lines still appear to travel through the cache hierarchy, but they do not overwrite stale data. With this targeted pressure and enclave single-stepping [360], we leak register values from cache lines in the secure state area (SSA). A second technique, Enclave Shaking, exploits the capability of the operating system to securely swap enclave pages. By alternatingly swapping enclave pages out and back in, the stored data is forced through the cache hierarchy, allowing ÆPIC Leak to leak the values without even continuing the execution of the enclave. We exploit ÆPIC Leak in combination with Cache Line Freezing and Enclave Shaking to extract AES-NI keys and RSA keys from Intel's IPP library and the Intel SGX

sealing and remote attestation keys. Our attack leaks memory from enclaves with 334.8 B/s and a success rate of 92.2 %.

Although we provide software workarounds for specific scenarios, such as AES-NI, we conclude that there is no short-term workaround for protecting enclave data without disabling the APIC memory-mapped range or disabling SGX. On January 2022, Intel announced the deprecation of SGX on the affected CPU generations [188] for client architectures, which coincidentally reduces the risk of widespread exploitation after our submission. However, while it is deprecated on client CPUs, SGX is still available on server CPUs (*i.e.*, 3rd generation of Xeon scalable server CPUs). An attacker only needs one up-to-date system to extract secrets from an enclave (e.g., bypassing Signal private contact discovery [252], leaking DRM secrets or attestation keys). Thus, if not mitigated, exploiting ÆPIC Leak is a significant threat to enclave security. Disabling the APIC I/O memory via a microcode update, or deprecating SGX are effective mitigations against the specific vulnerability discovered. However, we argue that a generic mitigation of the vulnerability class in future hardware is an open research problem we identify.

Contributions. The contributions of this chapter are:

1. We systematically analyze and categorize CPU vulnerabilities, showing that they have the same types as for software, and identifying blank spots.
2. In the blank spots, we discover ÆPIC Leak, an architectural vulnerability in the local APIC leaking data from SGX enclaves, including data in use and data at rest.
3. We design two complementary techniques that leverage microarchitectural optimizations to control which cache line ÆPIC Leak samples from the cache hierarchy.
4. We evaluate our techniques by leaking cryptographic keys, including Intel's official key from the quoting enclave.

Outline. Section 6.2 provides background. Section 6.3 analyzes and categorizes CPU vulnerabilities, leading to blank spots with potentially undiscovered vulnerabilities. Section 6.4 details the ÆPIC Leak vulnerability and its threat model. In Section 6.5, we show how ÆPIC Leak leaks data from SGX enclaves. We discuss mitigations in Section 6.6 and conclude in Section 6.7.

Responsible Disclosure. We responsibly disclosed our findings to Intel on December 8th, 2021. Intel acknowledged our findings on December 22nd, 2021, assigned CVE-2022-21233 and is working on possible mitigations.

Code Access. Our proof of concept of the attacks is open-sourced at https://github.com/IAIK/AEPIC.

6.2 Background

This section covers fundamental background for the reader to understand the rest of the chapter.

6.2.1 APIC

The Advanced Programmable Interrupt Controller (APIC) manages and routes interrupts in modern CPUs. The APIC is split into two different components: The *Local APIC* integrated into each logical core and the external *I/O APIC* in the Intel's System Chip Set. The Local APIC manages interprocessor interrupts (IPIs) and receives interrupts from the processor interrupts pins, forwarding them to the core to be handled, while the I/O APIC receives external interrupts events and forwards them to the target local APICs [182].

Local APIC. A Local APIC can receive, generate, and forward interrupts, both to its local core (through IPIs or local interrupt sources, *i.e.*, timer interrupts, performance monitoring counter interrupts, thermal sensor interrupts), to other cores (through IPIs), and from external devices (through the I/O APIC). Each Local APIC is made up of a set of APIC registers to control its functionality or expose the state of the interrupts in the system. A processor can generate IPIs or set up local interrupts via APIC registers of its own Local APIC.

Local APIC Registers. By default, modern APICs operate in xAPIC mode, which exposes Local APIC registers as a memory-mapped 4 kB region in the physical address space. The address of the region is set in the `IA32_APIC_BASE` MSR and independent for each logical core [182]. At startup, the region is set at physical address `0xFEE00000` but it can be moved on a per-core basis by changing the value of the `IA32_APIC_BASE` MSR. APIC registers are either 32, 64, or 256 bits, but they are mapped into the memory-mapped region as 32-bit values, always aligned to 128-bit boundaries. Thus, registers wider than 32 bits are split and mapped over multiple 128-bit aligned regions in the memory-mapped area. This means that bytes 4 to 15 in each 16-byte (128-bits) region are never architecturally defined. Intel states that *any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed* [182]. The APIC can be set in *x2APIC* mode if supported, which extends xAPIC mode with different improvements, like enhancing the performance of interrupt delivery and providing MSR-based access to APIC registers which disables the memory-mapped interface. The OS can enable or disable x2APIC mode by setting bit 10 of `IA32_APIC_BASE` MSR.

6.2.2 Memory Subsystem

CPUs rely on a hierarchical memory subsystem with data cached over multiple levels. Lower level caches provide faster memory with smaller storage capabilities for data frequently accessed, while higher-level caches offer bigger storage at cost of increased latency. Modern Intel CPUs usually have at the lowest level a private instruction

cache (L1I) and data cache (L1D), and at the second level a private unified cache (L2). The Last-Level Cache (LLC or L3) is usually shared across all physical cores.

Path to Main Memory. The CPU tries to serve each memory access from the lowest cache level possible. It allocates the resources necessary to track the memory requests, *i.e.*, *load* or *store buffers*. Upon completion of the address translation, if any of the physical tags in the indexed cache set matches the physical address, the data is returned from the L1D. In case no tag matches (*i.e.*, the data is not in L1D), the CPU allocates a line-fill-buffer (LFB) entry to interface with the L2 cache. Line fill buffers act as a decoupling component between L1 and L2 caches to keep track of outstanding requests, uncacheable memory accesses, and non-temporal moves [314, 368]. The CPU then performs the lookup in L2, which loads the LFB entry in case the data is present, returns it to L1D, and back to the load buffers. In case data is not present in L2, the CPU must issue an offcore request to the LLC cache. It reserves a *fill buffer* entry to hold the data in the *superqueue* [212, 221] between L2 and LLC, issues the load over the *ring interconnect* [282] and waits for the request to be completed. The superqueue decouples the interaction between the L2 and the LLC caches, in a similar way the line fill buffers do between L1 and L2. The ring interconnect is an on-die interconnect used for uncore communication between CPU cores, LLC, memory controller, and the integrated GPU. The load request is satisfied either by the LLC or the memory controller, and the fill-buffer entry in the superqueue collects the value, which sends the data back to the core.

6.2.3 Intel SGX

Intel Software Guard Extension (SGX) provides a Trusted Execution Environment (TEE) on x86 processors. Introduced in Skylake CPUs, SGX offers hardware isolation and local and remote attestation for so-called *enclaves* even on possibly attacker-controlled machines [182]. SGX enclaves reside in the virtual address space of a userspace process, but their physical memory is backed by the protected Enclave Page Cache (EPC). Stores to EPC are automatically encrypted, and loads are decrypted by the memory encryption engine. While enclave memory is inaccessible to attackers probing the memory bus [86], CPUs affected by transient-execution vulnerabilities can leak the values from the microarchitecture [314, 358, 368].

Enclaves can only be executed from pre-configured entry points using the `eenter` instruction and exit using an `eexit` instruction. If a fault or interrupt occurs while an enclave is running, the processor issues an Asynchronous Enclave Exit (AEX), securely storing and clearing all the enclave CPU registers at the time of enclave interruption in a Save State Area (SSA) inside EPC. An `eresume` instruction restores enclave execution from the SSA frame.

Due to the limited EPC size, untrusted system software can leverage the `ewb` and `eldu` instructions to move encrypted EPC pages to main memory and back, without revealing the content. When an enclave page is moved from main memory back to EPC using `eldu`, it is decrypted and cryptographically verified to ensure its content has not been tampered with, bringing the plaintext data to the L1 cache [358].

SGX supports local and remote attestation. During the enclave creation process, the CPU collects cryptographic measurements about the starting enclave and its

signature in two different Measurement Registers (`MRSIGNER` and `MRENCLAVE`). An enclave can generate a signed local attestation for a target enclave using the `ereport` instruction, which can be cryptographically verified by the target enclave using a key obtained through the `egetkey` instruction. The report of the local attestation includes the enclave's initial code and data as measurement registers in addition to other security-related information [182]. Intel provides a trusted `quoting enclave` to sign locally-generated identity reports using an Intel-private key and enabling remote attestation. The `egetkey` instruction also provides a sealing key that the enclaves can use to securely `seal` secrets for untrusted persistent storage.

SGX enclaves have been compromised in numerous ways over the past years, e.g., memory-safety violations [230], insecure synchronization [384], asynchronous exception management [89], and side channels [261,317,361,362]. SGX has also been the target of transient-execution attacks [299,314,358,368].

6.2.4 Transient-Execution Attacks

On x86, the instruction stream, once fetched, is decoded into smaller micro-operations (μops) to simplify the underlying microarchitecture and enable low-level optimizations. The μops are decoded *in-order* and executed *out of order* over the different execution units, keeping track of the dependencies to satisfy them. The results are committed in order to the architecture, thus ensuring correctness. Given the highly parallel nature of modern CPUs, *branch prediction* has been introduced to avoid stalls, *speculatively* executing the predicted path. If the prediction turns out correct, the speculatively executed μops are committed to the architectural state, while in case of a misprediction, the results are discarded by the CPU. All non-committed μops are discarded if exceptions arise during out-of-order execution. Any discarded μop does not affect the architectural state, but it can affect the microarchitectural state (e.g., cache state). Such instructions are called transient instructions [61,210,236].

6.3 Software and Hardware Vulnerabilities

In this section, we systematically analyze existing documented CPU vulnerabilities on x86 CPUs, showing that the underlying root causes are the same as for software vulnerabilities. We demonstrate that the CWE classification of software vulnerabilities can be applied both to transient-execution vulnerabilities as well as architectural CPU vulnerabilities. Table 6.1 provides this classification.

6.3.1 Types of Vulnerabilities

For a long time, system security relied on the correctness of the underlying hardware, ignoring the possibility of security vulnerabilities on the CPU [202]. With the discovery of transient-execution attacks [210,236], this view has changed drastically. Since the first publication of such attacks, numerous vulnerabilities have been discovered in CPUs [59,61,298,299,314,358,359,368]. However, transient-execution attacks are neither the only nor the first discovered CPU vulnerabilities. The history of CPU vulnerabilities that affect a large number of users goes back to well-known bugs such as the Intel Pentium FDIV bug [75] described in 1995 or the Intel F00F

Table 6.1. Classification of transient and architectural vulnerabilities according to CWE originally targeted at software vulnerabilities. CWE-441 has no architectural counterpart yet. ÆPIC Leak represents the architectural counterpart for CWE-665.

Vulnerability Type		Transient Vulnerability	Architectural Vulnerability
CWE-416	Use-after-free	ZombieLoad [314], RIDL [368], Fallout [59], Spectre-STL [166]	iTLB multihit [184]
CWE-441	Confused Deputy	SWAPGS [245]	-
CWE-119	Out-of-bounds Operation	Spectre-PHT [210], Spectre v1.1 [205], Meltdown-BND [61]	GPU cache-line leak [183]
CWE-843	Type Confusion	Foreshadow-VMM [386]	F00F bug [79]
CWE-682	Incorrect Calculation	LVI-FP [298]	Plundervolt [270], V0LTpwn [201], VoltJockey [296], FDIV bug [75]
CWE-362	Race Condition	Meltdown [236], Foreshadow [358]	AMD Ryzen IRETQ bug [102]
CWE-691	Insufficient CF Management	Spectre-BTB [210], Spectre-RSB [216, 248]	Skylake bug [231]
CWE-74	Improper Neutralization (Injection)	LVI [359]	SEVerity [266]
CWE-665	Improper Initialization	CrossTalk [299], Medusa [264]	ÆPIC Leak (**this chapter**)

bug [79] described in 1997. These bugs did not pose a significant security risk back then. However, today, with cloud computing and trusted-execution environments, such small bugs would be exploitable. DVFS attacks [201, 270, 296] can induce a similar effect as the FDIV bug by causing wrong results in multiplications (instead of divisions), which has been used to break the confidentiality and integrity of Intel SGX. Similarly, LVI-FP [298] induced wrong floating-point calculations in the transient domain, which has been exploited to disclose arbitrary memory in the browser. Hence, as we have seen with software, simple bugs can become exploitable vulnerabilities when exploitation techniques improve [168, 323].

When analyzing existing CPU vulnerabilities, we can–at the high level–categorize them into *architectural* and *transient vulnerabilities*. Architectural vulnerabilities are exploitable by relying only on architecturally-defined interfaces and features. Transient vulnerabilities do not have an architecturally-visible effect as they are only visible on the microarchitectural level and hence require side channels to observe them.

Architectural Vulnerabilities. Architectural vulnerabilities are visible without requiring any further indirection or side effects. x86 CPUs have been affected by several architectural vulnerabilities over the years. Vulnerable components in the architecture may incur invalid states due to design or implementation errors from the manufacturer, causing unwanted behaviours like system hangs, shutdowns, or, in the worst case, undefined states possibly exploitable. For example, the F00F bug was triggered by an invalid opcode that led to the lock-up of the CPU until it was rebooted [79]. The FDIV bug is in this category as well, as it simply provides wrong results for specific operands provided to the floating-point division instruction [75]. Although well-known, these bugs are not the only architectural CPU vulnerabilities. Many architectural bugs were never documented but only mentioned in CPU erratas [183, 184]. The specification update for the 11^{th} generation

of Intel CPUs (released 2020) already contains 73 errata. While many of these errata might not be exploitable, e.g., incorrect values reported or failure to resume correctly from sleep states, the missing details make it impossible to guarantee non-exploitability. Recent vulnerabilities that have been found mostly by researchers [102, 183, 184, 201, 231, 266, 270, 296] are exploitable, though. These vulnerabilities allow an attacker to crash a system [102, 184], leak data from parts of cache lines of a different security domain [183], modify computation results in a different security domain [201, 266, 270, 296], or change the control flow of a different application [231]. Although all these bugs are observable on the architectural level, understanding the root cause is often still difficult [218]. Moreover, while it is often not difficult to trigger the bugs, it is extremely difficult to exploit them in a reliable way that goes beyond a denial-of-service attack [102, 184, 218, 231].

Transient Vulnerabilities. Transient vulnerabilities are not directly visible on the architectural level, as they affect the microarchitecture. Observing these vulnerabilities requires a side channel [61]. Well-known transient vulnerabilities include Spectre [210] and Meltdown [236]. Meltdown-type attacks exploit delayed exception handling in out-of-order execution, while Spectre-type attacks leverage branch mispredictions. To leak data from the transient domain, the secret data is encoded into microarchitectural elements not cleared upon discarding transient instructions and transferred to the architectural state via a covert channel [35, 163, 234, 315, 383, 394]. As these vulnerabilities require indirect observation, they are much harder to detect accidentally. Similar to architectural vulnerabilities, many of them might not be exploitable [61]. However, as with architectural vulnerabilities, several of these vulnerabilities have been exploited successfully [35, 59, 166, 205, 210, 216, 236, 248, 298, 299, 314, 358, 359, 368, 386]. These vulnerabilities allow an attacker to read architecturally inaccessible data from the own process [166, 210], change the transient control flow of processes [205, 210, 216, 248], inject data into the transient domain [298, 359], and leak data from different security domains [59, 236, 299, 314, 358, 368, 386]. The last generation of Intel CPUs (Sunny-Cove-based CPUs) is not vulnerable to Meltdown-type attacks due to in-silicon mitigations.

6.3.2 Classification of Vulnerabilities

For software (and now also hardware) vulnerabilities, there is the CWE (Common Weakness Enumeration) classification. This classification contains more than 900 categories of vulnerabilities [351]. Intuitively, one would assume that the hardware vulnerabilities cover CPU vulnerabilities. However, while they are indeed classified in the hardware-bug categories in the CWE, sometimes even with their own categories, we argue that these categories are not necessary to enumerate the vulnerabilities. Looking at modern CPUs, they are designed using hardware-description languages (HDLs) [352]. Hence, CPUs can, to some extent, also be considered as *software*.

Our analysis shows that the underlying root causes of CPU vulnerabilities are not so different from (complex) software vulnerabilities. Thus, they can be classified using the existing software-vulnerability categories (cf. Table 6.1). This classification works both for architectural and transient vulnerabilities.

Out-of-bounds Operation (CWE-119). The transient-execution attack Spectre-PHT [61, 210] can be classified under CWE-119 "Improper Restriction of Operations within the Bounds of a Memory Buffer". The description of this category states: "The software performs operations on a memory buffer, but it can read from or write to a memory location outside of the intended boundary of the buffer." [351], which is precisely what is happening in Spectre-PHT, except that it is not the software but the CPU. Meltdown-BND [61] exploits a similar problem, where the hardware transiently ignores the bounds check for a buffer. Although there are not many details, the architectural vulnerability Intel SA-00219 [183] also fits into this category. On affected CPUs, the integrated graphics card has an incorrect bounds check that allows reading the first 64 bit of a cache line used inside SGX enclaves.

Use after Free (CWE-416). According to Schwarz et al. [314], the root cause of the transient-execution attacks known as microarchitectural data sampling (MDS) [59, 314, 368] is a use-after-free vulnerability in internal CPU buffers. The old content of these internal buffers, *i.e.*, the line-fill buffer and the store buffer, is used transiently in a faulting load, although the entry was already free'd by a previously finished load (or store). Similarly, in Spectre-STL [166], the CPU uses old stale memory locations that should have already been overwritten by newer stores, *i.e.*, it reads from a resource that was already "released". The iTLB multihit vulnerability [184] is an architectural instance of a use-after-free vulnerability. In this vulnerability, the CPU tries to use an old TLB entry that is not valid anymore, while a newer valid TLB entry already exists for the virtual address. Hence, although the old entry should have been released by creating the new entry, the CPU still tries to use the released one, leading to a CPU lockup [184].

Confused Deputy (CWE-441). A confused deputy vulnerability sees an intermediary forwarding a request to a target resource without preserving information about access permissions of the origin source. When the SWAPGS instruction is speculatively executed in kernel mode, it swaps the kernel GS register with the user GS register during the transient window. The transient swap causes the CPU to use user-provided values in the GS register [245]. The SWAPGS instruction acts as a confused deputy to the instructions dereferencing GS, leaving no trace of the origin of the GS value that was coming from userspace and not kernelspace. We did not identify any corresponding architectural vulnerability in this category.

Type Confusion (CWE-843). In the Foreshadow-VMM [386] variant of Foreshadow [358], the CPU suffers from a type confusion in the page-table entry of a guest page table. On a non-present fault inside the VM, the CPU treats the page-table entry like a host page table, interpreting the stored page frame number as a host physical address instead of a guest physical address. The F00F bug [79] can also be considered as a type confusion: the CPU locked the bus as it confused the register access of the opcode with a memory access, preventing the bus lock from being released as the CPU did not observe the completed memory access.

Incorrect Calculation (CWE-682). The LVI-FP vulnerability [298] shows that the transient result of floating-point values can be modified in certain corner cases where the operation requires a microcode assist. While the calculation is corrected architecturally, subsequent code that is executed transiently works with incorrect values. The FDIV bug is the famous example of an architectural incorrect calculation, where the result of floating-point divisions was incorrect for specific operands [75].

Race Condition (CWE-362). The first Meltdown-type attacks Meltdown-US [236] and Foreshadow [358] can be considered race conditions. In both cases, the data is already accessed and forwarded to dependent operations before the CPU realizes that the virtual address points to architecturally inaccessible data. While there are not many details available about the AMD Ryzen IRETQ bug [102], it is very likely a race condition, as it can only be triggered when executing the `iretq` instruction on one hyperthread, while running a CPU-bound loop on the other hyperthread [102]. In this setup, the hyperthread executing the `iretq` stalls until the sibling hyperthread pauses.

Insufficient Control-Flow Management (CWE-691). For both Spectre-BTB [210] and Spectre-RSB [216, 248], an attacker can change the transient control flow unexpectedly. As the CPU does not properly distinguish between different applications for branch-prediction targets, an attacker can inject an arbitrary branch target. On Intel Skylake CPUs, there is an architectural vulnerability that is not well understood but has similar effects [231]. Using 8-bit registers in a tight loop on one hyperthread can lead to unexpected changes of the instruction pointer on the sibling hyperthread.

Improper Neutralization (*Injection*) (CWE-74). LVI [359] injects values into a victim's transient data stream. In these attacks, the CPU does not properly neutralize the input to a faulting (or assisting) load, forwarding unrelated attacker-controlled data, *i.e.*, dependent operations receive incorrect data. This matches the description of CWE-74: "The software constructs all or part of a command, data structure, or record using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify how it is parsed or interpreted when it is sent to a downstream component." [351]. On AMD, there is an architectural vulnerability in this category called SEVerity [266]. Due to missing integrity checks of encrypted memory, an attacker can inject code into SEV-protected VMs.

Improper Initialization (CWE-665). The CrossTalk [299] transient-execution attack exploits the improper initialization of the internal staging buffer of the CPU. This buffer is used for the hardware random-number generator, as well as for the `cpuid` instruction. In both cases, only a part of the buffer is used, and the remaining part of the buffer is not cleared. However, the entire buffer is transmitted to the line-fill buffer, from where the improperly-initialized buffer can be leaked via RIDL [368] or ZombieLoad [314]. In this chapter, we show the first architectural vulnerability in this category. We show that the reserved part of the APIC registers on Ice Lake and

Alder Lake CPUs are not properly initialized, leaking stale data that was loaded from or stored to the LLC cache.

6.3.3 Missing Architectural Counterpart Discovery

Except for CWE-665 (Improper Initialization) and CWE-441 (Confused Deputy), we identified both transient and architectural vulnerabilities in every category in Table 6.1. We target the blank spot in CWE-665 by systematically analyzing the possible targets for architectural vulnerabilities caused by improper initialization. We focus on *data loads* where the underlying data structure is larger than the loaded data. For this, we focus on the I/O address space. As data leakage from valid memory addresses would have already been discovered, we do not expect any architectural vulnerabilities there. Similarly, previous work investigated the address space of model-specific registers [108] without discovering any data leakage.

In our experimental setup, we iterate over the entire I/O address space by mapping the address space page-by-page into the user space. Similarly to the approach described by Moghimi et al. [264], we groom microarchitectural buffers on the hyperthread while reading from the I/O address space. The grooming application simply reads and writes known data, ensuring that they end up in the store buffer, fill buffers, and cache hierarchy. If a value read from the I/O address space matches the known data, the physical address is reported as a potential source of data leakage.

Such a scan takes around 3 h to 4 h depending on the system we tested. On all Ice Lake and Alder Lake CPUs, this scan reported a physical address that architecturally leaks data from the sibling hyperthread: `0xFEE00000`. In Section 6.4, we provide an analysis of this architectural information leakage, showing that it is indeed caused by improper initialization. The scanning also led to several crashes, e.g., when reading from the Serial IO GPIO host controller. As scanning is only possible from ring 0, *i.e.*, the kernel, we do not consider this behavior security-relevant. While reading from address `0xFEE00000` is also only possible from the kernel, such an attacker is valid when attacking SGX enclaves.

6.4 ÆPIC Leak Overview

In this section, we introduce ÆPIC Leak, an architectural vulnerability in Intel CPUs that exploits undefined behavior in the APIC to leak data from the cache hierarchy. We provide an overview of ÆPIC Leak in Section 6.4.1, its threat model in Section 6.4.2 and analyze the root cause in Section 6.4.3. Based on the analysis, we introduce required building blocks for exploitation in Section 6.4.4.

6.4.1 Attack Overview

Figure 6.1 shows a high-level overview of ÆPIC Leak. ÆPIC Leak leaks values by *architecturally* reading the undefined range of APIC registers from ring 0, *i.e.*, the OS. Accessing bytes 4 to 15 of each 16 −byte register results in undefined behavior according to Intel [182]. This undefined behaviour includes reading either zeros or *0xFF*, system hangs, or triple faults on most CPUs. However, as discovered via the I/O address-space scan (cf. Section 6.3.3), this is not the case on Sunny-Cove-based

Table 6.2. Subset of tested CPUs and whether they are vulnerable (✓) or not (✗) to ÆPIC Leak. All tested Sunny-Cove-based CPUs are vulnerable.

CPU	Microarchitecture	Based on	ÆPIC Leak
Core i3-1005G1	Ice Lake	Sunny Cove	✓
Core i5-1035G1	Ice Lake	Sunny Cove	✓
Core i7-10510U	Comet Lake	Skylake	✗
Core i5-1135G7	Tiger Lake	Willow Cove	✗
Core i9-12900K	Alder Lake	Sunny Cove	✓
Xeon Platinum 8375C	Ice Lake SP	Sunny Cove	✓

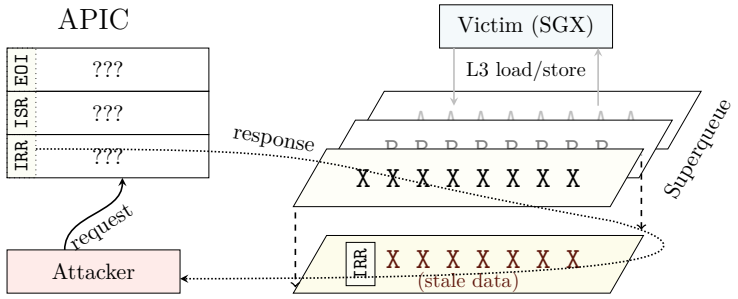


Figure 6.1. ÆPIC Leak reads a reserved part of an APIC register. The APIC uses the superqueue between L2 and LLC to transfer the data to the core. The reserved parts do not overwrite the superqueue entry, exposing stale values from previous reads and writes of other applications to the attacker.

CPUs. Instead, stale data from the superqueue is returned. Accessing any defined or undefined register in the byte-range 4-15, with a load width between 1 and 4 bytes, returns such stale data. Hence, ÆPIC Leak can atomically leak a 32-bit value per read. Load widths of 8 bytes or more return *0xFF*, and thus do not leak data.

The uninitialized data returned from ÆPIC Leak is not restricted to any security domain, *i.e.*, the origin can be user-space applications, the kernel, and, most importantly, SGX enclaves. Our hypothesis is that the invalid offsets in APIC registers are not properly initialized, *i.e.*, zeroed. Our experiments indicate that the superqueue is used as a temporary buffer for APIC requests. The superqueue entry contains stale data of recent memory loads and stores that traveled from the L2 to the L3 or the other direction. The APIC only overwrites the architecturally-defined parts of the register and leaves the stale values in the reserved part.

There is no correlation between the APIC register used for leaking data and the leaked data. Reading any reserved address within the APIC range `0xFEE00000`-`0xFEE003FF` leads to the same leakage. The only control over the leaked data is the cache-line offset. The cache-line offset of the used APIC address always matches the cache-line offset of the leaked data, which is also the case for MDS attacks [59, 314, 368].

As valid APIC register parts overwrite the stale value, the leakage pattern is as illustrated in Figure 6.2. For every 16 B block, the first 4 B contain valid APIC data,

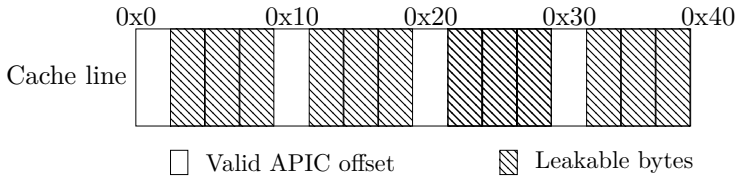


Figure 6.2. Leakable bytes in a 64-byte cache line.

followed by 12 B stale data. Hence, ÆPIC Leak deterministically leaks 48 B from a cache line. Another limitation is that ÆPIC Leak only leaks even cache lines, *i.e.*, cache lines that start at an address that is a multiple of 128. As cache line pairs are typically transferred in pairs [181], we hypothesize that the second cache line is transferred first and then immediately overwritten by the first cache line, leaving only the stale data of the first cache line in the superqueue. Still, the leakage of ÆPIC Leak covers 37.5 % of any page. Section 6.5 shows that this is sufficient to, e.g., extract AES-NI keys from SGX enclaves, and presents different techniques to circumvent this limitation. However, this is exactly what we propose to leverage, to mitigate ÆPIC Leak at the software level (cf. Section 6.6.3).

6.4.2 Threat Model

Following most microarchitectural attacks on Intel SGX, we assume the attacker can execute privileged native code on the target machine. At the hardware level, we assume a Sunny-Cove-based Intel CPU (e.g., 10^{th} and 12^{th} generation code name "Ice Lake" and "Alder Lake" and 3^{rd} generation Xeon scalable "Ice Lake SP"). These CPUs are not vulnerable to any Meltdown-type attacks, such as Meltdown [236], Foreshadow [358, 386], RIDL [368], or ZombieLoad [314]. ÆPIC Leak observes memory operations inside an Intel SGX enclave. We assume either a malicious hypervisor targeting secrets in guest enclaves or a privileged attacker willing to extract secrets from local enclaves, e.g., bypassing private contact discovery on Signal Servers [252], leaking DRM secrets or even SGX attestation keys. ÆPIC Leak only requires the OS or hypervisor to access the physical Local APIC to leak secrets, with no difference between the two settings. The attacker is either running on the same physical core, either on the sibling logical core or on the same logical core, e.g., if hyperthreading is disabled. While SGX enclaves can detect if hyperthreading is enabled during remote attestation [177], there is no recommendation to disable hyperthreading on CPUs with silicon fixes against Meltdown-type attacks. Thus, on Sunny-Cove-based CPUs, hyperthreading can be enabled. Still, even without hyperthreading, ÆPIC Leak can leak memory operations inside an SGX enclave, just with a reduced leakage rate.

In line with the SGX threat model, an attacker can rely on arbitrary operating-system features, such as the modification of page-table entries [362], the precise interrupts of enclaves using timer interrupts [360], or the execution of privileged SGX instructions, such as `EWB` to evict EPC pages.

Virtualized Environments. A malicious virtual machine with access to the host Local APIC could exploit ÆPIC Leak to observe data from other tenants or

the hypervisor. However, no hypervisor we analysed exposes direct access to the host Local APIC. Usually, the APIC MMIO region, when enabled, is emulated by the hypervisor by intercepting the accesses to the region and managing the virtual interrupts [350]. In case Intel APIC virtualization (Intel APICv [182]) is enabled, the physical CPU emulates APIC functionality for the virtual CPUs in dedicated pages. We empirically verified that ÆPIC Leak does not work with APIC virtualization and APICv mode to leak from a guest VM. Thus, ÆPIC Leak does not allow guest virtualized systems to leak data. On the contrary, a malicious hypervisor could leverage ÆPIC Leak to leak secrets from guest VMs, leveraging its own Local APIC, irrespective of the guest APIC configuration.

Other Vendors and CPUs. We tested all Intel Core microarchitectures from Sandy Bridge (2^{nd} generation) to Alder Lake (12^{th} generation), and AMD CPUs from Zen to Zen 3. We did not discover any vulnerable CPU other than the ones based on Sunny Cove. Table 6.2 reports a subset of the CPU we tested, see Appendix G.3 for the full list. We observe hangs or reads of *0x00* or *0xFF* on unaffected CPUs.

6.4.3 Leakage Analysis

In this section, we analyze the leakage of ÆPIC Leak, *i.e.*, from which microarchitectural element the data originates. We designed several experiments that show how the leakage source of ÆPIC Leak is different from previous microarchitectural attacks and demonstrate that ÆPIC Leak allows picking up stale values from the superqueue. We performed our tests on an Ice Lake Core i5-1035G1 machine, with Ubuntu 20.04.1, kernel 5.4.0-96, and the last microcode update installed (cf. Table 6.2).

6.4.3.1 Ruling out Microarchitectural Elements.

As we cannot directly observe from which microarchitectural element ÆPIC Leak leaks, we instead rule out microarchitectural elements from which ÆPIC Leak does not leak, expanding and systematizing the methodology from Schwarz et al. [314]. Our methodology is general to be applied to the study of the leakage of other CPU bugs. In this section, we describe the experiments we designed for all microarchitectural elements that are not involved, *i.e.*, where ÆPIC Leak still leaks the targeted data after clearing or circumventing them.

L1 Data Cache. By flushing the L1D via `MSR_0x10B` [177] and disabling hyperthreading, we ensure that the targeted data is not stored in the L1 while being leaked.

Line-Fill Buffer and Load Ports. We use the software sequences provided by Intel [192] to clear intermediate buffers, including the LFB and load ports, and still leak values.

L1 Instruction Cache. ÆPIC Leak leaks code and data, which travel through different paths in the hardware (e.g., code does not go through the LFB). It is

unlikely that both paths (L1D and L1I) are affected and we see some combined leakage. Thus, we rule out the L1 cache and its line fill buffer.

Store Buffer. ÆPIC Leak is not limited to store operations but also leaks memory loads. Thus, we can eliminate the store buffer as leakage source. Moreover, ÆPIC Leak cannot leak *transient* stores which are only stored in the store buffer [59].

L2 Cache. ÆPIC Leak cannot leak data that is kept in L2 and not evicted towards L3. Thus we can exclude stale data in L2 as the source of leakage.

L3 Cache. ÆPIC Leak does not leak data kept in L3 while being exclusively used by other cores, and thus, not loaded towards the local L2 cache. This rules out all CPU caches.

Ring Bus. ÆPIC Leak cannot leak values processed by the GPU or from LLC slices exclusively used by other physical cores, also ruling out the ring bus. Moreover, ÆPIC Leak also works on Xeon CPUs without a ring bus [127].

Staging Buffer. ÆPIC Leak does not leak values from `cpuid` or `rdrand`, ruling out the staging buffer.

Memory. We also rule out the *DRAM* and *memory controller* by marking a memory region as *uncachable* to ensure that every store and load circumvents the cache hierarchy. ÆPIC Leak does not leak these loads and stores.

System Agent. As ÆPIC Leak does not leak values from *PCI* devices, we exclude this subsystem as leakage source.

Our experiments rule out the known internal buffers up to the L2 cache and the components in the uncore subsystem. Thus, we hypothesize that the leakage source is the internal buffer between the L2 and LLC cache, *i.e.*, the **superqueue**. We achieve the best leakage when building eviction sets that evict data from L2 but not from L3, and when relying on cache-line bouncing [256]. In both cases, the data deterministically moves through the superqueue between L2 and L3.

6.4.3.2 Performance Counter Analysis.

ÆPIC Leak does not trigger an architectural fault when performing a load instruction, even on reserved and undefined offsets of the MMIO region. However, we observed subtle microarchitectural differences when performing a load from a 16 B-aligned offset, whether it contains defined data or has been reserved by the specification. For every load to a reserved or undefined region, we observe a higher latency (437 cycles ($n = 1000$, $\sigma_{\bar{x}}$=0.57)) in contrast to a valid offset (47 cycles ($n = 1000$, $\sigma_{\bar{x}}$=0.03)). The `MACHINE_CLEARS.COUNT` performance counter indicates that the invalid load triggers an exception not forwarded to the architectural level. Furthermore, `OFFCORE_REQUESTS_OUTSTANDING.X` performance counters indicate that the core sends offcore requests as the loads are not satisfied by the local

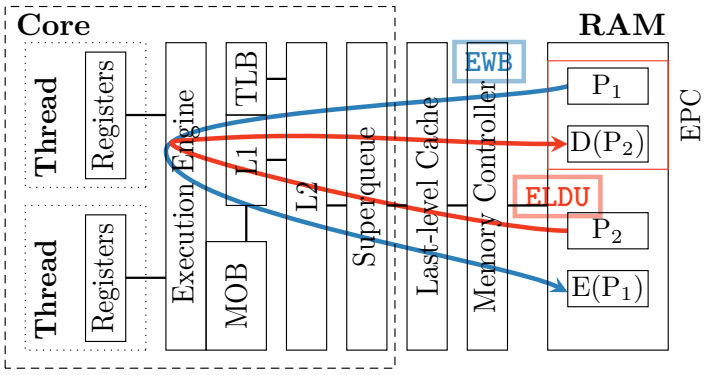


Figure 6.3. When executing the `ewb` instruction, a transparently encrypted page gets re-encrypted and moved to the non-EPC main memory. During this process, the unencrypted content of the page *flows* through the cache hierarchy. Vice versa for decryption with the `eldu` instruction.

APIC, increasing the `CYCLE_ACTIVITY.STALLS_X` and, hence, the observed access time. As the data is not in any cache (and the PTE marked uncachable), the miss creates an entry in the superqueue and allocates a line fill buffer [232]. Table G.2 in Appendix G.3 gives an overview of all performance counters that show differences for defined and undefined offsets.

6.4.4 Building Blocks

Reading from a reserved part of the APIC does not provide any control over which cache line is leaked. However, we introduce three building blocks to influence which cache line is leaked. First, we force the target lines into the superqueue. Second, we increase the leakage of the specific target line. Finally, we extract the target line from the noisy measurements.

Forcing Data into the Superqueue. To target specific data, ÆPIC Leak first forces target cache lines into the superqueue. Van Bulck et al. [358] demonstrated that `ewb` and `eldu` swapping instructions bring plaintext data into the L1 cache while moving EPC pages. As the EPC pages are copied from memory to the L1 cache, and back from the L1 cache to memory, the–at this point unencrypted–data also travels through the superqueue. Hence, an attacker can bring data from an arbitrary enclave page into the superqueue using these instructions (cf. Figure 6.3). ÆPIC Leak uses a modified version of the Linux SGX driver to identify the enclave coupled with a target process and continuously swaps the target pages of the enclave. There is no need for ÆPIC Leak to run in the same or parallel hyperthread of the victim process, as the EPC swapping mechanism works independently from the enclave owning the page. Furthermore, as the EPC memory is persistent during the enclave's existence, ÆPIC Leak does not require the enclave to be active, *i.e.*, executing ECALLs during the leak. We refer to this technique as Enclave Shaking. Direct integration into the driver allows targeting an arbitrary enclave on the system, including Intel's quoting enclave.

Increasing the Leakage. While Enclave Shaking forces data from an EPC page into the superqueue, there is no control over which cache line is leaked. To solve that problem, we introduce Cache Line Freezing, a novel technique that provides control over which cache line of the page is leaked. Cache Line Freezing exploits that the cache access pattern of the hyperthread running in parallel to the attacker influences which entry is used, and thus leaked, from the superqueue.

Counterintuitively, Cache Line Freezing continuously accesses a page offset x of unrelated pages to *increase* the probability of leaking the cache line at the same offset x in the enclave. Specifically, the parallel hyperthread evicts the cache set of the target cache line with a crafted eviction set made up by continuously accessing several contiguous zero-filled pages at the same page offset of the target line. In our setup, it is sufficient to keep iterating over 256 virtually-contiguous pages at offset x, to trigger the effect.

The access pattern ensures that zero-filled cache lines with the same offset as the target cache line are continuously evicted from the L2 cache, and thus interact with the superqueue. Note that Cache Line Freezing does not work if the cache lines of the eviction set are set to values different from '0', or if their pages are all mapped to the kernel zero-page, *i.e.*, have never been written.

While the exact interaction of Cache Line Freezing with the superqueue is unknown, we hypothesize that zero-filled loads and stores are optimized. This would be in line with the observation that Ice Lake can eliminate stores of '0's to cache lines that only contain '0's [112]. Furthermore, we observe that if only one half of the cache lines of the eviction set is filled with zeros, we can only leak that same half of the target line with this method. This indicates that the granularity of this zero optimization is 32 B, in line with the memory bus width [181]. We assume that the zero data is marked differently in the superqueue, *i.e.*, only in the metadata without overwriting the entry. As a result, the eviction set keeps the entry in the superqueue used without overwriting it. Thus, stale data is preserved over a longer duration, increasing the probability to leak it. Notice that Cache Line Freezing is the only building block of ÆPIC Leak where enabling hyperthreading has an impact. When hyperthreading is disabled, Cache Line Freezing must interleave with the attacker process on the same logical core, reducing its efficacy.

Cache Line Freezing allows an attacker to precisely select which cache line numbers to leak from the victim, and thus, to control which line is sampled. In case Cache Line Freezing is not leveraged, ÆPIC Leak would simply degrade to a sampling-based attack, similarly to MDS attacks, and additional techniques might be needed to reconstruct the original order of the leaked data [314, 368].

Extracting the Target Line. In addition to the content of the target cache line of the targeted EPC page ($\mathrm{CL}_{\mathrm{target}}$), unrelated values are leaked. Unrelated values include code and data involved in swapping pages and leaking values as well as data from general system activity. Since ÆPIC Leak is an architectural bug that deterministically reads stale data in the superqueue, the only noise it incurs is leaking such unrelated values. To filter these values, we establish a noise profile by leaking the content of the cache line with the same cache-line index from a different EPC page ($\mathrm{CL}_{\mathrm{noise}}$). Based on this noise profile, we can remove all values that have

a similar frequency for CL_{noise} and CL_{target}. These values are likely independent from the content of the EPC page. Infrequent values that only occur for CL_{noise} or CL_{target} are likely secret-independent values from other applications or the OS and can thus also be ignored. The remaining values observed for CL_{target} are sorted by frequency. The value occurring with the highest frequency is likely the actual value of CL_{target}.

Due to the zero optimization (cf. previous paragraph), ÆPIC Leak cannot directly leak zero-filled blocks, as they are not stored in the superqueue. Instead, ÆPIC Leak can infer that a cache line contains zeros if there is not a single value with a distinct frequency, *i.e.*, the two most-frequent values have a similar frequency.

6.4.5 Performance Evaluation

To evaluate ÆPIC Leak's leaking characteristics, we set up a debug enclave that generates secret data via the `rdrand` instruction. This data is generated during an initial ECALL, and the page is targeted with ÆPIC Leak. To verify the correctness of the leaked data, we use the `edbrg` to read the generated page from the debug enclave *after* the leakage to ensure no other source is contributing to the leakage.

Repeating each cache line leak 2000 times, we achieve a leakage rate of 334.8 B/s with an average error rate of 7.8 % ($n = 100$, $\sigma = 2.4\,\%$). Decreasing the number of repetitions to 200 , the leakage rate increases to 1.76 kB/s with an average error rate of 16.0 % ($n = 100$, $\sigma = 4.1\,\%$) due to the increased noise of unrelated values. Note that in contrast to transient-execution attacks, all leaked values are correct. Noise only refers to data of other applications. Due to ÆPIC Leak limitations (cf. Section 6.4.1), this approach leaks 37.5 % of a page. This percentage can be further extended by combining different exploitations techniques, as we show in Section 6.5.

6.5 ÆPIC Leak Exploitation

In this section, we describe three attacks leveraging ÆPIC Leak against SGX enclaves. While in theory, ÆPIC Leak could leak memory from VMs or the hypervisor, no major hypervisor maps the host APIC MMIO region in the guest. We evaluate our attacks on the Ice Lake Core i5-1035G1.

6.5.1 Attack Techniques

We describe two attack techniques with ÆPIC Leak. We either target the data section of an enclave to leak secret *data at rest*, or we target the SSA area to leak *data in use* in the registers. Due to the limitations of which cache-line parts ÆPIC Leak can leak (cf. Section 6.4.1), the most effective technique to leak the target secret depends on the victim application. We observe no difference while leaking data from debug, pre-release or release enclaves.

Leaking Data and Code Pages. The straightforward use case for ÆPIC Leak is to combine Enclave Shaking and Cache Line Freezing to leak the data (and code) at rest of an SGX enclave. With Enclave Shaking and Cache Line Freezing, we target every cache line of a target page to leak 48 B of each even cache line within

Table 6.3. Leakable SSA registers. For underlined GP-registers (e.g., rdi) ÆPIC Leak can only leak the upper 32-bit as the lower 32-bit are overshadowed by valid APIC registers.

Class	Registers
General Purpose	rdi r8 r9 r10 r11 r12 r13 r14
SIMD	xmm0-1 xmm6-9

the page. This results in an overall leakage rate of 37.5 % of the page content. We repeat this process for each enclave page to recover a memory dump of an enclave. This technique is usable while the enclave is not running, resulting in a consistent state of the enclave data.

Leaking Register Values. Although ÆPIC Leak only leaks values from the superqueue, we can also use it to leak register values. During an asynchronous event, e.g., an interrupt, the hardware stores the current enclave registers in the SSA. Hence, the current register values are stored in the EPC. From there, we can again use Enclave Shaking and Cache Line Freezing to target a specific cache line containing one of the enclave registers and partially reconstruct the value of this register. Furthermore, by combining ÆPIC Leak with SGX-step [360], we can precisely single step the enclave, interrupting the enclave after each instruction. Hence, leaking the partial register state is possible after each executed instruction. As ÆPIC Leak does not require the enclave to run, we can target the SSA page with no timing restrictions, potentially recovering a full register trace of the enclave. However, due to the leakage limitations, ÆPIC Leak is restricted to the registers specified in Table 6.3.

Based on the register leakage, we identify a generic technique to leak data copied inside enclaves: the `__memcpy` function uses the `rdi` register as temporary storage to move data from the source over `rdi` to the destination. Since ÆPIC Leak can leak the upper 32 bit of the `rdi` register, this allows leaking 50 % of any data copied with `__memcpy` inside enclaves.

6.5.2 Breaking AES-NI

Our first attack targets the 128-bit key in the constant time AES encryption provided by the Intel IPP library [180]. The IPP library leverages AES-NI for cryptographic primitives. The AES-NI primitives are tightly entangled with the enclave execution to, e.g., unseal and seal data or transfer data outside the enclave. We use the provided AES example from the official IPP GitHub repository [190]. The example uses the `ippsAESInit` function to initialize the AES context and the `ippsAESDecryptCTR` function to decrypt data with the AES counter mode. Leaking the key is possible either if it is at rest in the data page, or if it is in use in a register.

Key on Data Page. We can dump all the enclave pages after the secret key is transferred to the enclave and resides in memory. If the attacker knows the memory

offset where the key is stored, this offset can be targeted directly. Given that there is no ASLR in enclaves [319], and the code is typically not confidential [86], this is a realistic assumption. Depending on the enclave memory layout, this technique has an ad-hoc probability of 50 % to leak the key: if it is stored in an even cache line, extracting the key is possible, if it is stored at an odd cache line it cannot be leaked. In the latter case, an attacker can leak the key when it is in use.

Key in SIMD Register. We assume that the IPP primitives used in an enclave are usually not modified by an enclave developer. Therefore, we can find the functions leaking the key without analyzing the remaining enclave. Furthermore, we assume that the enclave code is not encrypted. For encrypted code, we could first either leak the decryption key, or simply the decrypted code.

We developed an sgx-gdb [176] script that traces a debug version of the target enclave. This script prints the content of all leakable registers listed in Table 6.3, which are stored in the SSA. We identified that the `k0_aes_DecKeyExpansion_NI` function, which is independent of the AES implementation, temporarily stores the AES key in the `xmm1` register. Hence, by interrupting the enclave during that function, ÆPIC Leak can leak 96 bit of the AES key from the SSA. We can recover the remaining 32 bit of the key in the `k0_aes128_KeyExpansion_NI` function. Furthermore, we also leak 96 bit of the initial value over the `xmm0` register in the `k0_EncryptStreamCTR32_AES_NI` function. The remaining 32 bit can also be easily bruteforced, as it is exactly known which bits are missing. On the i9-12900K, we can evaluate on average 403 million AES keys per second. Hence, in the worst case, it takes 10.7 s to bruteforce the missing bits.

Evaluation. We evaluate ÆPIC Leak with 100 different random keys and try to leak the AES keys with a single run of the attack. A full key recovery takes on average 1.35 s ($n = 100$, $\sigma = 15.70\,\%$) with a success rate of 94 %. In the remaining 6 cases, we leaked unrelated data from different applications. However, as an attacker can typically restart enclaves arbitrarily often, as it is the case with the Quoting Enclave, the attack can simply be repeated until the correct key is leaked.

6.5.3 Breaking RSA

To show that ÆPIC Leak is not limited to secrets in single registers, we target RSA keys from the IPP library reference example [190]. The enclave contains the secret primes P and Q as well as the private key parts dQ, dP and $qInv$. The public modulus N is computed in `ippsRSA_SetPrivateKeyType2` when initializing the RSA context.

Key on Data Page. Similar to AES-NI, we can target the memory used to store the RSA key parts. RSA keys are usually not stored directly in registers and are larger than 128 bit. Therefore, dumping the enclave data pages already has a high chance to leak parts of the stored RSA key.

Key in GP Registers. We can leak the RSA primes P and Q during the calculation of the public modulus N. The bits of the prime numbers P and Q

temporarily flow through `r10` in the function `k0_cpDec_BNU`, and dP and dQ in the function `k0_ippsRSA_SetPrivateKeyType2`, and thus can be leaked.

Evaluation. We target RSA-1024 and leak 100 random 512 bit RSA primes with ÆPIC Leak. Leaking one of the secret parameters is already sufficient to fully recover all the remaining parameters and decrypt data. We count the attack on RSA as successful if we can fully recover at least one of the four RSA parameters. Leaking the parameters from registers has a success rate of 72 %. The attack takes on average 81.81 s ($n = 100$, $\sigma = 48.92\,\%$). In 18 cases, the parameters are not leaked as single-stepping the target instruction fails. In 10 cases, we leak data from other processes. However, the attack can typically be repeated until the correct key is leaked.

6.5.4 Breaking SGX Attestation

As previous work [358, 367, 369], we demonstrate leakage of sealing keys. With the sealing keys, it is possible to unseal sealed data as well as to decrypt the attestation keys, the fundamental security primitives used in SGX. The derivation process to get access to such a key is done in hardware with the `egetkey` instruction [86]. We can target the results of this instruction within the SGX implementation `sgx_unseal_data`. This function uses the generated `egetkey` key to derive the AES round keys used to unseal the encrypted data.

To test the attack, we build and debug an enclave that uses the `sgx_seal_data` and `sgx_unseal _data` functions to seal and unseal enclave data. By tracing the occurences of the `enclu` instruction with `rax=1` we can precisly target the `egetkey` instruction. By following the hardcoded addresses to this instruction, we can find the `sgx_get_key` function without additional debug information. We use the offset of the `sgx_get_key` function to monitor its accesses and start ÆPIC Leak after observing the first access within our target enclave. From this point, we partially leak the `xmm0` and `xmm1` registers with Enclave Shaking and Cache Line Freezing and attack the AES key expansion as demonstrated in Section 6.5.2. We decrypt the sealed data passed to the untrusted environment with the extracted sealing key.

Extracting the EPID Private Key. We attack the official Intel quoting enclave [386] by modifying the untrusted sgx-psw aesmd service. The service handles the inter-process communication between the various Intel enclaves. In the modified service, we target the first call to the `verify_blob` function, which passes the encrypted EPID private key blob, retrieved from the provisioning enclave to the quoting enclave. During this ECALL, we use ÆPIC Leak to extract the blob's sealing key as described above. We use the extracted key together with the known zeroed initial value to decrypt the blob with `sgx_rijndael128GCM_decrypt`, and successfully verified the tag: as the GCM decryption is authenticated, this proves that the key is correct. Extracting the EPID keys allows an attacker to forge remote attestations, breaking the whole SGX system, as enclaves can then be emulated. Thus, SGX could not be trusted anymore on any platform until the keys are replaced. In addition, such an attack may also break TDX confidentiality, which bases its attestation on SGX [187].

6.6 Mitigations

In this section, we discuss mitigations in hardware (Section 6.6.1), firmware (Section 6.6.2), and software (Section 6.6.3).

6.6.1 Hardware

As a long-term solution, ÆPIC Leak has to be fixed in hardware. Given that older Intel CPU microarchitectures are not affected by ÆPIC Leak, we assume that fixing the issue in silicon is not complex. Similar to uninitialized variables in software, it might be sufficient to set the most-significant 12 B of any APIC to a defined value, such as '0' or '-1'. When accessing the APIC using 64 bit reads, the return for all reads is already '-1', regardless of whether the address points to a valid or reserved part of any APIC register. Hence, such functionality to return properly-initialized data already exists.

6.6.2 Firmware

In addition to hardware, mitigations can also be deployed on the firmware level, *i.e.*, as microcode update. Based on mitigations for other CPU vulnerabilities [192], we suspect that mitigations on the firmware level are the most promising mid-term solutions until the hardware is fixed. Intel can deploy such firmware fixes as microcode updates distributed and applied by the OS. As the microcode security version number is part of the SGX attestation [86], enclaves can refuse to run if the microcode updates are not applied. We propose three approaches to mitigate ÆPIC Leak name in microcode with different advantages and disadvantages.

Disable SGX. Even if disabling SGX is not a real mitigation, ÆPIC Leak only targets SGX enclaves, thus, microcode can simply disable SGX. Without SGX, there is no target within the threat model of ÆPIC Leak. Coincidentally, Intel deprecated SGX on Ice Lake and Alder Lake client CPUs [188]. However, ÆPIC Leak is still relevant, as Intel did not deprecate SGX on server CPUs (e.g., Ice Lake SP). Thus, while exploiting an enclave with ÆPIC Leak would not be possible on client CPUs (preventing widespread exploitation), it is still possible by using server CPUs. Leaking the Intel keys on a single up-to-date machine is sufficient to break the SGX ecosystem, as these keys can be used to emulate attestation. Hence, ÆPIC Leak must also be mitigated on server CPUs.

Enforce x2APIC. ÆPIC Leak exploits that the legacy xAPIC and not the x2APIC is used on most systems. One of the main differences between xAPIC and x2APIC is the interface. The xAPIC is accessed using memory-mapped I/O (MMIO). This is the interface exploited with ÆPIC Leak. In contrast, the x2APIC does not support the MMIO interface for performance reasons [175]. Instead, the communication interface of the x2APIC is based on MSRs. We verified that the MMIO range is indeed disabled when enabling x2APIC, fully preventing ÆPIC Leak. Reads from the MMIO range when x2APIC is enabled return -1.

The x2APIC specification [175] states that switching from x2APIC to xAPIC is only possible by disabling the local APIC unit. As this can only be done by writing to the `IA32_APIC_BASE` MSR, a microcode update could enable the x2APIC at boot and prevent the disabling of the x2APIC. An enforced x2APIC is supported by Linux (tested on Ubuntu 20.04.1, kernel 5.4.0-96), and ensures that ÆPIC Leak cannot be mounted. This solution is the only firmware-based solution that does not incur any performance penalties. In case enforcing x2APIC mode would not be possible in microcode, an alternative solution would be to insert the APIC mode in the attestation process. The enclave attestation may simply fail if x2APIC mode is not enabled. As a positive side effect to fully mitigating ÆPIC Leak, enforcing x2APIC might even slightly improve the system performance.

Disable Caching for EPC. The EPC range is by default marked as write-back memory using a memory-type range register (MTRR). A microcode update could easily change the memory type for the EPC range to uncachable. As shown in Section 6.4.3, ÆPIC Leak cannot leak load or stores to uncachable memory. Hence, an uncachable EPC range would fully prevent ÆPIC Leak. Costanet al. [87] also proposed an uncachable EPC range to protect enclaves against cache attacks. While SGX explicitly supports an uncachable EPC range, it is not clear whether the memory type is part of the attestation [87]. Moreover, setting the entire EPC range to uncachable leads to a huge performance impact for enclaves, as no part of an enclave can benefit from caching anymore.

Flush Caches on EEXIT. As ÆPIC Leak leaks values traveling through the cache hierarchy, a possible mitigation is to flush all caches on an enclave exit (EEXIT). However, this is only sufficient if hyperthreading is disabled. With enabled hyperthreading, ÆPIC Leak can leak values from the enclave while it is running. Flushing caches with the same limitation, *i.e.*, disabling hyperthreading, is also the state-of-the-art mitigation for L1TF [177] and MDS attacks [179] on affected CPUs. The state of hyperthreading is already included in the attestation. Hence, SGX enclaves can also refuse to run if hyperthreading is enabled on the system.

We verified that the already-existing `wbinvd` successfully prevents the leakage if hyperthreading is disabled. The `wbinvd` instruction invalidates all cache levels and writes modified values back to the main memory. While we did not see any leakage after invalidating all cache levels, the invalidation is not very efficient. On average, we measured 321 655 cycles ($n = 5000$, $\sigma_{\bar{x}} = 406.3$) for executing the instruction. As this invalidation is required on every EEXIT, this mitigation has a huge impact on ECALL and OCALL latency. However, we expect that as the L1 flush MSR [177], Intel can implement an L2 flush MSR. As ÆPIC Leak is limited to leaking data moving between L2 and LLC, an LLC flush might not be necessary. While a huge performance overhead, such a buffer flushing was also used for L1TF and MDS attacks.

6.6.3 Software

As the OS or hypervisor are untrusted, mitigations implemented there are ineffective. However, mitigations can be implemented into the trusted software part of the SGX

ecosystem, such as the enclave itself, or indirectly via the attestation.

Secret Alignment. A limitation of ÆPIC Leak is that the first 4 bytes of every 16-byte block cannot be leaked. Hence, we propose a software solution that splits secrets and stores the parts of the secrets only in these non-leakable 4 bytes. Our software workaround is similar to the Intel-proposed workaround for SA-00219 [183]. For CPUs affected by SA-00219, no secrets can be stored in the first 8 bytes of a cache line. Hence, Intel added functionality to the SGX SDK to misalign buffers, ensuring that they do not start at the beginning of a cache line. For ÆPIC Leak it is more complicated, as only 4 consecutive bytes can be used, in contrast to the 56 bytes for SA-00219. We propose to rely on AVX scatter and gather instructions to automatically spread a secret over memory such that only the non-leakable parts of memory are used. Listing G.1 (Appendix G.1) shows a sample proof-of-concept implementation for 128-bit secrets, such as AES-NI keys. By relying on the scatter and gather instruction for single-precision floats, these functions can spread 4-byte blocks over one cache line. As the source and destination memory addresses are 64-byte aligned, all parts of the secret are overshadowed by valid APIC registers, hence there is no leakage.

This software workaround protects data at rest, as well as the loading from and storing to memory. However, there is still a small remaining attack surface left. When the secrets are already loaded to CPU registers, they are spilled to main memory on an (asynchronous) enclave exit. As this is done by the hardware, there is no possibility for the software to protect the secrets at this point. Thus, if an attacker triggers such an exit in the short time window where the secrets are in the CPU registers, the attacker can leak up to 96 bits of the secrets via the SSA. However, as ÆPIC Leak can only leak up to 96 bits of every *even* cache line, there are parts of the SSA that cannot be leaked (cf. Section 6.4.1). Hence, if only `xmm{2-5}` and `xmm{10-15}` are used for (round) keys, AES-NI can still be used securely inside an enclave.

Transient Secrets. As ÆPIC Leak leaks secrets that are moved between the L2 and the LLC cache, a possible software mitigation could also ensure that secrets never leave the CPU registers and the L1 cache. Previous work showed that it is possible to implement cryptographic algorithms, e.g., AES, by only using CPU registers [268]. However, in an enclave setting, an attacker can arbitrarily interrupt an enclave with high precision [360], forcing every register to be stored to main memory. Enclaves cannot opt-out from storing certain registers in the SSA [182]. Hence, the only workaround is to ensure that secrets are never architectural. With Mimosa, Guan et al. [151] leveraged hardware transactional memory to ensure secrets never leave the private cache and cannot be spilled to memory. Unfortunately, hardware transactional memory is not available on Sunny-Cove-based CPUs. As a more obscure variant, an enclave could leverage speculative execution to only work on secrets in the transient domain [375]. However, this would require an enclave to mount side-channel attacks to make the computed results visible. Listing G.2 (Appendix G.2) shows a sample code for realising this software workaround for AES encryption.

6.7 Conclusion

We presented ÆPIC Leak, the first architectural CPU vulnerability that allows leaking values from the cache hierarchy. ÆPIC Leak works on the newest Intel CPUs based on Ice Lake, Alder Lake, and Ice Lake SP and does not rely on hyperthreading enabled. ÆPIC Leak enables attacks against SGX enclaves on Ice Lake CPUs, forcing specific data into caches and leaking targeted secrets. We show attacks that allow leaking data held in memory and registers. We demonstrate how ÆPIC Leak completely breaks the guarantees provided by SGX, deterministically leaking AES secret keys, RSA private keys, and extracting the SGX sealing key for remote attestation. We finally propose several firmware and software mitigations that would prevent ÆPIC Leak from leaking sensitive data or completely prevent ÆPIC Leak.

This chapter showed how CPU vulnerabilities share the same root causes as software ones. We motivate it, pointing out how modern CPUs are effectively written as software. We cannot help but notice, though, how the inner working of modern CPUs is barely documented, and they are often treated as black boxes. Gaining insights into their inner workings would allow security researchers to examine the implementation of the defined CPU abstraction in search of inconsistencies that may result in novel bugs. In the next chapter, we break down the CPU abstraction and investigate the internals of modern CPU μcode. We exploit CPU vulnerabilities to gain control of CPU μcode and propose the first framework to trace and patch μcode, giving an unprecedented view of CPU internals.

Chapter 7

Reverse Engineering and Customization of Intel Microcode

7.1 Introduction

Microcode is the hidden software layer between the instruction set and the underlying hardware. In most Complex Instruction Set Architectures (CISC), each instruction, or *macro-instruction*, is translated into one or more *micro-operations* (*μops*) that are executed by the underlying hardware [161]. In total, there are over 2700 distinct μops in Intel x86 [199]. Many simple instructions map to a single μop. However, more complex instructions are essentially entire programs and can map to > 50 μops [182]. Microcode is a crucial optimization for these instructions, as μops are much simpler to implement in hardware and can be pipelined more efficiently [161].

Microcode is notoriously challenging to verify [92]. Independent auditing of microcode, subjecting it to static and dynamic analysis, would supplement manufacturers' verification efforts and help build trust in this hidden software. Moreover, tools enabling such analysis would facilitate research into CPU behavior. Dynamic microcode tracing, for example, would provide the fine-grained microarchitectural control that is so elusive in microarchitectural attack research [119] and in CPU fuzzing for undocumented behavior [106, 107, 118, 211] or hardware defects [222]. Furthermore, the ability to modify microcode would enable research into microcoded security mechanisms such as microcode-assisted address sanitization and customizable `rdtsc` precision [213].

Unfortunately, x86 microcode is confidential. Intel partially documented the μop sequences used for instructions in the Pentium Pro [174], but only μop counts have been published for instructions on newer CPUs, making reverse engineering research necessary [1]. Microcode has been reverse-engineered to enable customization on AMD Opteron [215] and Intel P6 (Pentium Pro to Pentium III) [52]. However, newer x86 CPUs have much stronger cryptographic protection for microcode updates. The updates are encrypted and signed to prevent unauthorized patching or reverse engineering, and the decrypted microcode never leaves the internal buffers of the CPU. Beyond the protection of intellectual property, security is a powerful motivation

for this cryptographic protection. Prior work has explored security concerns around microcode, such as the potential for backdoors [115, 346]. If the update mechanism were compromised, an attacker could maliciously patch microcode, for example, to introduce a backdoor to reveal cryptographic keys to JavaScript in the browser [346]. Reverse engineering research must, therefore, carefully balance the potential security benefits against the potential risks.

In recent years, microcode research has seen a considerable evolution thanks to the work of Ermolov et al. [123, 125, 126]. They achieved Red-Unlock on Goldmont and Goldmont Plus CPUs, a CPU mode that enables JTAG debugging of internal CPU components using external hardware [123]. Furthermore, they identified two undocumented instructions accessible on Red-Unlocked CPUs, `udbgrd` and `udgbwr`, that enable read/write access to internal microarchitectural components from software [126].

Motivated by this crucial breakthrough, in this chapter we build upon their work, posing the following research questions:

1. What are the semantics of microcode in modern Intel CPUs?
2. Is the microcode update process secure?
3. Could microcode customization bring security or performance benefits?

To answer these questions, we reverse-engineer microcode semantics and reconstruct microcode patching capabilities. We develop the first decompiler for Goldmont microcode to analyze how the CPU interacts with its internal components during microcode updates. We leverage the undocumented instructions to mimic these interactions to create microcode read and write primitives. Building upon these, we design and implement CustomProcessingUnit: the first framework for static and dynamic analysis of Intel microcode, supporting the Goldmont microarchitecture. Our framework can assemble microcode patches, install these in the CPU and then trace microcode execution in real-time, enabling CPU debugging at the μop level without additional hardware.

We leverage our framework to reverse-engineer the confidential Intel microcode update algorithm and analyze it in depth. For the first time, we perform a public and independent security analysis of the design and implementation of the update algorithm on Goldmont, evaluating its attack surface and possible security holes. Our analysis reveals a minor weakness in the implementation: the update is stored in the L2 cache during decryption, which is potentially exploitable, although we did not succeed in doing so. Such analysis is a first step toward independent auditing of microcode to verify manufacturers' security claims.

Moreover, in three additional case studies, we explore the benefits of CPU customization at the microcode level for performance and security. First, we bring Pointer Authentication Codes [18] to x86 for the first time, presenting a fast microcode implementation of pointer signing and verification in ~25 clock cycles. We thus show how we can enhance x86 CPUs with fundamental security concepts from other architectures. We evaluate the security of our x86 PAC implementation by reproducing the PACMAN attack [302] for the first time on x86. Thanks to our framework, we deepen the analysis by investigating alternative PAC implementations

that mitigate such an attack at the microcode level, providing the first public PAC implementation not vulnerable to PACMAN. Second, we design and implement fast software breakpoints, which we call *μsoftware breakpoints*, to execute the breakpoint handlers directly in microcode. This provides a speedup of ~1000x over `int3` instructions, showcasing how microcode customization can bring performance benefits. Third, we patch the `div` instruction to execute in constant time to prevent timing side-channel attacks [31], bringing a 1.6x speedup over state-of-the-art constant-time implementations, improving both performance and security.

In brief, we present the following contributions:

1. We introduce the first framework for static and dynamic analysis of Intel Goldmont (GLM) microcode for Atom CPUs, featuring support for microcode tracing and patching to provide complete control.

2. We demonstrate how our framework aids CPU reverse engineering by uncovering the details of the confidential Intel microcode update algorithm.

3. In three further case studies, we illustrate how complete control over microcode can bring security and performance benefits. We implement Pointer Authentication Codes (PAC) for x86, fast software breakpoints, and constant-time hardware division.

By sharing this research, we hope to make microcode research accessible to a broader audience and to help the community improve its understanding of microcode security guarantees. CustomProcessingUnit is open-source at https://github.com/pietroborrello/ghidra-atom-microcode (static analysis module) and https://github.com/pietroborrello/CustomProcessingUnit (dynamic analysis module). We hope that it will facilitate further auditing of Intel microcode and inspire the development of additional tooling for other CPUs.

Outline. We provide relevant technical background in Section 7.2. Section 7.3 presents our framework for static and dynamic analysis and describes the reverse engineering we conducted to create it. Sections 7.4, 7.5, 7.6, and 7.7 provide case studies of our framework, including reverse engineering of the microcode update algorithm in Section 7.4. We conclude in Section 7.8.

Ethical Considerations. Before deciding to publish our framework, we assessed its malicious potential. Our static analysis functionality requires decrypted microcode and does not compromise Intel's microcode update encryption and integrity validation. Our dynamic analysis functionality requires the CPU to be in Red Unlock mode. The only publicly-known method to achieve this requires exploitation of a patched vulnerability. It is only feasible on Intel's system-on-chip CPUs and, in practice, has been achieved on a small number of devices (see Section 7.2). While Red Unlock may be achieved on more devices in the future, in our assessment, the potential security benefits of making microcode analysis accessible to a broader audience outweigh this risk.

7.2 Background

In this section, we introduce the relevant technical background required for understanding the rest of the chapter. Note that the relevant background for our case studies is covered in their respective sections (7.4, 7.5, 7.6, 7.7).

7.2.1 Microcode Structure

The Instruction Decoding Unit (IDU) is the component responsible for translating CISC instructions to μops. Most Intel CPUs have multiple decoders: several simple decoders to translate x86 instructions that map to a single μop, a complex decoder to translate instructions that map to 1-4 μops, and a *Microcode Sequencer* responsible for translating microcoded instructions [191, 215]. Microcoded instructions are the most complex instructions that require advanced logic to be executed. Examples are `cpuid` that returns detailed information about the CPU and `wrmsr` that modifies internal settings in model-specific registers (MSRs).

The microcode is stored in a dedicated read-only memory inside the CPU (MSROM). Instruction definitions are organized in *triads*, consisting of three μops and a sequence word. Sequence words affect the control flow of the executed triad and can also act as synchronization primitives for the dataflow akin to `lfence` instructions. Goldmont (GLM) CPUs have space for 7936 triads in the MSROM [122].

7.2.1.1 Microcode Patches

Modern CPUs allow the microcode to be updated at runtime with microcode patches. This enables patching of bugs in complex instructions [75] and implementation of new features. Thus, the CPU needs a dedicated writable region to hold the patches (MSRAM). On GLM, there is space for 128 triads in the MSRAM [122]. Updates are applied either by the BIOS at boot time or by the operating system. For Intel CPUs, the update routine is triggered by writing the virtual address where the microcode patch has been loaded to the MSR `IA32_BIOS_UPDT_TRIG`. Intel's updates are signed and encrypted [66, 125, 158]. For GLM and GLM Plus, Ermolov et al. documented that the update is RSA signed and RC4 encrypted, providing a decryption algorithm [124].

7.2.1.2 Microcode Hooks

To run patched microcoded instructions, a *microcode hook* redirects control flow from the MSROM to the MSRAM [125, 213]. To implement the hooks, microcode updates set the *match registers* in the CPU to the microcode addresses that need to be redirected. Each time the microcode is executed from the MSROM at an address contained in one of the match registers, the control flow is automatically redirected to the corresponding patch address in the MSRAM. On GLM, there is space for up to 64 hooks [122].

7.2.1.3 Control Register Bus (CRBUS) and Local Data Access Test Port (LDAT)

The CRBUS is an internal bus that connects all internal CPU units and exposes core controls and configurations (e.g., control registers and some MSRs are mapped) [126]. Each unit has its own range of addresses on the bus that can be used to query its state and update its configuration. It is used by microcode but is not intended to be architecturally accessible to software. LDAT is a debug interface between local CPU arrays and the CRBUS that facilitates post-silicon validation [233]. Each array provides an LDAT port for read/write access over the CRBUS. Combined, the CRBUS and LDAT provide access to the internal state of CPU core units such as the Microcode Sequencer, Instruction Fetch Unit, caches, and TLB [51].

7.2.1.4 Red Unlock

This special mode provides access to the CRBUS and LDAT via JTAG using a USB debug cable or proprietary hardware [126]. In Intel's threat model, Red Unlock (which they refer to as 'Protection Class Intel') is only possible with Intel's authentication key [186]. However, Ermolov et al. published a proof of concept to Red Unlock GLM by exploiting a (now patched) vulnerability in the Intel Management Engine (ME) [123]. Leveraging Red Unlock, they exported the MSROM and MSRAM contents and reverse-engineered the format of μops, providing a disassembler for GLM microcode. The proof of concept has also been ported to Skylake and Kaby Lake [5]. In contrast to GLM, this only enables Red Unlock on the ME rather than the CPU because only Intel's system-on-chip designs have a shared DFX AGG unit (and, therefore, a shared unlock state) for the chipset and main CPU cores [126].

7.2.1.5 Undocumented Debug Instructions

Ermolov et al. discovered the existence of two undocumented instructions on Intel CPUs, `udbgrd` and `udbgwr` [126]. These instructions are the final puzzle piece for microcode customization. On Red-Unlocked CPUs, they can be used to read and write all of the internal components made accessible by the CRBUS and LDAT from software without any additional hardware. Crucially, this includes the Microcode Sequencer arrays [126]. While only GLM and GLM Plus have been publicly Red-Unlocked, the existence of these instructions on other microarchitectures has been inferred using performance counters [44]. In this chapter, we leverage these instructions to create the first analysis framework for observing and modifying CPU microcode.

7.3 Framework

In this section, we present our framework for microcode reverse engineering and customization. For static analysis, we develop a Ghidra module to enable microcode decompilation and reverse engineering. For dynamic analysis, we implement a UEFI application to trace and patch microcode execution.

```
4  void rc4_decrypt(ulong i,ulong j,byte *ptr,int len,byte *S,long callback)
5
6  {
7    byte bVar1;
8    byte bVar2;
9
10   do {
11     i = i + 1;
12     bVar1 = S[i];
13     j = bVar1 + j;
14     bVar2 = S[j];
15     S[i] = bVar2;
16     S[j] = bVar1;
17     *ptr = S[bVar2 + bVar1] ^ *ptr;
18     ptr = ptr + 1;
19     len += -1;
20   } while (len != 0);
21   (*(callback * 0x10))();
22   return;
23 }
24
```

Figure 7.1. Microcode decompiled within Ghidra using our processor module. This function, `rc4_decrypt`, is used in the microcode update routine.

7.3.1 Static Analysis

We leverage the findings of Ermolov et al. [126] to implement a processor module for the Ghidra decompiler. They provide decrypted GLM microcode and the contents of the Microcode Sequencer arrays on GitHub [125]. The Microcode Sequencer array content can also be extracted using CustomProcessingUnit (see 7.3.2).

We define the semantics of each μop by reconstructing their effects from the naming scheme in the published disassembler (which, in turn, was constructed by observing the effect of μops on registers and from leaked opcode lists from Intel [125]). For the μops that were not straightforward to understand, we conduct dynamic analysis (see 7.3.2) to observe the side effects of these μops in isolation. In total, we define semantics for 8350 μops in Ghidra's processor specification language, SLEIGH [117].

Our static analysis module takes as input the dump of microcode ROM and RAM (including sequence words) and packs them as a binary blob parsable by our Ghidra processor module. For each triad, the packer analyzes the sequence words relative to that triad, and encodes the sequence-word semantics in the respective μop of the triad. Thus, for each triad of four 48-bit μops (including the `nop` at the end) and one 32-bit sequence word, it emits four 128-bit μops to be analyzed by the Ghidra processor module. Removing the concepts of triads and sequence words simplifies the design of the Ghidra processor module.

CPU microcode is highly optimized: several basic blocks are shared between functionalities, there is no distinction between jumps and calls, and basic blocks are highly interleaved among each other to optimize for code reuse (and thus size) instead of code locality. The decompiler conducts basic analysis to identify function boundaries, reconstruct high-level control flow and internal data structures, and cross-reference these. Figure 7.1 shows an example of clean control flow reconstructed in Ghidra using our processor module. We leverage our decompiler to analyze microcode throughout this chapter.

```
def ucode_sequencer_write(SELECTOR, ADDR, VAL):
  CRBUS[0x6a1] = 0x30000 | (SELECTOR << 8)
  CRBUS[0x6a0] = ADDR
  CRBUS[0x6a4] = VAL & 0xffffffff
  CRBUS[0x6a5] = VAL >> 32
  CRBUS[0x6a1] = 0

with SELECTOR:
2 -> SEQW Patch RAM
3 -> Match Registers
4 -> UCODE Patch RAM
```

Listing 7.1. Slightly simplified sequence of commands to write the value `VAL` to the address `ADDR` of the selected (with `SELECTOR`) microcode array. Each `CRBUS` write is an invocation of the `udbgwr` instruction with `rcx=0`.

7.3.2 Dynamic Analysis

We now introduce our framework's microcode dynamic analysis module, which is the first of its kind for Intel CPUs. In this section, we describe the reverse engineering we conducted to build it, present the implementation details, and customize `rdrand` as an example of instruction patching.

Execution Context The module is capable of hooking, patching, and fully tracing microcode execution. It consists of a UEFI application that runs before the OS bootloader. This provides a noiseless environment for experiments and complete control over the system. Alternatively, the same MSR operations could be implemented in a Linux kernel module, compromising increased noise for a more feature-rich execution environment.

Hardware Setup All our tests are performed on GLM, namely Intel Celeron N3350 with cpuid `0x000506C9` and `0x000506CA`. We execute all our experiments with a fixed CPU frequency of 1.10 GHz.

7.3.2.1 Reverse-Engineering LDAT Accesses

We use our decompiler to reverse-engineer the CRBUS access patterns during microcode updates that allow the CPU to overwrite the MSRAM. Our starting point is the CRBUS address range mapping to the LDAT port of the Microcode Sequencer, which has been documented in prior work [51]. By cross-referencing these addresses with the decompiled microcode, we can find and analyze the microcode update routine. This lets us interact with the Microcode Sequencer by reproducing the commands that the update routine sends to its LDAT port. Listing 7.1 shows our primitive to write to the Microcode Sequencer's arrays. We build our dynamic analysis module upon our `ucode_sequencer_write` primitive, developing a similar primitive to read from the microcode arrays.

7.3.2.2 Microcode Hooks

To modify the behavior of an instruction in a custom microcode patch, we need to configure the match registers to 'hook' that instruction. By dumping the content of the match registers after a regular microcode update has been applied, we can reverse-engineer the format of the match-register entries. The following snippet shows how to compute a match entry register to hook an MSROM instruction address `match_address` to redirect execution to the MSRAM `patch_address`:

```
def compute_match_register(match_address, patch_address):
  patch_offset = ((patch_addr - 0x7c00) / 2) << 16;
  return (0x3e000000 | patch_offset | match_address |
    enabled)
```

Note that the last bit of `match_address` overlaps with the enabled bit, which, when set to 0, disables the hook. The last bit of the match address is ignored by the CPU, and, to our understanding, only even addresses can be hooked.

7.3.2.3 Microcode Patches

Combined with our static analysis capabilities, we can modify instruction behavior. To better express the desired semantics of our patches, we develop a microcode assembler. Our assembler supports most μops and hides complex details such as instruction addresses and registers by supporting high-level constructs like labels and variables. Hooks can be generated with the `.patch` directive to set up a match-register entry automatically. As μop immediates are restricted to 16 bits, the assembler also supports macros to deal with 64-bit constants by emitting multiple instructions. Listing 7.3 shows an example of a microcode routine we can assemble.

7.3.2.4 Microcode Traces

By leveraging microcode patches and hooks, we can obtain microcode execution traces. We define a specific microcode patch that when executed reads the timestamp counter of the CPU, saves it in a specific location, disables itself and then continues execution. The hook disables itself by zeroing out the corresponding entry in the match register, accessing the CRBUS similarly to our `ucode_sequencer_write` primitive. The ability of the hook to disable itself is fundamental to making the microcode tracing work: the CPU would otherwise enter an infinite loop when resuming execution at the same address.

We apply hooks that redirect execution to our custom patch to every possible microcode address. Thus, when executing an instruction `I`, the framework dumps the timestamp at which each specific μop has been executed. Since there are a limited number of match registers, the framework iteratively executes the instruction `I`, each time registering a subset of the hooks it needs and collecting subtraces. In our implementation, we register one hook at a time for simplicity. In a post-processing stage, we reorder the μops based on the timestamp to obtain an instruction trace. Since the instruction `I` is executed multiple times, it must have a deterministic microcode control flow to obtain a coherent trace.

Algorithm 2 shows the pseudocode of the microcode hook that is installed in the CPU to collect the timestamp counter and resume execution. Algorithm 3 shows

Algorithm 2: Pseudocode of the microcode hook that dumps the timestamp and resumes execution.

```
// Assume this hook is installed at index
// 0 of the match registers
function dump_ts_and_resume(addr)
   saved_ts ← read_clock()
   // disable hook by overwriting
   // the entry
   ucode_sequencer_write(
      sel:  3, // select match registers
      idx:  0, // assume idx 0
      val:  0  // 0 to disable
   )
   resume_execution(addr)
```

the pseudocode of the tracing collection stage algorithm. As we can only hook *even* microcode addresses, when two even addresses are executed contiguously, we infer in the post-processing stage that the *odd* address between the two has also been executed.

7.3.2.5 Customizing `rdrand`

With CustomProcessingUnit's microcode hooking, patching, and tracing capabilities, we can customize the semantics of x86 instructions. As a proof of concept, we customize the behavior of the `rdrand` instruction. In most x86 processors, this generates a hardware-generated cryptographically secure random number. The carry flag in the `rflags` register indicates the success or failure of the operation after execution.

We trace the execution of `rdrand` to determine which μop to hook. Listing 7.2 shows the full execution trace. Analysis of the trace shows the semantics of the instruction reflected in its μops: it reads the hardware-generated random number by reading I/O port `0x40004e00` at address `0x1866`, whose value is returned in the specified register (μop `0x186a`). The carry flag is updated based on the value returned, conditionally assigning the bit 1 to the temporary register `tmp2` (address `0x1869`) and updating the `rflags` register (address `0x18c`).

To patch the instruction semantics, we only need the instruction entry point in the MSROM: address `0x0428`. We use CustomProcessingUnit to hook this entry point and redirect execution to our custom patch in MSRAM. Listing 7.3 shows our patch to make `rdrand` return `"Hello World!"` in the registers `rax` and `rbx`. We verify the patch works by repeated execution of `rdrand`.

In the following sections, we demonstrate how CustomProcessingUnit facilitates microcode reverse engineering and customization. We reverse-engineer the confidential microcode update algorithm and present three other case studies illustrating how microcode customization can improve software performance and security.

Algorithm 3: Pseudocode of the microcode tracing algorithm.

```
function trace_instruction(I)
    trace ← []
    // microcode addresses go
    // from 0 to 0x7c00 in GLM
    for addr in 0 ..  0x7c00 do
        install_hook(addr)
        saved_ts ← 0
        start_ts ← read_clock()
        // this triggers the dump_ts_and_resume hook if the
           microcode of I executes ‘addr‘.
        // the microcode hook removes itself and saves the
           timestamp in a global variable ‘saved_ts‘
        execute(I)
        end_ts ← saved_ts
        if end_ts > 0 then
            trace.append(end_ts - start_ts, addr)
    return sort(trace)
```

```
rdrand_trace:
0428: tmp4:= ZEROEXT_DSZ32(0x0000002b)
0429: tmp2:= ZEROEXT_DSZ32(0x40004e00)
042a: tmp0:= ZEROEXT_DSZ32(0x00000439) SEQW GOTO U1861
1861: tmp1:= READURAM(0x0035, 64)
1862: TESTUSTATE(SYS, 0x20)? SEQW GOTO U1866
1866: tmp1:= PORTIN_DSZ64_ASZ16_SC1(tmp2)
1868: tmp1:= OR_DSZ64(0x00000000, tmp1)
1869: tmp2:= SELECTCC_DSZ64_CONDNZ(tmp1, 0x00000001)
186a: r64dst:= ZEROEXT_DSZ32N(tmp1)
186c: MOVEINSERTFLGS_DSZ32(tmp2) SEQW UEND0
```

Listing 7.2. μop execution trace of the `rdrand` instruction.

7.4 Case Study: Reverse-Engineering the Microcode Update Routine

The details of the decryption and validation performed in the microcode update routine are not documented by Intel [182]. An Intel patent describes that the patch is validated in a secure memory separate from the microcode RAM and is encrypted using public-key encryption. In particular embodiments, a private key and public key hash value are embedded within the CPU, and the patch is signed with 2048-bit RSA [347]. Experimental timing and fault analysis in prior work supports the hypothesis that 2048-bit RSA is used to sign a padded SHA2 digest [66, 158].

We leveraged the microcode tracing and decompilation capabilities of CustomProcessingUnit to precisely reverse-engineer the full microcode routine for patch decryption and verification. We release our microcode decryptor and parser along with CustomProcessingUnit. Concurrently to our work, Ermolov et al. released a

```
.org 0x7c00
.patch 0x0428 # RDRAND ENTRY POINT
rax:= ZEROEXT_MACRO(0x6f57206f6c6c6548) # "Hello Wo"
rbx:= ZEROEXT_MACRO(0x21646c72) # "rld!\\x00"
```

Listing 7.3. `rdrand` patch that makes the instruction return "Hello World!" in the registers `rax-rbx`.

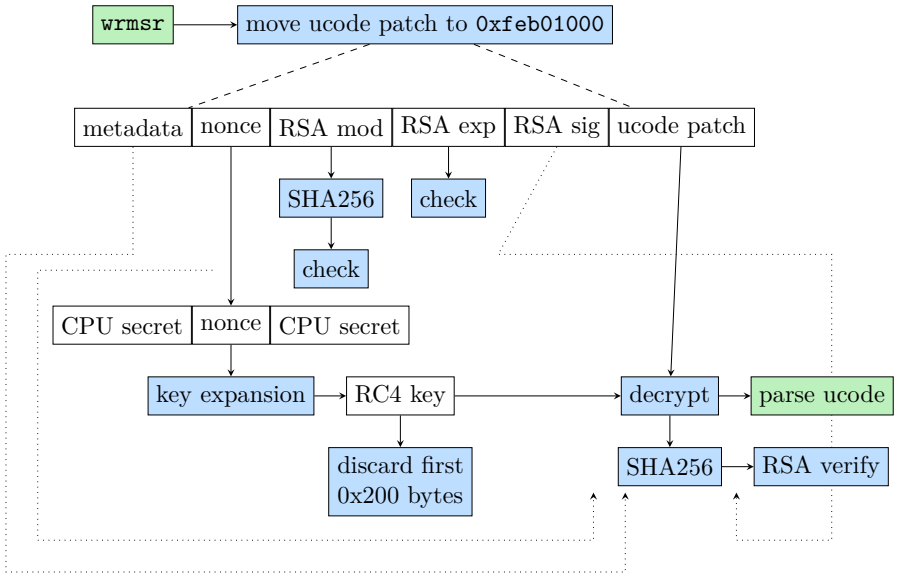


Figure 7.2. High level overview of the μcode update algorithm for GLM CPUs.

tool to decrypt microcode updates based on their GLM reverse engineering [124].

7.4.1 Reverse Engineering

A microcode update is triggered by writing the linear address of the microcode update loaded in memory to the `IA32_BIOS_UPDT_TRIG` MSR. To trace the update with our microcode tracer, we need to repeat the instruction sequence (see Section 7.3.2.4). Since the same μcode update cannot be applied multiple times, and applying a μcode update would override the MSRAM and match register that we use for tracing, we must corrupt the microcode update so that the update fails.

We corrupt the single byte that produces a failing update with the maximum latency. Intuitively, the failing update with the maximum latency should provide a repeatable μop trace that most closely resembles a successful update. We identify this as byte offset `0x1c0` in the update signature. This produces an update that fails at the signature verification stage, just before actually applying the update to MSRAM.

Using CustomProcessingUnit, we produce a full trace of the faulty microcode update routine and reverse-engineer it with the help of our decompiler. Figure 7.2 shows a high-level overview of the algorithm, illustrating how the important parts of the μcode update are parsed and verified by the CPU before applying the update.

Decryption Algorithm The CPU first checks the validity of the pointer passed to the `wrmsr` instruction, erroring out for non-canonical addresses, addresses that may wrap around the address space, and patch sizes $> 256KB$ or $< 646B$. After these basic checks, the μcode update routine enables a secure memory region at physical addresses `0xfeb00000-0xfec00000`, setting the last bit of the CRBUS address `0x51b`. It then copies the μcode patch from the user-specified address to `0xfeb01000` in the secure region to check and decrypt it in place. The range `0xfeb00000-0xfeb01000` of the secure memory is used as a scratch memory area to temporarily hold variables and metadata for the cryptographic algorithms used in the routine.

After copying the patch, the routine checks basic metadata. It ensures that the `cpuid` of the CPU is supported by the update and that the security version number of the new update is not lower than the number in the currently loaded update. It then initializes a SHA256 state in the scratch range and computes the SHA256 of the RSA modulus stored in the μcode update (bytes 0xb0 to 0x1b0), verifying that it matches a known hard-coded hash (`a1b4b7417f0fdcdb0feaa26eb5b78fb2cb86153f0ce98803f5cb84ae3a45901d`). It checks that the RSA exponent (bytes 0x1b0 to 0x1b4) is 0x11 (17).

The routine then proceeds to compute the decryption key for the μcode update. It generates a seed combining a 32-byte nonce from the μcode update (bytes 0x90 to 0xb0) with a 16-byte prefix and suffix. This is a hard-coded value from the μcode routine (`0e77b29d9e91765da26648998b6813ab`) which we call the CPU secret. The 64-byte seed is expanded to 256B by recursively computing the SHA256 hash eight times and saving each 32-byte internal state. The resulting 256B are used to initialize an RC4 keystream.

The first 512B of the RC4 keystream are discarded, and then the μcode routine proceeds to decrypt the microcode using RC4. Next, the decrypted μcode is cryptographically verified. The routine computes the SHA256 of the resulting patch, including patch metadata (e.g., μcode revision number, release date, length, cpuid target, and nonce), and verifies this against the RSA signature.

Patch Application Once the update is verified, the patch is applied. Reverse engineering the decrypted microcode update patch routine shows that a microcode update is a custom bytecode that the CPU decodes in an interpreter loop to execute various commands while updating. Such commands include: resetting or writing microcode RAM, sequence words and match registers; sending commands to internal components through the CRBUS (e.g., to disable match registers during the update); writing internal buffers; invoking custom microcode routines; and even control flow commands to decode different commands based on the CPU state.

Secure Memory To prevent the decrypted patch from being read or tampered with, the update process must use a secure memory region. We observe that microcode is decrypted at the temporary physical address `0xfeb01000`. Attempting to read this address during normal execution returns `0xff`, as occurs when trying to read other protected memory regions such as SGX enclave memory. The address appears to be dynamically enabled for the microcode update process by writing a

bit to the CRBUS address `0x51b`. It has a fast access time ($\approx$ 20 cycles), fits up to 256KB of data before a replacement policy is applied, and content is not shared between cores. Based on these observations, we conclude that this address is actually a special view on the L2 cache. This matches an embodiment described in an Intel patent, in which access to a cache is blocked to all other operations during decoding, validation, and installation of the update [347].

7.4.2 Security Analysis

The microcode is encrypted and signed to prevent reverse engineering and tampering. In this section, we investigate the effectiveness of the actual update-routine implementation towards those guarantees. Bypassing encryption would allow microcode to be analyzed on other Intel microarchitectures for which the CPU secret has not yet been leaked, while bypassing anti-tampering would enable the loading of arbitrary microcode on CPUs without Red Unlock, with all the associated security risks and customization benefits that entail. Although we specifically analyze our reverse-engineered GLM implementation, we expect many of these findings to generalize to other microarchitectures.

RC4 Encryption RC4 is vulnerable to many known attacks [46, 209], although some mitigations are in place in the routine. The key (CPU secret) length of 16B is sufficient to prevent brute-forcing, and the additional use of a unique 16B nonce (*i.e.*, initialization vector, or IV) to initialize the key expansion prevents keystream reuse attacks [46]. As the IV is included in the RSA signature, the attacker cannot modify it, preventing chosen-IV attacks. RC4's weak key schedule leads to statistical bias in the generated keystream, making it vulnerable to related-key recovery attacks when the key and IV are combined together trivially, for example, via concatenation as they are in the microcode routine [209]. However, the routine does discard the initial 512B of the keystream to reduce bias. While attacks have been demonstrated exploiting bias in later bytes, these require high volumes of ciphertexts (> 1000000 [209]), whereas very few microcode updates are published. Currently, only 453 Intel production microcode patches are available in a well-known repository [255], and as these cover multiple microarchitectures, they also do not all use the same key. Therefore, these mitigations are likely sufficient in practice, provided that (as is extremely likely) microcode updates continue to be produced in low volumes.

RSA Signature The use of RSA provides strong anti-tampering protection. The 2048-bit modulus is sufficient to prevent brute-forcing, PKCS#1 v1.5 padding is used, and the signature check appears experimentally to be constant-time [158].

Several methods are known to bypass RSA signatures when the public key is not correctly checked [40, 260]. However, while the RSA modulus and exponent for the signature are provided directly in the update metadata, and are therefore attacker-controlled, they must match the expected values hard-coded in the update routine. Thus, it is not possible to pass different public key information, replacing the modulus or exponent to bypass the signature verification.

The hash that the RSA signature is computed on includes the μcode update metadata (see Section 7.4). This prevents metadata tampering, such as changing the security revision number to downgrade the microcode or modifying the cpuid value to apply a patch for a different CPU model. However, some of these metadata values are used before being verified. The length of the microcode update is included in the hash, but it is actively used before. Since the algorithm includes checks on the minimum and maximum values, we could not leverage such a potential time-of-check-time-of-use vulnerability to, e.g., cause integer overflows in the routine.

Corrupting Updates An attacker could try to leverage a race condition and corrupt the decrypted microcode in memory before it is applied. However, the microcode is decrypted in place inside the reserved secure memory area that we hypothesize is the L2 cache. The view on the secure memory is only enabled during microcode updates, and cores trying to access its physical address during normal execution read only `0xff`. We verified that each core has a unique secure memory area (as the L2 cache is not shared); thus, its content cannot be modified by a different core during an update. Since the update is triggered by a serializing `wrmsr` instruction, the hyperthread parallel to the logical core executing the update is frozen and cannot modify the secure memory either.

We can obtain partial leaks of decrypted microcode. We achieve this by mapping the APIC MMIO region at the address `0xfeb01000` that the microcode will be moved to. This effectively makes the APIC MMIO region shadow the secure memory region and hijack it from the microcode routine, similar to the Memory Sinkhole attack [105]. However, we cannot fully leak or corrupt updates due to the limitations on the writable areas of the APIC MMIO region [182].

An alternative would be to map the APIC to the scratch region used in the secure memory to corrupt the SHA256 state used to compute hashes. While this is possible and corrupts the output of the hash algorithms (which would bypass the signature check), it also corrupts the hash of the RSA public key, which is checked, and thus the update fails.

Microarchitectural Attacks One could try to leverage microarchitectural attacks to leak microcode updates during decryption. Most microarchitectural attacks require execution on either the same core or a hyperthread during the update and can only leak internal buffers close to the CPU, such as the L1 cache or Line Fill Buffers [59, 236, 314, 358, 368]. The secure memory is accessed in μcode using μops that access directly uncacheable physical addresses, thus avoiding caches. While the Line Fill Buffers could be filled with such data, hyperthreading cannot be used during the update, and internal buffers seem to be flushed afterward, making the attack ineffective.

ÆPIC Leak targets structures deeper in the memory hierarchy, *i.e.*, the superqueue [49]. However, as the superqueue holds entries flushed from the L2 cache to the LLC, the decrypted microcode does not pass through. An inverse attack could attempt to leverage ÆPIC Leak to insert values from the superqueue. By modifying the content of the superqueue before the update and once again mapping the APIC MMIO region over the secure scratch memory, an attacker could make the CPU

read leaked values from the superqueue instead and thus corrupt computations with finer control. However, the μcode update routine also flushes internal buffers *before* the update.

7.5 Case Study: x86 Pointer Authentication Codes

7.5.1 Background

Pointer Authentication Codes (PACs) aim to protect sensitive pointers from attacks that may leverage memory corruption vulnerabilities to hijack the control flow. PACs were introduced in ARMv8.3 [18] and are used in several ARM-based systems to provide strong security guarantees [297]. A PAC is a message authentication code embedded in the high bits of the pointer it protects. The code depends on the pointer itself, a 64-bit context, and a secret key from a set of five possible keys. The algorithm used to compute the code is vendor-specific [23], but the standard recommends the QARMA family of tweakable block ciphers [22]. On most CPUs, PAC is implemented directly in hardware in a single μop [109].

The ARM instruction set provides different instructions to compute the PAC and embed it in the pointer (e.g., `pacia, pacib`), to verify and remove it (`autia, autib`), or to simply clear it. The suffix of the instruction selects which key is used to compute or verify the signature. Upon successful verification, `aut` instructions remove the PAC from the high bits of the pointer, while on verification failure, the bits are not cleared. Thus, once signed, a pointer can be accessed only after being verified, and causes a memory access fault otherwise.

7.5.2 Implementation

We present the first public implementation of PAC on x86, enabling cheap and strong hardware-based control-flow-integrity protection on Intel platforms. We implement instructions to sign and verify 64-bit pointers leveraging CustomProcessingUnit.

As a proof of concept, we define two new microcode routines that sign and verify a context and a pointer, with a PAC saved in the pointer's high bits. The size of the PAC can be customized in our design, provided that it does not conflict with the used bits in the pointers, and any unused microcoded instructions can be chosen as the signing and verification instructions. By default, we select a 16-bit PAC and patch `verw` and `verr`, programming the match registers to hook them with our routines.

Our microcoded signing algorithm uses a single round of SipHash [20]. The SipHash algorithm has been developed for keyed hashing optimized for small inputs, which is exactly our use case with 64-bit pointers. We favor it over QARMA as the latter uses bit shuffles, which are faster in hardware but costly in microcode. We define a 64-bit secret key to be kept in an internal buffer of the CPU, the staging buffer, so that it is never architecturally exposed.

Listing 7.4 shows our microcode for the PAC signing algorithm. The authentication routine is similar, but in addition, it verifies that the existing PAC on the input pointer matches the one generated from scratch, corrupting the high bits if they do not match. Our x86 microcode implementation of the PAC signature takes 25

```
.org 0x7c00
# declare variables
let [ptr] := r64dst;        let [v0] := tmp1
let [ctx] := r64src;        let [v1] := tmp2
let [key] := tmp0;          let [v2] := tmp3
let [key_addr] := 0xba40;   let [v3] := tmp4
let [pac] := tmp5

# --- initialize ---
[key] := LDSTGBUF_DSZ64_ASZ16_SC1([key_addr])
# v0 = 0x736f6d6570736575 ^ key;
[v0] := ZEROEXT_MACRO(0x736f6d6570736575)
[v0] := XOR_DSZ64([v0], [key])
# v1 = 0x646f72616e646f6d ^ ctx;
[v1] := ZEROEXT_MACRO(0x736f6d6570736575)
[v1] := XOR_DSZ64([v1], [ctx])
# v2 = 0x6c7967656e657261 ^ key;
[v2] := ZEROEXT_MACRO(0x736f6d6570736575)
[v2] := XOR_DSZ64([v2], [key])
# v3 = 0x7465646279746573 ^ ctx;
[v3] := ZEROEXT_MACRO(0x736f6d6570736575)
[v3] := XOR_DSZ64([v3], [ctx])
# --- update ---
[v3] := XOR_DSZ64([v3], [ptr]) # v3 ^= ptr;
[v0] := ADD_DSZ64([v0], [v1]) # v0 += v1;
[v2] := ADD_DSZ64([v2], [v3]) # v2 += v3;
[v1] := ROL_DSZ64([v1], 0x0d) # v1 = RotateLeft<13>(v1);
[v3] := ROL_DSZ64([v3], 0x10) # v3 = RotateLeft<16>(v3);
[v1] := XOR_DSZ64([v1], [v0]) # v1 ^= v0;
[v3] := XOR_DSZ64([v3], [v2]) # v3 ^= v2;
[v0] := ROL_DSZ64([v0], 0x20) # v0 = RotateLeft<32>(v0);
[v2] := ADD_DSZ64([v2], [v1]) # v2 += v1;
[v0] := ADD_DSZ64([v0], [v3]) # v0 += v3;
[v1] := ROL_DSZ64([v1], 0x11) # v1 = RotateLeft<17>(v1);
[v3] := ROL_DSZ64([v3], 0x15) # v3 = RotateLeft<21>(v3);
[v1] := XOR_DSZ64([v1], [v2]) # v1 ^= v2;
[v3] := XOR_DSZ64([v3], [v0]) # v3 ^= v0;
[v2] := ROL_DSZ64([v2], 0x20) # v2 = RotateLeft<32>(v2);
[v0] := XOR_DSZ64([v0], [ptr]) # v0 ^= ptr;
# --- finalize ---
[v2] := XOR_DSZ64([v2], 0xff) # v2 ^= 0xFF;
[v0] := ADD_DSZ64([v0], [v1]) # v0 += v1;
[v2] := ADD_DSZ64([v2], [v3]) # v2 += v3;
[v1] := ROL_DSZ64([v1], 0x0d) # v1 = RotateLeft<13>(v1);
[v3] := ROL_DSZ64([v3], 0x10) # v3 = RotateLeft<16>(v3);
[v1] := XOR_DSZ64([v1], [v0]) # v1 ^= v0;
[v3] := XOR_DSZ64([v3], [v2]) # v3 ^= v2;
[v0] := ROL_DSZ64([v0], 0x20) # v0 = RotateLeft<32>(v0);
[v2] := ADD_DSZ64([v2], [v1]) # v2 += v1;
[v0] := ADD_DSZ64([v0], [v3]) # v0 += v3;
[v1] := ROL_DSZ64([v1], 0x11) # v1 = RotateLeft<17>(v1);
[v3] := ROL_DSZ64([v3], 0x15) # v3 = RotateLeft<21>(v3);
[v1] := XOR_DSZ64([v1], [v2]) # v1 ^= v2;
[v3] := XOR_DSZ64([v3], [v0]) # v3 ^= v0;
[v2] := ROL_DSZ64([v2], 0x20) # v2 = RotateLeft<32>(v2);

# pac =  ((v0 ^ v1) ^ (v2 ^ v3)) << 48;
[pac] := XOR_DSZ64([v0], [v1])
[pac] := XOR_DSZ64([pac], [v2])
[pac] := XOR_DSZ64([pac], [v3])
[pac] := SHL_DSZ64([pac], 0x30)
# sign ptr
[ptr] := XOR_DSZ64([pac], [ptr])
```

Listing 7.4. x86 PAC signature microcode routine leveraging a single round of SipHash.

clock cycles to execute, while the authentication operation takes 26. We verify that the implementation works as expected by signing pointers and verifying that they authenticate when not tampered with and cause an invalid memory access otherwise.

7.5.3 Security Analysis

For the security analysis of our x86 PAC implementation, we focus on the resistance of our implementation to speculative execution attacks such as PACMAN [302]. We refer the reader to the original SipHash paper for a security evaluation of the SipHash algorithm [20].

PACMAN This attack bypasses pointer authentication by speculatively authenticating a corrupted pointer, using side channels to identify a correct PAC. Because the authentication routine is speculatively rather than architecturally executed, the attack can brute-force the PAC without causing the program to crash.

We re-implement the attack on our CPU, effectively developing the first x86 PACMAN attack. As in the original paper, we leverage a PACMAN gadget that authenticates and then accesses a pointer behind a branch whose direction we can determine. We train the branch prediction by executing the branch with a valid pointer with a correct PAC for 10 iterations. We then use an artificial memory-corruption vulnerability to override the pointer with an attacker-controlled value and PAC and change the branch condition so the authentication instructions are no longer architecturally executed. While the gadget is not executed architecturally, it is executed speculatively, and the pointer, if valid, is dereferenced speculatively. For simplicity, we make the pointer point to shared memory between the attacker and the victim and use Flush+Reload [394] to infer whether the pointer location was accessed, *i.e.*, the PAC was correct.

Implementing the attack on our x86 PAC implementation fails to leak valid PAC values. This is due to the smaller Reorder Buffer (ROB, 78 entries) and Physical Register File (PRF, 56 entries) in GLM CPUs [15]. Our x86 PAC implementation consists of 54 μops, filling up the entire speculative window and preventing subsequent speculative access. However, on a CPU with a wider ROB and PRF, the attack would succeed as it is the PAC design, not the implementation, that is vulnerable to PACMAN. To test this hypothesis, we write a weaker version of the PAC implementation in 27 μops, which only partially implements SipHash. With this shorter PAC implementation, the attack successfully brute-forces a valid PAC for a given pointer in less than a second.

Mitigation We leverage CustomProcessingUnit to investigate how PACMAN could be mitigated in microcode. As a first attempt, we could add a speculation barrier to the PAC implementation. However, such a solution would incur considerable overhead, slowing the execution of the PAC instruction (by around 10 cycles) and other instructions in the pipeline. Moreover, an attacker could find a gadget where the pointer authentication occurs on a non-speculative path while the access occurs on a speculative one, re-enabling the attack.

Thus, we investigate a solution that does not involve fences. The core concept is to make the pointer-authentication operations fault when an invalid PAC is detected

while also removing the PAC from the pointer. This means that the pointer is always valid in the speculative path, removing the side channel on PAC validity. Faulting ensures that memory corruption attacks using corrupted pointers architecturally still fail. Our mitigated implementation that triggers an exception when detecting an invalid PAC has no overhead with respect to the original, and is the first public PAC implementation not vulnerable to the PACMAN attack. However, one drawback is that this design re-enables Spectre attacks [210] using corrupted pointers in the speculative path. A further remaining attack surface is the use of a port contention side-channel [6] to detect the micro-operations that cause a fault in the speculative path. This attack surface could be closed by disabling hyperthreading or preventing an attacker from running parallel to the victim.

7.6 Case Study: μsoftware Breakpoints

7.6.1 Background

Software breakpoints are widely used both for debugging [140] and instrumentation [129, 244, 272, 406]. The `int3` (`0xcc`) instruction triggers an interrupt calling the debug exception handler with a breakpoint exception [182]. The interrupt routine handling the breakpoint exception in the kernel then generates a `SIGTRAP` signal to the user space process. Thus, for every breakpoint hit during execution, the application issuing a breakpoint incurs a context switch to the interrupt routine in kernel space and another context switch back to user space.

Applications can leverage interfaces provided by the operating system to debug child processes through breakpoints (e.g., `ptrace` for Linux). These make it easy to trigger the execution of specific instructions once a breakpoint is hit. While such interfaces are convenient, they incur significant overhead (up to 10 000 cycles in our system) due to the multiple context switches between processes. It is, therefore, crucial to work around this performance limitation for high-performance instrumentation.

One such high-performance application of breakpoint-based instrumentation is binary-level fuzzing. Fuzzing is one of the most prominent techniques to find memory-corruption vulnerabilities [132, 241, 289, 400]. It uses coverage-guided feedback to discover random inputs that exercise different paths of a program to expose bugs. While source-level fuzzing involves custom compilation passes to insert coverage collection instrumentation, binary-level fuzzing usually relies on binary instrumentation [372] or breakpoint-based instrumentation [129, 244, 272, 406].

Breakpoint-based instrumentation relies on aggressive optimizations to mitigate the performance hit of software breakpoints, e.g., removing the breakpoint completely once it has first been hit and an input reaching that coverage point has been collected [129]. Such an optimization eventually converges to zero performance overhead but sacrifices useful path information on branches that have already been hit [137].

```
.org 0x7c00
.patch 0xc40 # icebp entry point
.entry 0

let [cov_map] := tmp1
let [rip] := tmp0

# load address of coverage map from staging buffer
[cov_map]:= LDSTGBUF_DSZ64_ASZ16_SC1(0xba00)

# get instruction pointer low bits
[rip]:= ZEROEXT_DSZ64(IMM_MACRO_ALIAS_RIP) !m0
[rip]:= AND_DSZ64(0xffff, [rip])

# set coverage for basic block
STADPPHYS_DSZ8_ASZ64_SC1([cov_map], [rip], 0x01)
```

Listing 7.5. μsoftware breakpoint to collect code coverage.

7.6.2 Implementation

We leverage CustomProcessingUnit to implement a new type of software breakpoints that we name *μsoftware breakpoints*. To implement these, we change the semantics of the `icebp`/`int1` (`0xf1`) instruction, which should not be used by normal software. Placing the breakpoint logic directly in microcode avoids the cost of interrupts or context switches and is thus extremely fast.

As a proof of concept, we implement μsoftware breakpoints for coverage collection in binary-level fuzzing. We save the address of the coverage map in an internal buffer of the CPU for ease of access inside microcode and simply update the coverage based on the current instruction pointer during breakpoint execution. Listing 7.5 shows an example implementation of μsoftware breakpoints for coverage-guided fuzzing.

As a microbenchmark, we measure the overhead of executing a single μsoftware breakpoint to collect coverage information, averaging 1 million executions. For each μsoftware breakpoint, the CPU has a latency of 10 cycles, which is mostly due to the switch to the microcode RAM by the Microcode Sequencer. This is around 1000x faster than `ptrace`-based instrumentation. To compare with the fastest non-microcoded implementation, we also implement the same logic for coverage collection directly in the kernel debug exception handler. While such an implementation is faster than user space coverage collection, we still measure a latency of 388 cycles, making μsoftware breakpoints 38.8x faster.

7.7 Case Study: Constant-time Hardware Division

7.7.1 Background

Side-channel attacks allow adversaries to leak secret values by observing secret-dependent side effects of computation, such as differences in execution time or microarchitectural state [31, 236]. Constant-time programming aims to produce algorithms resistant to timing side-channel attacks, implemented so that the same

```
.org 0x7c00
.patch 0x6c8 # div entry point
.entry 0

let [dividend] := rax;              let [temp1]    := tmp3
let [divisor]  := rcx;              let [temp2]    := tmp4
let [size]     := 0x3f;             let [temp3]    := tmp5
let [quotient] := tmp0;             let [temp4]    := tmp7
let [temp]     := tmp1;             let [temp5]    := tmp8
let [i]        := tmp2;             let [comp]     := tmp6
[temp] := ZEROEXT_DSZ64(0x0);       [i] := ZEROEXT_DSZ64([size])
[quotient] := ZEROEXT_DSZ64(0x0)

<loop>
# if (i < 0 ) goto end;
UJMPCC_DIRECT_NOTTAKEN_CONDB([i], <end>)

# temp = (temp << 1uLL) | ((dividend >> i) & 1);
[temp1]:= SHL_DSZ64([temp], 0x1)
[temp2]:= SHR_DSZ64([dividend], [i])
[temp2]:= AND_DSZ64([temp2], 0x1)
[temp] := OR_DSZ64([temp1], [temp2])

# comp = (temp >= divisor);
[comp]  := SUB_DSZ64([divisor], [temp])

# temp -= comp? divisor : 0;
[temp3]:= SELECTCC_DSZ64_CONDB([comp], [divisor])
[temp] := SUB_DSZ64([temp3], [temp])

# quotient |= comp ? 1uLL << i : 0;
[temp4]:= SHL_DSZ64(0x1, [i])
[temp5]:= SELECTCC_DSZ64_CONDB([comp], [temp4])
[quotient] := OR_DSZ64([quotient], [temp5])

# i--; goto loop
[i] := SUB_DSZ64(0x1, [i]) SEQW GOTO <loop>

<end>
# return quotient, ignore the remainder for simplicity
rax := ZEROEXT_DSZ64([quotient])
rdx := ZEROEXT_DSZ64(0x0)
```

Listing 7.6. Constant-time `div` microcode routine.

instruction and memory access patterns occur regardless of the secret input [31, 185]. Several solutions have been proposed to automatically rewrite software to be constant-time [48, 65, 301, 332, 389]. They typically involve transforming programs during compilation to consistently execute the same sequence of operations irrespective of their input.

While this is effective for instruction traces and memory access patterns, there are some instructions whose latency depends on the input values [189]. Examples of such instructions are division or remainder operations and many floating-point operations. Executing these instructions on secret data may leak information about their operands or the result. Automated solutions to mitigate such side channels rely on software wrappers implementing these operations in constant-time. However, substituting a single instruction with a full software wrapper incurs substantial overhead. It increases the code size and requires complete recompilation of the

program or precise binary patching.

7.7.2 Implementation

As a case study, we patch one of these instructions, unsigned integer division (`div`), to provide constant-time guarantees at the microcode level instead. Intel CPUs implement division directly in hardware [62], but we verified that the `div` instruction itself is microcoded and thus can be patched. The microcode of `div` simply calls the μops that control hardware operations to perform the division, taking between 22 and 40 cycles to execute.

We take the state-of-the-art 64-bit constant-time division software implementation from Constantine [48] and reimplement it in microcode. The Constantine software implementation takes 694 cycles. Our microcode implementation (depicted in Listing 7.6) takes just 438 cycles to execute, 1.58x faster than in software. This speedup is thanks to the reduced number of fetched and decoded instructions, higher instruction cache locality, reduced register pressure, and faster microcode jumps (that have control over the branch predictors).

This has the further advantage of not requiring any transformation of the input program. Once the patch is installed, any `div` instruction executed is constant-time, and it can be enabled and disabled as needed to prevent unnecessary overhead when timing guarantees are not required.

7.8 Conclusion

In this chapter, we presented a static and dynamic analysis framework for reverse engineering of Intel x86 microcode for the Goldmont microarchitecture. We demonstrated our framework's utility and the potential security and performance benefits of microcode customization in four case studies.

Our framework builds upon research reverse engineering Intel's debug infrastructure [126]. Debug infrastructure is crucial for post-silicon validation, but its security should rely on transparent mechanisms rather than on security by obscurity. Given our increasing reliance on critical digital infrastructure, both manufacturer documentation [186] and reverse engineering efforts will play an important role in ensuring the underlying hardware is secure.

This chapter ends our journey on the security of modern systems. We provided an in-depth understanding of several system layers that may be the target of an attacker compromising a secure system. We followed the path from the most shallow software application layer to the deepest CPU abstractions, sequentially dropping assumptions on the attacker capabilities and proposing solutions to improve the security of several layers. This chapter represents the lowest level reachable while examining software security (*i.e.*, CPU μcode) before reaching the hardware level: a different journey worth its own story.

Chapter 8

Conclusions and Future Work

In this thesis, we improve several existing techniques and propose new ones to find and mitigate complex vulnerabilities in secure systems. We take a comprehensive approach to address over a hundred bugs in several layers of modern systems, from application software to CPUs.

In Chapter 2, we propose `raindrop`: a novel approach to obfuscate binaries using code reuse techniques. This provides strong protection against manual and automatic reverse engineering attacks. We show how our technique could withstand state-of-the-art deobfuscation attempts, on par or superior to alternative techniques. An interesting evolution of this approach would be to study how to combine code reuse–based obfuscation with state-of-the-art techniques like virtualization (VM) obfuscation. Our technique can be naturally applied on top of VM obfuscation to hide the logic of VM handlers. However, both techniques can be fine-tuned to take advantage of their combination, for example, by emitting handlers more prone to be translated to ropchains or by designing specific ropchain-based obfuscations on the VM bytecode.

Chapter 3 presents a novel approach to improve the bug-finding capabilities of modern fuzzers by enhancing the context sensitivity during explorations. Specifically, we move away from an all-or-nothing approach by favoring context sensitivity in program regions with high dataflow diversity. We show how this technique improves over the state of the art. In future work, we plan to study how other different prioritization schemes may affect the predictive context sensitivity, other than dataflow diversity. Although we show how dataflow diversity improves bug finding when used to select context priority, there may be better choices for different classes of applications, and more effective strategies may exist.

In Chapter 4, we propose UNCONTAINED: a novel sanitizer and static analysis approach to find subtle type confusion bugs in C polymorphic code. Our dynamic approach leverages the peculiarities of structure embedding to precisely infer the runtime destination type for a downcasting operation by looking at redzones. In future work, we may leverage a similar approach to broaden the type of bugs our sanitizer can detect by spotting generic type confusion bugs other than container confusion. This would need an in-depth study to understand whether redzones are a good enough indicator to infer the actual destination type for generic type cast operations when not involving structure embedding. Our static analysis approach

recognizes patterns of incorrect downcasting. While having a limited false positive rate, we may decrease it in future work by adding more heuristics. Additionally, we plan to extend the heuristics to find more classes of container confusion bugs, generalizing over the simple, yet effective, bug classes we defined.

Chapter 5 presents CONSTANTINE: a compilation framework to mitigate microarchitectural side channels automatically. It pursues a radical approach to linearize both the control and data flow. Our framework provides strong protection against time- and contention-based side channels, with a low overhead for various cryptographic implementations. In future work, we plan to explore how the framework can be extended to transient execution attacks to mitigate speculative leaks. This needs us to rethink how the framework detects vulnerable program portions and how it mitigates them. To this end, we would need to design a taint analysis that precisely models speculative execution and its effects. Additionally, we would need to linearize any branch or access affected by speculative conditions and design adequate protection that would not be bypassed by speculative execution.

In Chapter 6, we extensively study architectural and transient-execution vulnerabilities and their relationship. We show how they share the same types of root causes and how they can be classified according to software vulnerabilities root causes. By examining blank spots in our classification, we find ÆPIC Leak: a novel architectural bug on modern CPUs, able to leak sensitive data without side channels. We show the unique properties of our attack, which does not require side channels, hyperthreading, or transient execution while being able to control which data is leaked. In future work, we will continue looking for CPU bugs by trying to fill empty spots in our classification (*i.e.*, , Architectural Confused Deputy - CWE-441), finding novel categories for CPU bugs, or finding different bugs in existing categories.

Finally, Chapter 7, presents CustomProcessigUnit: the first public framework to statically and dynamically inspect, trace and analyze CPU microcode. We describe several case studies to show how our framework can be highly effective in understanding CPU internals and enhancing CPU features for performance and security. We plan to extend the work by leveraging the framework to find novel CPU bugs: it offers unprecedented capabilities to inspect CPU state while executing, possibly allowing us to observe transient execution effects deeper than ever. Additionally, it allows research on undocumented CPU features, like finding undocumented instructions and analyzing undocumented Model Specific Registers.

Our work offers comprehensive contributions to the security of several layers of secure systems. We show how all our frameworks can be leveraged to find or mitigate a wide range of security vulnerabilities. By releasing all our software prototypes as open source, we hope to foster future security research.

Appendix A

Scientific Publications

Besides the publications that are directly included in this thesis, the author was involved in multiple publications. We report the full list, in chronological order, both as main author and non-main author, at the time of writing.

- M. Angelini, G. Blasilli, P. Borrello*, E. Coppa, D.C. D'Elia, S. Ferracci, S. Lenti, G. Santucci. "Ropmate: Visually Assisting the Creation of ROP-Based Exploits". In the *IEEE Symposium on Visualization for Cyber Security* (VizSec), 2018.
  **Main author, alphabetical order.*

- P. Borrello, E. Coppa, D.C. D'Elia, C. Demetrescu. "The Rop Needle: Hiding Trigger-Based Injection Vectors via Code Reuse". In the *ACM/SIGAPP Symposium On Applied Computing* (SAC), 2019.

- P. Borrello, E. Coppa, and D. C. D'Elia. "Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation". In the *Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), 2021.

- P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida. "Constantine: Automatic side-channel resistance using efficient control and data flow linearization.". In the *ACM SIGSAC Conference on Computer and Communications Security* (CCS), 2021.

- M. Schwarzl, P. Borrello, A. Kogler, T. Schuster, K. Varda, D. Gruss, M. Schwarz. Robust and Scalable Process Isolation against Spectre in the Cloud. In the *European Symposium on Research in Computer Security* (ESORICS), 2022.

- P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In the *USENIX Security Symposium*, 2022.

- M. Schwarzl, P. Borrello, G Saileshwar, H. Mueller, D. Gruss, M. Schwarz."Practical Timing Side-Channel Attacks on Memory Compression". In the *IEEE Symposium on Security and Privacy* (S&P), 2023.

- P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, M. Schwarz. "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode". *IEEE Workshop on Offensive Technologies* (WOOT), 2023.

- J. Koschel*, P. Borrello*, D.C. D'Elia, H. Bos, C. Giuffrida. "UNCONTAINED: Uncovering Container Confusion in the Linux Kernel". *USENIX Security Symposium*, 2023.
  **Equal contribution joint first authors.*

- P. Borrello, A. Fioraldi, D.C. D'Elia, D. Balzarotti, L. Querzoni, C. Giuffrida. "Predictive Context-sensitive Fuzzing". *Under review*, 2023.

Appendix B

Practical Contributions

Table B.1 lists the open source software we developed. Table B.2 lists the bugs we reported to the vendors (and their corresponding CVEs if applicable) while working on the main contributions of this thesis.

Table B.1. Open source software developed during the course of this thesis. The entries marked with * will be made available to the community after the thesis publication.

Chapter 2	
`raindrop`	https://github.com/pietroborrello/raindrop
Chapter 3	
Predictive Ctx-Sensitive Fuzzing*	https://github.com/pietroborrello/predictive-ctx-fuzzing
Chapter 4	
UNCONTAINED Sanitizer*	https://github.com/vusec/uncontained-sanitizer
UNCONTAINED Dataflow Analysis*	https://github.com/vusec/uncontained-dataflow
Kernel Tools	https://github.com/Jakob-Koschel/kernel-tools
Chapter 5	
CONSTANTINE	https://github.com/pietroborrello/constantine
Chapter 6	
ÆPIC Leak	https://github.com/IAIK/AEPIC
Chapter 7	
CustomProcessingUnit	https://github.com/pietroborrello/CustomProcessingUnit
μcode Decompiler	https://github.com/pietroborrello/ghidra-atom-microcode

Table B.2. Bugs we reported while working on the main contributions of this thesis, with their assigned CVEs, if any.

Project (n° Bugs)	**CVE List**
`ffmpeg` (1)	CVE-2022-1475
`njs` (1)	CVE-2022-28049
`stb` (4)	CVE-2022-28041, CVE-2022-28042, CVE-2022-28048
`libhevc` (1)	
`matio` (1)	CVE-2022-1515
`exiv2` (6)	
`harfbuzz` (1)	CVE-2022-33068
`httplib` (1)	
`lrzip` (3)	CVE-2022-28044, CVE-2022-33067
`protobuf-c` (2)	CVE-2022-33070
`solidity` (2)	CVE-2022-33069
Linux kernel (89)	CVE-2023-25012, CVE-2023-1073, CVE-2023-1074, CVE-2023-1075, CVE-2023-1076, CVE-2023-1077, CVE-2023-1078, CVE-2023-1079, *others currently being assigned*
`wolfSSL` (1)	CVE-2020-11713
Intel Sunny Cove CPUs (1)	CVE-2022-21233

Appendix C

Chapter 2: Additional Material

We report practical details of the rewriter implementation, and additional settings and findings from the evaluation.

Table C.1. N:= number of program points (instructions), A:= total number of gadgets used across all ROP chains; B:= number of unique gadgets used across all ROP chains; C:= average number of gadgets used per program point.

Benchmark	N	$ROP_{0.00}$			$ROP_{0.05}$			$ROP_{0.25}$			$ROP_{0.50}$			$ROP_{0.75}$			$ROP_{1.00}$		
		A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
b-trees	170	583	144	3.43	966	322	5.68	1513	555	8.90	2090	719	12.29	2734	856	16.08	3411	940	20.06
fannkuch	89	193	104	2.17	471	199	5.29	798	377	8.97	1085	491	12.19	1310	589	14.72	1608	644	18.07
fasta	256	733	226	2.86	1372	431	5.36	2135	733	8.34	3015	919	11.78	3830	1025	14.96	4717	1124	18.43
fasta-redux	263	670	246	2.55	1313	499	4.99	1775	676	6.75	2842	950	10.81	3914	1105	14.88	4727	1163	17.97
mandelbrot	135	323	143	2.39	793	316	5.87	1047	419	7.76	1598	539	11.84	2341	642	17.34	2666	659	19.75
n-body	288	680	260	2.36	1443	535	5.01	2026	747	7.03	3259	1003	11.32	4444	1137	15.43	5487	1220	19.05
pidigits	144	462	120	3.21	814	285	5.65	1189	457	8.26	1769	621	12.28	2211	710	15.35	2883	803	20.02
regex-redux	162	522	147	3.22	869	285	5.36	1437	546	8.87	1946	713	12.01	2539	844	15.67	3268	935	20.17
rev-comp	176	558	174	3.17	1265	364	7.19	1770	605	10.06	2322	720	13.19	2810	838	15.97	3367	924	19.13
sp-norm	115	329	119	2.86	529	195	4.60	845	381	7.35	1361	554	11.83	1761	660	15.31	2234	737	19.43
Avg / Geo. Mean	179.80	505.30	168.30	2.79	983.50	343.10	5.46	1453.50	549.60	8.17	2128.70	722.90	11.94	2789.40	840.60	15.55	3436.80	914.90	19.19

C.1 Implementation Aspects

C.1.1 Switch Tables

As compilers recur to indirect jumps to efficiently implement switch constructs, we use CFG reconstruction heuristics to reveal possible targets, and then use the original target calculation sequence used for the dispatching to our advantage. Instead of jumping to the location corresponding to the desired code block in the original program, at rewriting time we store at such location the RSP displacement in the chain for the corresponding code block, then during execution we use the address computed by the dispatching sequence to read the correct offset. Thus when lowering, e.g., a `jmp reg` the rewriter can combine gadgets to achieve:

```
movsx reg, byte ptr [reg]  ## offset for RSP
shl reg, 0x3
add rsp, reg
```

where `reg` contains the jump target address for the original program. In the end, the main difference in our ROP encoding between a direct jump and an indirect jump is

that the former retrieves the offset to add to RSP from the chain (Section 2.4.2.2), while the latter retrieves it from the location of the original code block. We use offsets of 1, 2, o 4 bytes depending on the position of the desired block in the chain from the jump site (the example uses a 1-byte value).

C.1.2 From Native to ROP and Back

Upon generation of a ROP chain the rewriter replaces the original function implementation in the program with a stub that switches the stack and activates the chain (Section 2.4.2.3). The pivoting stub performs three steps: (a) it reserves a new entry for storing `other_rsp` in the stack-switching array `ss`, (b) it saves RSP in `other_rsp`, (c) it loads the address of the chain (`chain_address`) for the function to execute into RSP and starts the chain execution. A pivoting stub can be implemented in 22 bytes as:

```
push ss
pop rax
add qword ptr [rax], 0x8
add rax, qword ptr [rax]    ## step (a) ends
mov qword ptr [rax], rsp    ## step (b)
push chain_address          ## step (c)
pop rsp
ret
```

This code is optimized to use only a single caller-save register (e.g., `rax`) and uses `push-pop` sequences in place of `mov` instructions to minimize the number of bytes required to encode the sequence.

When a chain reaches its epilogue and a native function has to be resumed, a symmetric *unpivoting* sequence takes care of removing the entry created for storing `other_rsp` for the active chain and restores the native stack pointer into RSP.

The implementation should realize e.g.:

```
pop r11                     ## ss
sub qword ptr [r11], 0x8
add r11, qword ptr [r11]
add r11, 0x8
mov rsp, qword ptr [r11]
```

Notice that only register `r11` will be clobbered by this sequence. Also, while the pivoting sequence is made of native instructions, the sequence above will be realized by gadgets.

C.1.3 Tail Jumps

To handle optimized tail jump instances the rewriter uses an unpivoting variant that, instead of returning to the calling function, jumps to the target of the tail jump:

```
pop r11                     ## ss
sub qword ptr [r11], 0x8
```

```
add r11, qword ptr [r11]
add r11, 0x8
pop rax                      ## jmp target
mov rsp, qword ptr [r11]; jmp rax
```

The last line represents a JOP gadget.

C.2 Evaluation Additions

C.2.1 Functions for Obfuscation Resilience Experiments

To produce the 72 hash functions for the G_1 tests of Section 2.7.2 we used the *RandomFuns* feature of Tigress with a command line:

```
tigress --Verbosity=1 --Seed={seed}
    --Environment=x86_64:Linux:Gcc:6.3.0
    --Transform=RandomFuns
        --RandomFunsName=target
        --RandomFunsTrace=0
        --RandomFunsType={data_type}
        --RandomFunsInputSize=1
        --RandomFunsLocalStaticStateSize=1
        --RandomFunsGlobalStaticStateSize=0
        --RandomFunsLocalDynamicStateSize=1
        --RandomFunsGlobalDynamicStateSize=0
        --RandomFunsBoolSize=3
        --RandomFunsLoopSize=25
        --RandomFunsCodeSize=1000
        --RandomFunsOutputSize=1
        --RandomFunsControlStructures={control}
        --RandomFunsPointTest=true
    --out={output_file} {base_path}/empty.c
```

and different combinations for the parameters `control`, `seed`, and `data_type`. Table C.2 lists the values we used for `control`; `seed` was from $\{1, 2, 3\}$, while `data_type` was `char`, `short`, `int`, or `long`.

For the *code coverage* scenario we used the same sets of parameters and altered the command in the following way: we set `Random FunsPointTest` to `false` to disable the secret value checking step, and `RandomFunsTrace` to `2` to annotate CFG split and join points.

C.2.2 VM Obfuscation

To apply during our experiments one or more layers of VM obfuscation (with or without implicit VPC loads) to a piece of code we used Tigress with the following parameters:

Table C.2. Values used for the `RandomFuns ControlStructures` parameter in Tigress to generate the 72 functions for Section 2.7.2.

`RandomFunsControlStructures` Parameter Value	Ctrl-flow depth	Num. of *if*-stmts	Num. of Loops
(if (bb 4) (bb 4))	1	1	0
(for (if (bb 4) (bb 4)))	2	1	1
(for (for (bb 4)))	2	0	2
(for (for (if (bb 4) (bb 4))))	3	1	2
(for (if (if (bb 4) (bb 4)) (if (bb 4) (bb 4))))	3	3	1
(if (if (if (bb 4) (bb 4)) (if (bb 4) (bb 4))) (if (bb 4) (bb 4)))	3	5	0

```
tigress --Environment=x86_64:Linux:Gcc:6.3.0
    --Transform=InitOpaque --Functions=main
        --Transform=InitImplicitFlow --Functions=main
        --InitImplicitFlowHandlerCount=0
        --InitImplicitFlowKinds=counter_int, \
        counter_float,bitcopy_unrolled,bitcopy_loop
    [transformations]
    --out={output_file} {input_file}
```

Where `[transformations]` is obtained by adding for each VM layer on `function` the following parameters:

```
    --Transform=Virtualize
        --VirtualizeDispatch={[call, switch]}
        --VirtualizeImplicitFlowPC={[PCUpdate,none]}
        --VirtualizeOpaqueStructs=array
        --Functions={function}
```

Following the documentation of Tigress, we alternate the *call* and *switch* methods for VPC dispatching across nested virtualization layers. In our tests with S2E, however, the engine did not seem evidently affected by the choice of a particular dispatching method over another (including the additional schemes supported by Tigress to this end).

C.2.3 Additional Deployability Experiments

Table C.1 reports several statistics collected by the rewriter during the translation of the 10 benchmarks from the `shootout` suite considered in Section 2.7.3. In particular, the table provides for different settings of ROP_k the following kinds of information: the number of program points (i.e., instructions) N that were obfuscated (not affected by k), the total number A of gadgets used by the chains from all the obfuscated functions, the number B of unique gadgets used across all chains, and the average number C of gadgets used per obfuscated program point.

A, B, and C increase as we add more instance of P3 with higher k values. The increase in the number B of used unique gadgets shall be interpreted as an indication

that the rewriter does not use a fixed set of gadgets to instantiate P3 (but can draw from multiple equivalent versions of a desired gadget functionality, bringing diversity) and that possibly different dead and symbolic registers get involved at different program points. At the same time, the rewriter performs gadget reuse across different chains: for instance, for $k = 1.00$ the rewriter uses the largest number of gadgets per obfuscated instruction (geometric mean of 19.19), but the ratio between the average values for A and B is $\sim$3.75, showing that a gadget is on average reused almost four times across all the chains.

Appendix D

Chapter 3: Additional Material

D.1 Additional Charts and Tables

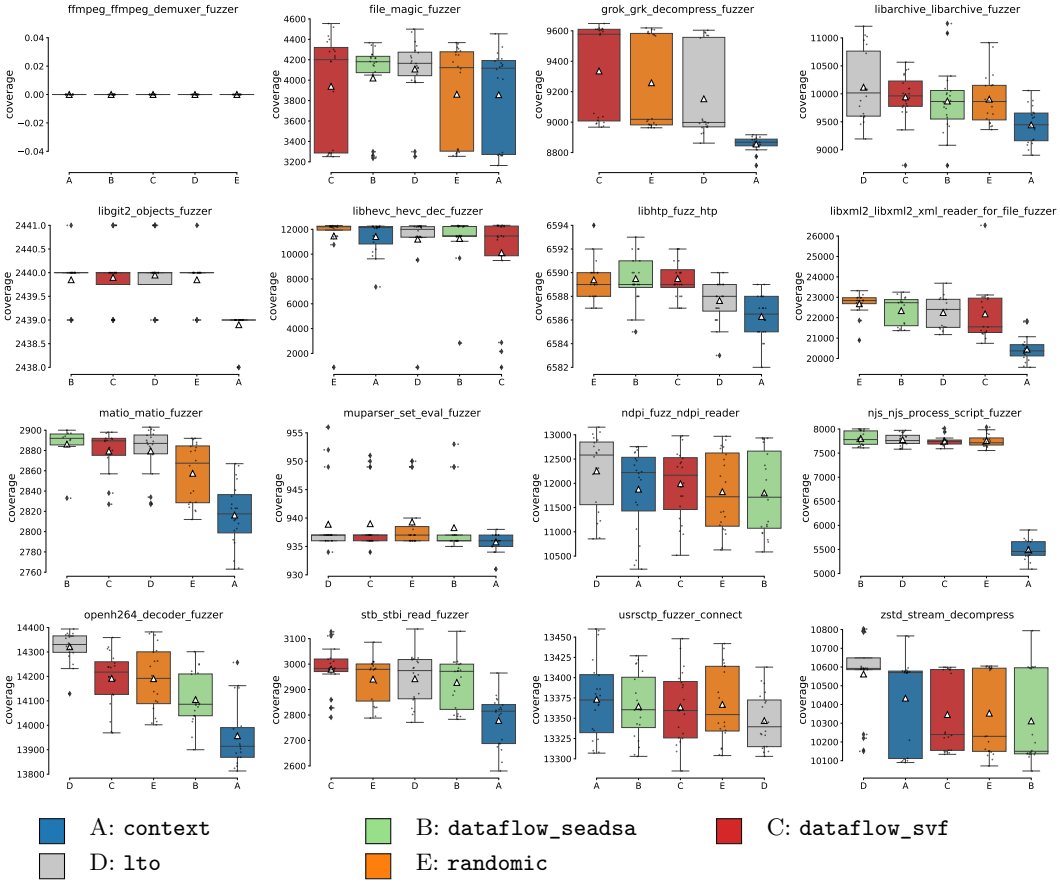
Table D.1 complements the boxplots from Figure 3.2 for the bugs found in each of the 20 runs on the FuzzBench subjects by presenting the total number of unique bugs encountered across all runs. For the sake of readability, the table also lists the language(s) in which each program is written (Table 3.1).

Figure D.1 reports complete coverage results for the FuzzBench programs mentioned in Section 3.6.3. Our fuzzers obtain coverage close to `lto` and higher than it

Table D.1. Unique bugs by benchmark (cf. RQ2 of Section 3.6.2)

Benchmark	Type	context	seadsa	svf	lto	randomic
`ffmpeg`	C, some C++	6	8	11	10	7
`file`	C, some C++	3	4	3	1	2
`grok`	C++	2	-	7	6	7
`libarchive`	C	0	0	0	2	0
`libgit2`	C	3	3	3	3	3
`libhevc`	C	2	2	1	2	1
`libhtp`	C++	5	5	5	6	5
`libxml2`	C	3	15	22	16	12
`matio`	C	43	33	26	26	34
`muparser`	C++	0	1	1	0	1
`ndpi`	C	12	15	17	18	13
`njs`	C	0	1	1	1	1
`openh264`	C++	7	8	8	8	8
`stb`	C/C++	15	13	18	11	15
`usrsctp`	C	0	0	0	0	0
`zstd`	C/C++	1	2	2	2	1
Total (16)	All	102	110	125	112	110
	C only (8)	63	69	70	68	64
	C++ only (4)	14	14	21	20	21
	Mixed (4)	25	27	34	24	25

Figure D.1. Boxplots for FuzzBench programs for the median number of discovered control-flow edges on 20 trials. Fuzzers are ordered at each benchmark by the median number of edges covered. Each boxplot includes the mean value (△) and raw data points (·). The coverage of `ffmpeg` is zero due to a bug of the Fuzzbench coverage measurer.



in at least one variant (typically `svf`) on all subjects, with the exception of `openh264` and `ztd`). At the time of writing, we could not obtain boxplots for `fmpeg` due to a bug in the coverage calculation tooling of FuzzBench that we informally reported to its maintainers. *We will provide the chart in a camera-ready version if accepted, building it manually if necessary.*

The CVE identifiers for the security issues in the FuzzBench programs mentioned in Section 3.6.4 are the following: CVE-2022-28041, CVE-2022-28042, and CVE-2022-28048 for `stb`, CVE-2022-1475 for `ffmpeg`, CVE-2022-1515 for `matio`, and CVE-2022-28049 for `njs`. As we mentioned in the chapter, the bugs involved in the last two issues were found also by the `lto` fuzzer. Finally, for the bugs found for the 7 additional case studies: CVE-2022-28044 and CVE-2022-33067 for `lrzip`, CVE-2022-33068 for `harfbuzz`, CVE-2022-33070 for `protobuf-c`, and CVE-2022-33069 for `solidity`.

D.2 A Fast Intra-procedural Alternative

To obtain the points-to sets for object arguments at call sites, we explored both union-based and inclusion-based pointer analyses (Section 3.2.2), which both track objects at inter-procedural level. Inclusion-based approaches typically offer more accurate results at the price of higher computational costs. Union-based approaches are faster but less accurate, despite a context-sensitive version can distinguish local aliasing created at different call sites of the enclosing function [220]. The analysis of real-world programs may be expensive even with state-of-the-art pointer analyses and, for inclusion-based solutions, possibly untenable for large targets [220, 286]. However, as it became apparent in our evaluation in Section 3.6.2, the precision of the pointer analysis may affect the effectiveness of our data-flow based cloning policy. While in our tests the one-time costs incurred during compilation were tenable for all subjects, one can think of scenarios, such as continuous integration test systems, that may benefit from smaller costs. Therefore, we designed a cheap object analysis that can complete fast and scale to arbitrary program sizes thanks to its intra-procedural nature. We then assessed how well it can assist our predictive approach in bug finding tasks.

Scalable Object Analysis. Our approach builds on the concept of *def-use* chains [16], which relate variable definitions to the program points that may use them before any intervening redefinition. Algorithm 4 depicts its workflow: in short, the analysis works as a backward slicing with a symbolization of memory allocation sites. Starting from the values on which a pointer depends on, it identifies the abstract objects as the allocation sites from where the pointer may have originated. An allocation site can be a stack location, a global variable, an incoming argument for the function, or dynamically allocated storage (for simplicity, we consider as such any invoked function having a pointer as return value). The analysis incurs a worst-case complexity that is linear in the number of instructions in the function and is trivially flow- and context-insensitive.

Bug finding capabilities. Our custom object analysis took part in the experiments described in Section 3.6.2 and 3.6.4. Figure D.2 reports the main results from our experiments. In the following, with `light` we will refer to the predictive fuzzer configured to use information from our lightweight object analysis to study the data-flows at call sites. Interestingly, `light` achieves respectable performance compared to the fuzzers based on the two inter-procedural, state-of-the-art pointer analyses of SVF and SeaDsa. On the FuzzBench programs of RQ2, `light` is able to find 102 of the 125 unique bugs that the `svf` fuzzer found and 93 of the 110 bugs that the `seadsa` fuzzer found. Moreover, `light` found 8 bugs that both the others missed, with a very good performance on `matio` where it peaked with 35 bugs (2 more than `seadsa` and 9 more than `svf`). Our hypothesis is that, since global allocation sites see a coarse-grained intra-procedural treatment, `light` gives less attention to rich inter-procedural data-flows from "hard" object-oriented benchmarks like `matio` (Section 3.6.2), favoring local flows that resulted in turn into more profitable cloning decisions. The total number of bugs found by `light` is a promising 117. The analysis is effective also on the additional C++ benchmarks of RQ4.

Algorithm 4: Lightweight Intra-procedural Object Analysis

```
function GetPointsToSet(ptr)
    result ← { }
    queue ← { ptr }
    while stmt ← queue.pop() do
        if Visited(stmt) then continue
        MarkAsVisited(stmt)
        if IsAllocationSite(stmt) then
            result ← result ∪ { stmt }
        else
            for operand ∈ stmt do
                queue.push(operand)
    return result
```

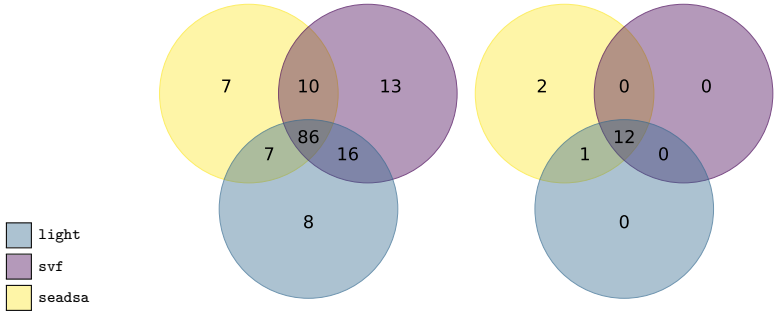


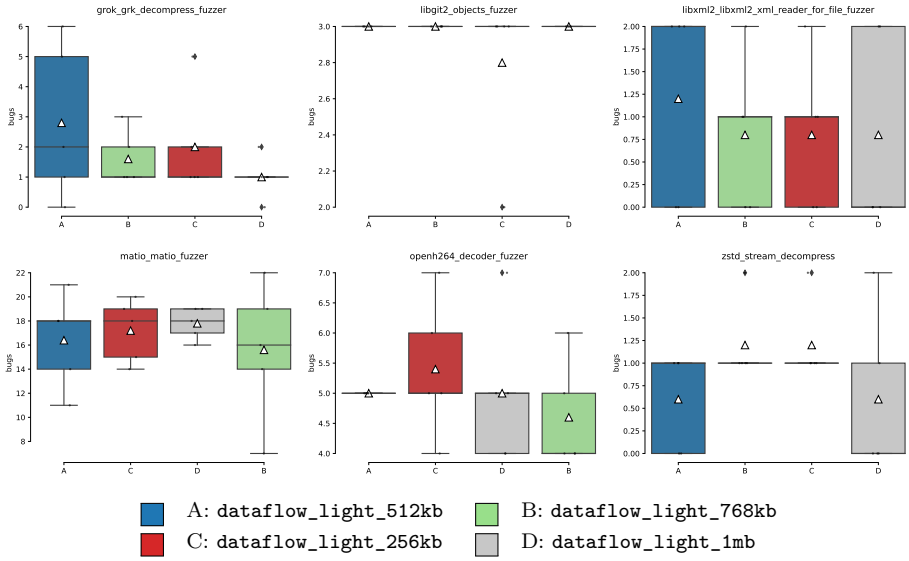
Figure D.2. Venn diagrams for unique bugs found by the fuzzers on the FuzzBench (left) and the 7 additional C++ (right) programs.

D.3 Cloning Budget

For the fine-tuning of our approach, we conducted preliminary experiments involving different cloning budgets. As the choice of allowing up to 2^{18} entries in the resulting collision-free coverage map resulted in no evident internal wastage for our fuzzers and can easily be accommodated (with a L2 cache of $\geq$ 256 KB) by modern architectures, we focused our attention on larger budget sizes. Therefore, we looked for a possible sweet spot for additional context-sensitivity when using higher-end hardware for testing. As the FuzzBench cloud appliances report a L2 cache size of "only" 256 KB, we used a local server equipped with a 2.40GHz Intel Xeon Platinum 8260 CPU and 1 MB of L2 cache per core. A larger cache can more efficiently handle the coverage map updates executed at each instrumented program point (although processing the map contents for a testcase upon execution termination will still face a higher latency, as we mentioned in Section 3.3).

For our purposes, we selected 6 subjects from the FuzzBench benchmarks used in the experiments from the chapter with the following characteristics: they should entail a number of potential context-sensitive edge instances higher than 2^{20} (so that some program portions would still be left context-insensitive when filling the 1-MB L2 budget), whereas the edges to instrument before any cloning should be fewer than 2^{18} (so that a 256-KB budget for the coverage map would still allow for cloning opportunities). For each benchmark, we executed 5 trials of 24 hours each using the `light` variant of our predictive approach (we chose `light` over the other

Figure D.3. Boxplots for cloning budget experiment showing the median number of uncovered bugs for each fuzzer on 5 trials. Fuzzers are ordered by median number of bugs.



variants for the sake of faster compilation time only).

Figure D.3 shows the boxplots with the median number of bugs for each benchmarks, while Table D.2 reports the average normalized score. The sweet spot for most of these benchmarks is 512 KB: a budget large enough to enable further effective predictions, but not excessively large as to cause evident internal wastage. More generally, machines that have larger L2 caches can in principle offer further opportunities to fuzzers.

However, there are at least two factors that can make larger maps detrimental to an uninformed use: the higher latency in processing large maps (at least for AFL-like fuzzers) upon execution termination and the risk of queue explosion from exploring uninteresting program points, where larger queues are often a symptom (we report the size of the queues for this experiment in Table D.3). The first aspect may require fine-tuning from a human expert depending on the program under analysis. For the second aspect, one may modify our prediction-driven cloning (Algorithm 1) before budget exhaustion whenever no call site features a data-flow diversity below some empirically determined threshold. On the other hand, in doing so, the (im)precision of the underlying pointer analysis may still lead to overlooking interesting call sites.

In light of the considerations above, the results presented in the chapter were obtained with a cloning budget of 256 KB as it performed remarkably well as off-the-shelf criterion in all our tests.

Table D.2. Average normalized score using median number of bugs from the cloning budget experiment.

Fuzzer	Average normalized score
`dataflow_light_512kb`	100.00
`dataflow_light_256kb`	83.33
`dataflow_light_768kb`	78.15
`dataflow_light_1mb`	58.33

Table D.3. Median queue size over the trials of the cloning budget experiment

Benchmark	256kb	512kb	768kb	1Mb
`grok`	2365	3061	2983	3388
`libgit2`	1513	1789	1767	1790
`libxml2`	5872	9031	8521	5850
`matio`	10040	10034	10527	10297
`openh264`	7819	9517	10474	11053
`zstd`	28847	41849	41691	40992
Geo-mean	5869	7376	7354	7820

Appendix E

Chapter 4: Additional Material

E.1 LMbench Evaluation

Table E.1 presents detailed results for the LMbench tests mentioned in Section 4.5.3.2.

E.2 Static Analysis Rules

Table E.2 shows the definitions for our static information flow analyses. For each pattern, we report the source where the dataflow starts from, the sinks that the dataflow searches, the path filters that inhibit the report (*i.e.*, stop path exploration) when met, and additional checks that the analysis performs at a sink before reporting a potential container confusion.

Table E.1. LMbench experiments: comparing the native execution baseline against UNCONTAINED and KASAN.

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

Table E.2. Details of rules for the patterns defined for static analysis. Showing the direction (B for backwards dataflow, F for forward dataflow), source and sink matched, and eventual filters and/or additional checks. ❺ employs a single rule in two parts.

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

Appendix F

Chapter 5: Additional Material

F.1 Conditional Selection

The `ct_select` primitive of Section 5.4.2 can be instantiated in different ways. We studied how the LLVM compiler optimizes different schemes for constant-time conditional selection to pick the best possible alternative(s) in CONSTANTINE.

For the discussion we consider the pointer selection primitive that we use to differentiate decoy and real store operations (i.e., to conditionally select whether we should actually modify memory contents), evaluating the alternatives listed below:

Scheme	C equivalent	*taken* values	CFL overhead
1	`asm cmov`	`{0;1}`	9.6x
2	`ptr = taken ? ptr : (void*)NULL`	`{0;1}`	7.5x
3	`ptr = (void*)((size_t)ptr & (-taken))`	`(size_t) {0;1}`	7.3x
4	`ptr = (void*)((size_t)ptr & taken)`	`{0;0xff..ff}`	7.7x
5	`ptr = (void*)((size_t)ptr * taken)`	`{0;1}`	7.2x

We assume to operate on a `void* ptr` pointer given as input to the `ct_store` primitive, and *taken* values being 1 on real paths and 0 on decoy ones unless otherwise stated. Instead of reporting end-to-end overheads, we mask the slowdown from DFL by configuring CFL to use a single shadow variable as in the solution of Coppens et al. [82], then we compute the relative slowdowns of our protected `mulmod` wolfSSL version (Section 5.7) for the different `ct_select` schemes, using the default non-CT implementation (W=1) as baseline.

Scheme 1 forces the backend to emit `cmov` instructions at the assembly level, similarly to predicated execution mechanism we discussed in Section 5.2. As this choice constrains the optimizer's decisions, it turns out to be the worst performing alternative just as expected. Scheme 2 is essentially an LLVM IR `select` statement around the *taken* indirection predicate, on which the compiler can reason about and optimize, then after IR-level optimizations the backend for most of its occurrences emits a `cmov` instruction as in scheme 1, testing the value of the *taken* variable for conditional assignment.

Thus, we investigated different alternatives that could avoid relying on condition flags that, besides, get clobbered in the process and may require frequent recomputation. While mask-based schemes 3 and 4 seemed at first the most promising avenues, it turned out that the additional operation needed either to generate the

taken mask from a boolean condition (scheme 3), or to maintain it at run-time and combining it with the boolean conditions coming from branch decisions (scheme 4), made these schemes suboptimal. Scheme 5 resulted in the most simple and most efficient one between the solutions we tested, producing the lowest overhead as it unleashes several arithmetic optimizations (e.g. peephole, global value numbering) at IR and backend optimization levels.

F.2 Striding

One of the key performance enablers that back our radical approach is the ability to stride over object fields efficiently. We thoroughly tested different possible implementations, and designed different solutions based on the size of the field that should be strode and the granularity λ at which the attacker could observe memory accesses. Several of these solutions leverage CPU SIMD Extensions: while we focused on AVX2 and AVX512 instructions for x86 architectures, the approach can easily be extended to other architectures supporting vectorization extensions (e.g., ARM SVE [341] , RISC-V "V" [305]). We group our solutions in three categories: simple object striding, vector `gather/scatter` operations, and vector bulk `load/stores`.

Simple Object Striding Given an access on pointer `ptr` over some field `f`, the most simple solution is to just access linearly `f` at every λ-th location. We perform each access using a striding pointer `s_ptr` at the offset `ptr` % λ of the λ-th location, so that while striding a field `s_ptr` will match the target pointer `ptr` exactly once, on the location where the memory access should happen. For load operations we conditionally maintain the value loaded from memory, propagating only the real result over the striding, while for store operation we load every value we access, conditionally updating it at the right location (Section 5.4.3.1). We report a simplified[1] snippet of a striding load operation for a `uint8_t`, where the conditional assignment is eventually realized e.g. using a `cmov` operation:

```
uint8_t ct_load(uint8_t* field, uint8_t* field_end, uint8_t* ptr) {
  // Default result
  uint8_t res = 0;
  // Get the offset of the pointer with respect to LAMBDA
  uint64_t target_lambda_off = ((uint64_t) ptr) & (LAMBDA-1);

  // Iterate over the possible offsets
  for(uint8_t* s_ptr = field; s_ptr < field_end; s_ptr += LAMBDA) {
    // Compute the current ptr
    uint8_t* curr_ptr = s_ptr + target_lambda_off;
    // Always load the value
    uint8_t _res = *(volatile uint8_t*)curr_ptr;
    // If curr_ptr matches ptr, select the value
    res = (curr_ptr == ptr)? _res : res;
  }
  return res;
}
```

[1] Additional, constant-time logic is present in the implementation to deal with corner cases, so to avoid striding outside the object if not aligned correctly in memory.

Vector gather/scatter Operations While for small fields the simple striding strategy presented above performs relatively well, larger sizes offer substantial room for improvement by leveraging SIMD extensions of commodity processors. Vectorization extensions offer instructions to gather (or scatter) multiple values in parallel from memory with a single operation. This allows us to design striding algorithms that touch in parallel N memory locations (up to 16 for x86 AVX512 on pointers that fit into 32 bits). Our algorithm maintains up to N indexes in parallel, from which to access memory from N different locations at the same time. We manage in parallel the multiple accessed values similarly to for simple object striding, with the N results being merged with an horizontal operation on the vector to produce a single value for loads.

Vector Bulk load/stores Vectorization extensions allow us to load multiple values from memory at once, but a `gather`/`scatter` operation is in general costly for the processor to deal with. Depending on the value of λ (e.g., with $\lambda = 1$ or $\lambda = 4$) most of the values could lie on the same cache line. Therefore we further optimized DFL with a third option which simply uses SIMD extensions to access a whole cache line (or half of it with `AVX2`) with a single operation, thus touching all the bytes in that line. In case of loads, the accessed vector gets conditionally propagated with constant-time operations based on the real address to be retrieved, while for stores it gets conditionally updated, and always written back.

Sizing Building on empirical measurements on Skylake X and Whiskey Lake microarchitectures, we came up with a simple decision procedure to choose the best possible handler by taking into account the *size* of the field to stride, and the granularity λ at which should be strode. Table F.1 reports the average number clock cycles we measured for a striding handler given different values of *size* and λ. We list only the values for load operations for brevity, as the store handler incur in similar effects. We computed the number of cycles needed for each handler to stride over an object as the average of 1000 executions of the same handler, each measured using `rdtscp` instructions. The SIMD based measurements are based on AVX2.

We can immediately notice how bulk loads are the most effective for small λ values, as they allow for accessing whole cache lines in a single instruction, so this is the default choice for such values. The situation is more complex for higher λ values. We speculate that the AVX set-up operation for the CPU, and for the management of parallel values which should be merged together to produce the result, are too expensive to be amortized by the few iterations required to stride small objects. Therefore in this case we choose the simple object striding strategy. However, for bigger objects the gather operation is the clear winner, resulting in the minimum overhead. The decision algorithm in pseudocode reads as:

```
if (LAMBDA < 16) select bulk
else if ((striding_size / LAMBDA) < 8) select simple
else select gather
```

Table F.1. Clock cycles for different load striding handlers.

handler	λ	size	cycles
simple	64	64	4
gather	64	64	17
bulk	64	64	9
simple	64	512	17
gather	64	512	17
bulk	64	512	26
simple	64	1024	34
gather	64	1024	25
bulk	64	1024	44
simple	4	64	34
gather	4	64	22
bulk	4	64	11
simple	4	512	298
gather	4	512	116
bulk	4	512	44
simple	4	1024	586
gather	4	1024	226
bulk	4	1024	101

F.3 Field Sensitivity

We improved the field-sensitivity of the Andersen points-to analysis of SVF in order to delay demotion to field-insensitivity and recover, partially or to a full extent, the intended object portions thanks to heuristic inspired by duck typing from programming language research. In short, our extension restricts the surface of the abstract object that can be dereferenced to further improve the field information precision. The extension is semantically sound for the programs we consider, and in most cases (90% for wolfSSL) could refine the SVF results up to the single desired field. We describe our extension using the following running example:

```
%struct.fp_int = type { i32, i32, [136 x i64] } ; size = 1096
%struct.ecc_point = type { [1 x %struct.fp_int],
                           [1 x %struct.fp_int],
                           [1 x %struct.fp_int] } ; size = 3288
%struct.ecc_key = type { i32, i32, i32, i32, %struct.ecc_set_type*,
                         i8*, %struct.ecc_point,
                         %struct.fp_int }; size = 4416
```

These datatype declarations in LLVM IR describe the `fp_int`, `ecc_point`, and `ecc_key` structures of wolfSSL. An expression `i{8, 32, 64}` denotes an integer type of the desired bit width. LLVM IR uses pointer expressions for load and store operation that come from a GEP (`GetElementPtr`) instruction such as `%p` below:

```
%v = <some %struct.fp_int object>
%p = getelementptr %struct.fp_int, %struct.fp_int %v, %i32 0, %i32 1
```

The syntax of a GEP instruction is as follows. The first argument is the type, the second is the base address for the computation, and the subsequent ones are indices

for operating on the elements of aggregate types (i.e. structures or arrays). The first index operates on the base address pointer, and any subsequent index would operate on the pointed-to expressions. Here `%p` takes the address of the second `i32` field of a `fp_int` structure. What happens with SVF is that it frequently reports a whole abstract object `ecc_key` in the points-to set for `%p`: this information if used as-is would require DFL to access all the 4416 object bytes during linearization.

Starting from this coarse-grained information, our technique identifies which portions of a large object could accommodate the pointer computation. In this simple example, we have that `ecc_key` can host one `fp_int` as outer member (last field), and three more through its `ecc_key` member (second-last). Hence we refine pointer metadata to set of each second `i32` field from these objects, and only four 4-byte locations now require access during DFL. In general, we follow the flow of pointer value computations and determine object portions suitability for such dereferencing as in duck typing.

F.4 Recursion and Thread Safety

Our implementation is lackluster in two respects that we could address with limited implementation effort, which we leave to future work as the programs we analyzed did not exercise them.

The first concerns handling recursive constructs. Direct recursion is straightforward: we may predict its maximal depth with profiling and apply the just-in-time linearization scheme seen for loops, padding recursive sequences with decoy calls for depths shorter than the prevision, using a global counter to track depths. For indirect recursion, we may start by identifying the functions involved in the sequence, as they would form a strongly connected component on the graph. Then we may apply standard compiler construction techniques, specifically the inlining approach of [397] to convert indirect recursion in direct recursion, and apply the just-in-time linearization scheme discussed above.

The second concerns multi-threading. As observed by the authors of Raccoon [301], programs must be free of data races for sensitive operations. The linearized code presently produced by Constantine is not re-entrant because of stack variable promotion (Section 5.4.3.3) and for the global variable we use to expose the current taken predicate to called functions. On the code transformation side, an implementation extension may be to avoid such promotion, then use thread-local storage for the predicate, and update the doubly linked lists for allocation sites atomically. As for DFL handlers, we may implement DFL handlers either using locking mechanisms, or moving to more efficient lockless implementations using atomic operations or, for non-small involved sizes (Appendix F.2), TSX transactions.

F.5 Correctness

We discuss an informal proof of correctness of the Constantine approach. As we anticipated in Section 5.5, to prove that our programs are semantically equivalent to their original representations, we break the claim into two parts: control-flow

Table F.2. Run-time overhead analysis of different CONSTANTINE configurations (i.e., $\lambda = 1$, AVX2). The numbers for SC-Eliminator and Soares et al. were obtained from executing the publicly available artifact evaluation material for their papers.

	program	AVX512 ($\lambda = 1$)	AVX512 ($\lambda = 4$)	AVX512 ($\lambda = 64$)	AVX2 ($\lambda = 4$)	AVX2 ($\lambda = 64$)	SC-Eliminator	Soares et al.
CHRONOS	aes	1.13	1.13	1.08	1.22	1.08	1.11	1.02
	des	1.12	1.19	1.14	1.36	1.15	1.09	1.00
	des3	1.49	1.49	1.36	1.86	1.37	1.12	1.02
	anubis	1.29	1.29	1.12	1.55	1.22	1.06	1.00
	cast5	1.13	1.13	1.06	1.25	1.12	1.06	1.02
	cast6	1.13	1.13	1.08	1.13	1.07	1.05	1.00
	fcrypt	1.04	1.04	1.03	1.06	1.01	1.05	1.00
	khazad	1.13	1.13	1.09	1.23	1.07	1.30	1.00
S-CP	aes_core	1.12	1.12	1.06	1.01	1.05	1.06	1.04
	cast-ssl	1.23	1.23	1.10	1.38	1.15	1.17	1.01
BOTAN	aes	1.05	1.05	1.03	1.09	1.01	1.01	-
	cast128	1.02	1.02	1.01	1.03	1.01	1.05	-
	des	1.01	1.01	1.01	1.01	1.01	1.08	-
	kasumi	1.01	1.01	1.01	1.03	1.01	1.01	-
	seed	1.02	1.02	1.01	1.03	1.01	1.03	-
	twofish	1.14	1.14	1.12	1.21	1.15	1.04	-
APP-CR	3way	1.00	1.00	1.00	1.00	1.00	1.03	1.15
	des	1.24	1.24	1.09	1.27	1.06	1.08	1.11
	loki91	1.51	1.51	1.43	1.48	1.48	1.97	1.24
LIBGCRYPT	camellia	1.02	1.02	1.01	1.02	1.01	1.08	1.01
	des	1.06	1.06	1.06	1.05	1.09	1.03	1.01
	seed	1.18	1.18	1.10	1.21	1.01	1.15	1.01
	twofish	1.97	1.97	1.92	2.45	2.10	1.41	1.24
RACCOON	binsearch	1.33	1.33	1.18	1.30	1.16	-	-
	dijkstra	3.87	3.45	1.51	2.83	1.50	-	-
	findmax	1.00	1.00	1.00	1.00	1.00	-	-
	histogram	2.66	2.66	1.68	4.39	1.87	-	-
	matmul	1.00	1.00	1.00	1.00	1.00	-	-
	rsort	1.87	1.87	1.84	1.50	1.45	-	-
PYCRYPTO	aes	1.13	1.13	1.06	1.33	1.11	-	-
	arc4	1.07	1.07	1.03	1.08	1.03	-	-
	blowfish	5.07	5.07	3.17	10.58	3.23	-	-
	cast	1.09	1.09	1.04	1.16	1.08	-	-
	des3	1.06	1.06	1.04	1.08	1.05	-	-
B/REL	tls-rempad-luk13	1.01	1.01	1.01	1.01	1.01	-	-
	aes_big	1.02	1.01	1.01	1.02	1.01	-	-
	des_tab	1.04	1.04	1.02	1.07	1.03	-	-
	geomean (total)	1.26	1.26	1.16	1.34	1.17	1.12	1.05

correctness and data-flow correctness For each part we assume that the other holds, so that the initial claim can hold by construction.

For control flows, we need to show that along real paths the transformed program performs all and only the computations that the original one would make. First, we rule out divergences from exceptional control flow since the original program is error-free and CFL sanitizes sequences that may throw (e.g., division) when in dummy execution. We then observe that by construction CFL forces the program to explore both outcomes of every branch, and the decision whether to treat each direction in dummy execution depends on the *taken* predicate value. CFL builds this predicate as the combination of the control-flow decisions that the (original) program takes on the program state, and from the data flow argument decoy paths have no effects on such state. All the linearized branch directions will be executed as many times and in the same interleaving observable in the original program; as for the special loop case, the amount of real and decoy iterations depends on the original loop condition and the *taken* predicate, so its real iterations closely match the original loop. This completes the argument.

For data flows, we need to show that values computed in dummy execution cannot flow into real paths, and that decoy paths preserve memory safety. The points-to metadata fed to DFL load and store wrappers make the program access the same memory objects along both real and decoy paths, and allocation metadata ensure that those objects are valid: memory safety is guaranteed. Also, only real paths can change memory contents during a store, hence only values written by real paths can affect data loads. Thus, we only need to reason about data flows from local variables assigned in dummy execution. LLVM IR hosts such variables in SSA virtual registers, and at any program point only one variable instance can be live [308]. For a top-level linearized branch, a `ct_select` statement chooses the incoming value from the real path (Section 5.4.2.1). For a nested branch (Figure 5.2) both directions may be part of dummy execution. Regardless of which value the inner `ct_select` will choose, the outer one eventually picks the value coming from the real path that did not contain the branch. Extending the argument to three or more nested branches is analogous: for a variable that outlives a linearized region, whenever such variable is later accessed on a real path, the value from a real path would assign to it (otherwise the original program would be reading an undefined value or control-flow correctness would be violated), while in decoy paths bogus value can freely flow. We discussed correctness for loops in Section 5.4.2.4.

F.6 Complete Run-Time Overhead Data

Table F.2 shows the complete set of our performance-oriented experiments: we benchmarked Constantine with different λ values $(1, 4, 64)$ and SIMD capabilities (AVX2 and AVX512), and also ran the artifacts from [389] and [332] on the same setup used for Section 5.6. For the latter we did not try the Raccoon microbenchmarks, mostly due to compatibility problems and limitations of the artifacts, while the SCE suite was part of their original evaluation (Soares et al. leave the `botan` group out of the artifact evaluation harness). Both systems provide much weaker security properties than Constantine, yet the average overhead numbers we observe are similar. Also, the availability of AVX512 instructions brings benefits for the $\lambda = 64$ setting as they allow DFL to touch more cache lines at once over large object portions (Appendix F.2). Interesting, protection for the presently unrealistic $\lambda = 1$ attack vector leads to overheads that are identical to the $\lambda = 4$ configuration for MemJam-like attacks, with `dijkstra` being the only exception (3.87x vs 3.45x).

Appendix G

Chapter 6: Additional Material

G.1 Software Workaround

```
#define load_secret128(mem) _mm_i32gather_ps((float*)(mem),
    _mm_set_epi32(12, 8, 4, 0), 4)

#define store_secret128(mem, secret)
    _mm_i32scatter_ps((float*)(mem), _mm_set_epi32(12, 8, 4, 0),
    secret, 4)

#define define_secret128(name, secret) unsigned int
    __attribute__((aligned(64))) name[] = {((unsigned
    int*)(secret))[0], 0, 0, 0, ((unsigned int*)(secret))[1], 0, 0,
    0, ((unsigned int*)(secret))[2], 0, 0, 0, ((unsigned
    int*)(secret))[3], 0, 0, 0, }

// define an 128-bit secret at compile time
define_secret128(secret, "ABCDEFGHIJKLMNO");
// load an 128-bit secret from memory into an XMM register
__m128 xsecret = load_secret128(secret);
// store an 128-bit secret to memory
store_secret128(secret2, xsecret);
```

Listing G.1. A software workaround for spreading 128-bit secrets to non-leakable 4-byte blocks using AVX.

Listing G.1 shows a possible software workaround for specific scenarios, e.g., for protecting AES-NI keys. Secrets can be scattered to memory locations shadowed by valid APIC registers and thus not leakable. This is possible with a single AVX instruction. Furthermore, scattered secrets can also be copied into one XMM register with a single instruction. If an XMM register is chosen, that cannot be leaked via the SSA, this drastically reduces the attack surface. To ensure that a secret key cannot be leaked, this method can additionally be combined with the transient computation shown in Appendix G.2.

G.2 Transient AES Computation

```
char aes_encrypt_get_byte(__m128 message, int target_byte) {
 char bytevalue[256 * 4096];
 // the following code is never executed architecturally, only
    transiently
 if(<mispredicted>) {
  // repeat for all AES-NI rounds
  __m128 roundkey = load_secret128(roundkey[i]);
  // AES-NI encryption rounds
  _mm_aesenc_si128(message, roundkey)
  [...]
  // encode in cache
  volatile char dummy = bytevalue[((message >> (8 * target_byte)) &
    0xFF) * 4096];
 }
 // recover encoded byte from cache
 for(int i = 0; i < 256; i++)
  if(flush_reload(bytevalue[i * 4096])) return i;
}
```

Listing G.2. The function transiently encrypts a message using AES-NI, encodes one chosen byte of the ciphertext in the cache, and recovers it from the cache using Flush+Reload. Although this function has to be called 16 times to get all bytes of the ciphertext, it ensures that registers containing key material never end up in the SSA.

Listing G.2 demonstrates how AES-NI can be used securely inside SGX by doing all computations in the transient domain. During transient execution, e.g., via a mispredicted branch, the round key is loaded using the scatter-based technique shown in Appendix G.1. The round key is then used in the transient execution to encrypt the plain text, and encode one byte of the result in the cache. After the transient execution, this encoded byte can be inferred using a side-channel attack. The encoding can be chosen in a way that it can be decoded from within the enclave using, e.g., Evict+Reload, but not from outside the enclave, e.g., with Prime+Probe. While this method is not very efficient, it can protect an AES key even if the CPU is affected by ÆPIC Leak.

G.3 Performance Counters and CPUs

Table G.1 shows all tested CPUs for ÆPIC Leak. Table G.2 shows performance counters that differ when loading from defined and undefined offsets of the local APIC MMIO region.

Table G.1. Tested CPUs that are (✓) or are not (✗) vulnerable. All tested Sunny-Cove-based CPUs are vulnerable.

CPU	Microarchitecture	Based on	ÆPIC Leak
Intel Core i5-2520M	Sandy Bridge	Sandy Bridge	✗
Intel Core i5-3230M	Ivy Bridge	Sandy Bridge	✗
Intel Core i3-4160T	Haswell	Haswell	✗
Intel Core i7-4790	Haswell	Haswell	✗
Intel Core i3-5010U	Broadwell	Haswell	✗
Intel Core i7-6700K	Skylake	Skylake	✗
Intel Core i3-7100T	Kaby Lake	Skylake	✗
Intel Core i3-8130U	Kaby Lake R	Skylake	✗
Intel Core i7-8565U	Whiskey Lake	Skylake	✗
Intel Core i7-8700K	Coffee Lake	Skylake	✗
Intel Core i9-9980HK	Coffee Lake	Skylake	✗
Intel Core i9-9900K	Coffee Lake	Skylake	✗
Intel Core i7-10510U	Comet Lake	Skylake	✗
Intel Core i3-1005G1	Ice Lake	Sunny Cove	✓
Intel Core i5-1035G1	Ice Lake	Sunny Cove	✓
Intel Core i5-1135G7	Tiger Lake	Willow Cove	✗
Intel Core i9-12900K	Alder Lake	Sunny Cove	✓
Intel Xeon E5-1630 v4	Broadwell	Haswell	✗
Intel Xeon E3-1505M v5	Skylake	Skylake	✗
Intel Xeon E-2176M	Coffee Lake	Skylake	✗
Intel Xeon Silver 4208	Cascade Lake-SP	Cascade Lake	✗
Intel Xeon Platinum 8375C	Ice Lake SP	Sunny Cove	✓
Intel Celeron N3350	Apollo Lake	Goldmont	✗
Intel Celeron J4005	Gemini Lake	Goldmont Plus	✗
Intel Celeron N4500	Jasper Lake	Tremont	✗
AMD Ryzen 5 2500U	Zen	Zen	✗
AMD Ryzen 5 3550H	Zen	Zen	✗
AMD Ryzen Threadripper 1920X	Zen	Zen	✗
AMD Ryzen 7 3700X	Zen 2	Zen 2	✗
AMD Ryzen 7 5800X	Zen 3	Zen 3	✗
AMD EPYC 7443	Zen 3	Zen 3	✗

Table G.2. Performance counter differences on loads to defined and undefined offsets of the APIC memory-mapped region on Intel i3-1005G1 CPU.

Performance Counter	Defined	Undefined
BR_INST_RETIRED.FAR_BRANCH	0.05 ($\sigma_{\bar{x}} = 0.00$)	1.05 ($\sigma_{\bar{x}} = 0.00$)
CORE_POWER.LVLO_TURBO_LICENSE	68.14 ($\sigma_{\bar{x}} = 0.11$)	627.76 ($\sigma_{\bar{x}} = 0.09$)
CPU_CLK_UNHALTED.DISTRIBUTED	68.13 ($\sigma_{\bar{x}} = 0.11$)	627.73 ($\sigma_{\bar{x}} = 0.10$)
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	0.75 ($\sigma_{\bar{x}} = 0.00$)	7.08 ($\sigma_{\bar{x}} = 0.00$)
CPU_CLK_UNHALTED.REF_DISTRIBUTED	0.75 ($\sigma_{\bar{x}} = 0.00$)	7.08 ($\sigma_{\bar{x}} = 0.00$)
CPU_CLK_UNHALTED.REF_XCLK	0.75 ($\sigma_{\bar{x}} = 0.00$)	7.08 ($\sigma_{\bar{x}} = 0.00$)
CYCLE_ACTIVITY.CYCLES_L1D_MISS	191.84 ($\sigma_{\bar{x}} = 0.36$)	176.34 ($\sigma_{\bar{x}} = 0.02$)
CYCLE_ACTIVITY.CYCLES_MEM_ANY	866.04 ($\sigma_{\bar{x}} = 0.73$)	2656.24 ($\sigma_{\bar{x}} = 0.12$)
CYCLE_ACTIVITY.STALLS_L1D_MISS	427.20 ($\sigma_{\bar{x}} = 0.58$)	2062.15 ($\sigma_{\bar{x}} = 0.20$)
CYCLE_ACTIVITY.STALLS_L2_MISS	235.36 ($\sigma_{\bar{x}} = 0.26$)	1885.75 ($\sigma_{\bar{x}} = 0.20$)
CYCLE_ACTIVITY.STALLS_L3_MISS	235.36 ($\sigma_{\bar{x}} = 0.26$)	1885.76 ($\sigma_{\bar{x}} = 0.20$)
CYCLE_ACTIVITY.STALLS_MEM_ANY	1101.16 ($\sigma_{\bar{x}} = 0.93$)	4541.89 ($\sigma_{\bar{x}} = 0.29$)
CYCLE_ACTIVITY.STALLS_TOTAL	235.37 ($\sigma_{\bar{x}} = 0.26$)	1886.16 ($\sigma_{\bar{x}} = 0.25$)
DSB2MITE_SWITCHES.COUNT	0.95 ($\sigma_{\bar{x}} = 0.01$)	5.91 ($\sigma_{\bar{x}} = 0.21$)
DSB2MITE_SWITCHES.PENALTY_CYCLES	0.95 ($\sigma_{\bar{x}} = 0.01$)	5.91 ($\sigma_{\bar{x}} = 0.21$)
EXE_ACTIVITY.1_PORTS_UTIL	1.95 ($\sigma_{\bar{x}} = 0.02$)	83.57 ($\sigma_{\bar{x}} = 0.02$)
EXE_ACTIVITY.2_PORTS_UTIL	2.64 ($\sigma_{\bar{x}} = 0.02$)	35.75 ($\sigma_{\bar{x}} = 0.02$)
EXE_ACTIVITY.3_PORTS_UTIL	2.05 ($\sigma_{\bar{x}} = 0.01$)	18.89 ($\sigma_{\bar{x}} = 0.01$)
EXE_ACTIVITY.4_PORTS_UTIL	0.76 ($\sigma_{\bar{x}} = 0.00$)	2.13 ($\sigma_{\bar{x}} = 0.00$)
ICACHE_64B.IFTAG_HIT	3.58 ($\sigma_{\bar{x}} = 0.01$)	20.80 ($\sigma_{\bar{x}} = 0.29$)
ICACHE_64B.IFTAG_STALL	2.01 ($\sigma_{\bar{x}} = 0.00$)	33.80 ($\sigma_{\bar{x}} = 1.21$)
IDQ.DSB_CYCLES_ANY	6.89 ($\sigma_{\bar{x}} = 0.00$)	24.52 ($\sigma_{\bar{x}} = 0.16$)
IDQ.DSB_CYCLES_OK	6.89 ($\sigma_{\bar{x}} = 0.00$)	24.52 ($\sigma_{\bar{x}} = 0.16$)
IDQ.DSB_UOPS	6.89 ($\sigma_{\bar{x}} = 0.00$)	24.52 ($\sigma_{\bar{x}} = 0.16$)
IDQ.MITE_CYCLES_ANY	6.11 ($\sigma_{\bar{x}} = 0.00$)	21.73 ($\sigma_{\bar{x}} = 0.14$)
IDQ.MITE_CYCLES_OK	6.11 ($\sigma_{\bar{x}} = 0.00$)	21.79 ($\sigma_{\bar{x}} = 0.15$)
IDQ.MITE_UOPS	6.11 ($\sigma_{\bar{x}} = 0.00$)	21.79 ($\sigma_{\bar{x}} = 0.15$)
IDQ.MS_CYCLES_ANY	16.32 ($\sigma_{\bar{x}} = 0.12$)	385.65 ($\sigma_{\bar{x}} = 0.19$)
IDQ.MS_SWITCHES	16.32 ($\sigma_{\bar{x}} = 0.12$)	385.65 ($\sigma_{\bar{x}} = 0.19$)
IDQ.MS_UOPS	16.38 ($\sigma_{\bar{x}} = 0.12$)	385.56 ($\sigma_{\bar{x}} = 0.19$)
IDQ_UOPS_NOT_DELIVERED.CORE	19.84 ($\sigma_{\bar{x}} = 0.20$)	411.07 ($\sigma_{\bar{x}} = 0.18$)
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE	19.84 ($\sigma_{\bar{x}} = 0.20$)	411.07 ($\sigma_{\bar{x}} = 0.18$)
IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	19.84 ($\sigma_{\bar{x}} = 0.20$)	411.07 ($\sigma_{\bar{x}} = 0.18$)
INST_RETIRED.STALL_CYCLES	12.10 ($\sigma_{\bar{x}} = 0.00$)	13.10 ($\sigma_{\bar{x}} = 0.00$)
INT_MISC.ALL_RECOVERY_CYCLES	0.31 ($\sigma_{\bar{x}} = 0.01$)	27.49 ($\sigma_{\bar{x}} = 0.01$)
INT_MISC.CLEAR_RESTEER_CYCLES	0.25 ($\sigma_{\bar{x}} = 0.01$)	25.32 ($\sigma_{\bar{x}} = 0.02$)
INT_MISC.RECOVERY_CYCLES	0.31 ($\sigma_{\bar{x}} = 0.01$)	27.19 ($\sigma_{\bar{x}} = 0.01$)
INT_MISC.UOP_DROPPING	0.01 ($\sigma_{\bar{x}} = 0.00$)	5.20 ($\sigma_{\bar{x}} = 0.00$)
ITLB_MISSES.WALK_ACTIVE	0.94 ($\sigma_{\bar{x}} = 0.00$)	10.76 ($\sigma_{\bar{x}} = 0.34$)
ITLB_MISSES.WALK_PENDING	0.94 ($\sigma_{\bar{x}} = 0.00$)	10.80 ($\sigma_{\bar{x}} = 0.34$)
L1D_PEND_MISS.PENDING	23.98 ($\sigma_{\bar{x}} = 0.04$)	22.06 ($\sigma_{\bar{x}} = 0.02$)
L1D_PEND_MISS.PENDING_CYCLES	23.98 ($\sigma_{\bar{x}} = 0.04$)	22.06 ($\sigma_{\bar{x}} = 0.02$)
MACHINE_CLEARS.COUNT	0.01 ($\sigma_{\bar{x}} = 0.01$)	1.10 ($\sigma_{\bar{x}} = 0.01$)
MEM_INST_RETIRED.ALL_LOADS	2.95 ($\sigma_{\bar{x}} = 0.00$)	3.95 ($\sigma_{\bar{x}} = 0.00$)
MEM_INST_RETIRED.ANY	3.55 ($\sigma_{\bar{x}} = 0.00$)	4.55 ($\sigma_{\bar{x}} = 0.00$)
MEM_LOAD_RETIRED.L1_HIT	1.95 ($\sigma_{\bar{x}} = 0.00$)	2.95 ($\sigma_{\bar{x}} = 0.00$)
RESOURCE_STALLS.SCOREBOARD	57.34 ($\sigma_{\bar{x}} = 0.05$)	458.50 ($\sigma_{\bar{x}} = 0.06$)
RS_EVENTS.EMPTY_CYCLES	54.92 ($\sigma_{\bar{x}} = 0.05$)	318.39 ($\sigma_{\bar{x}} = 0.06$)
RS_EVENTS.EMPTY_END	54.92 ($\sigma_{\bar{x}} = 0.05$)	318.39 ($\sigma_{\bar{x}} = 0.06$)
TIME	44.05 ($\sigma_{\bar{x}} = 0.04$)	433.64 ($\sigma_{\bar{x}} = 1.69$)
TOPDOWN.BACKEND_BOUND_SLOTS	289.81 ($\sigma_{\bar{x}} = 0.25$)	2346.41 ($\sigma_{\bar{x}} = 0.13$)
TOPDOWN.BR_MISPREDICT_SLOTS	0.86 ($\sigma_{\bar{x}} = 0.06$)	86.21 ($\sigma_{\bar{x}} = 0.05$)
TOPDOWN.SLOTS_P	341.06 ($\sigma_{\bar{x}} = 0.54$)	3138.51 ($\sigma_{\bar{x}} = 0.68$)
UOPS_DECODED.DEC0	0.46 ($\sigma_{\bar{x}} = 0.00$)	3.25 ($\sigma_{\bar{x}} = 0.10$)
UOPS_DISPATCHED.PORT_0	2.90 ($\sigma_{\bar{x}} = 0.02$)	47.32 ($\sigma_{\bar{x}} = 0.02$)
UOPS_DISPATCHED.PORT_1	2.86 ($\sigma_{\bar{x}} = 0.02$)	57.18 ($\sigma_{\bar{x}} = 0.02$)
UOPS_DISPATCHED.PORT_2_3	3.05 ($\sigma_{\bar{x}} = 0.00$)	9.05 ($\sigma_{\bar{x}} = 0.00$)
UOPS_DISPATCHED.PORT_4_9	4.30 ($\sigma_{\bar{x}} = 0.00$)	6.30 ($\sigma_{\bar{x}} = 0.00$)
UOPS_DISPATCHED.PORT_5	2.86 ($\sigma_{\bar{x}} = 0.02$)	42.42 ($\sigma_{\bar{x}} = 0.02$)
UOPS_DISPATCHED.PORT_6	5.67 ($\sigma_{\bar{x}} = 0.03$)	72.28 ($\sigma_{\bar{x}} = 0.03$)
UOPS_DISPATCHED.PORT_7_8	4.25 ($\sigma_{\bar{x}} = 0.00$)	7.25 ($\sigma_{\bar{x}} = 0.00$)
UOPS_EXECUTED.CORE	25.83 ($\sigma_{\bar{x}} = 0.09$)	242.72 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CORE_CYCLES_GE_1	25.83 ($\sigma_{\bar{x}} = 0.09$)	242.76 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CORE_CYCLES_GE_2	25.83 ($\sigma_{\bar{x}} = 0.09$)	242.76 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CORE_CYCLES_GE_3	25.83 ($\sigma_{\bar{x}} = 0.09$)	242.76 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CORE_CYCLES_GE_4	25.83 ($\sigma_{\bar{x}} = 0.09$)	242.76 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CYCLES_GE_1	25.74 ($\sigma_{\bar{x}} = 0.09$)	242.58 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CYCLES_GE_2	25.74 ($\sigma_{\bar{x}} = 0.09$)	242.58 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CYCLES_GE_3	25.74 ($\sigma_{\bar{x}} = 0.09$)	242.58 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.CYCLES_GE_4	25.74 ($\sigma_{\bar{x}} = 0.09$)	242.58 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.STALL_CYCLES	25.76 ($\sigma_{\bar{x}} = 0.09$)	242.68 ($\sigma_{\bar{x}} = 0.09$)
UOPS_EXECUTED.THREAD	25.76 ($\sigma_{\bar{x}} = 0.09$)	242.68 ($\sigma_{\bar{x}} = 0.09$)
UOPS_ISSUED.ANY	29.38 ($\sigma_{\bar{x}} = 0.12$)	272.26 ($\sigma_{\bar{x}} = 0.12$)
UOPS_ISSUED.STALL_CYCLES	29.45 ($\sigma_{\bar{x}} = 0.12$)	272.27 ($\sigma_{\bar{x}} = 0.12$)
UOPS_RETIRED.SLOTS	29.60 ($\sigma_{\bar{x}} = 0.09$)	249.53 ($\sigma_{\bar{x}} = 0.08$)
UOPS_RETIRED.STALL_CYCLES	29.60 ($\sigma_{\bar{x}} = 0.09$)	249.53 ($\sigma_{\bar{x}} = 0.08$)
UOPS_RETIRED.TOTAL_CYCLES	28.00 ($\sigma_{\bar{x}} = 0.08$)	247.95 ($\sigma_{\bar{x}} = 0.08$)

Bibliography

[1] A. Abel and J. Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS*, 2019.

[2] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *S&P*, 2022.

[3] J. Agat. Transforming out timing leaks. In *POPL*, 2000.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[5] Y. Alaoui. Exploiting Intel's Management Engine - Part 2: Enabling Red JTAG Unlock on Intel ME 11.x (INTEL-SA-00086), 2019.

[6] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. Port Contention for Fun and Profit. In *S&P*, 2019.

[7] S. Alvarez. Radare2. https://rada.re/n/. Online; accessed 11 June 2020.

[8] N. Amit, F. Jacobs, and M. Wei. Jumpswitches: Restoring the performance of indirect branches in the era of spectre. In *USENIX ATC*, 2019.

[9] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, 1997.

[10] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[11] D. Andriesse, H. Bos, and A. Slowinska. Parallax: Implicit code integrity verification using return-oriented programming. In *DSN*, 2015.

[12] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *S&P*, 2015.

[13] M. Andrysco, A. Nötzli, F. Brown, R. Jhala, and D. Stefan. Towards verified, constant-time floating point operations. In *CCS*, 2018.

[14] M. Angelini, G. Blasilli, P. Borrello, E. Coppa, D. C. D'Elia, S. Ferracci, S. Lenti, and G. Santucci. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *Proceedings of the 15th IEEE Symposium on Visualization for Cyber Security*, 2018.

[15] Anton Ertl. Reorder buffer size of various CPUs, 2019.

[16] A. W. Appel. Ssa is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998.

[17] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In *CCS*, 2020.

[18] ARM. Arm Architecture Reference Manual for A-profile architecture, 2022.

[19] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[20] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. Cryptology ePrint Archive, Paper 2012/351, 2012.

[21] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. K-calling context profiling. In *OOPSLA*, 2012.

[22] R. Avanzi. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology*, 2017.

[23] B. Azad. Examining Pointer Authentication on the iPhone XS, 2019.

[24] I. Baev and Q. I. Center. Profile-based indirect call promotion. In *LLVM Developers Meeting, Oct*, 2015.

[25] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 - A platform for analyzing x86 executables. In *CC*, 2005.

[26] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computer Surveys*, 51(3):50:1–50:39, 2018.

[27] S. Banescu, C. Collberg, and A. Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *USENIX Security Symposium*, 2017.

[28] S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.

[29] S. Banescu and A. Pretschner. Chapter five - A tutorial on software obfuscation. *Advances in Computers*, 108:283–353, 2018.

[30] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security Symposium*, 2022.

[31] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *CSF*, 2018.

[32] E. Bauman, Z. Lin, and K. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[33] K. Bhat, E. v. d. Kouwe, H. Bos, and C. Giuffrida. FIRestarter: Practical software crash recovery with targeted library-level fault injection. In *DSN*, 2021.

[34] K. Bhat, E. van der Kouwe, H. Bos, and C. Giuffrida. Probeguard: Mitigating probing attacks through reactive program transformations. In *ASPLOS*, 2019.

[35] A. Bhattacharyya, A. Sandulescu, M. Neugschwandt ner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[36] F. Biondi, S. Josse, A. Legay, and T. Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.

[37] P. Biswas, N. Burow, and M. Payer. Code specialization through dynamic feature observation. In *CODASPY*, 2021.

[38] Bitcoin. Android security vulnerability, 2013.

[39] T. Blazytko, M. Contag, C. Aschermann, and T. Holz. Syntia: Synthesizing the semantics of obfuscated code. In *USENIX Security Symposium*, 2017.

[40] D. Bleichenbacher. Forging Some RSA Signatures with Pencil and Paper. *CRYPTO*, 2006.

[41] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *ASIACCS*, 2011.

[42] M. Böhme and S. Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, 2016.

[43] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *CCS*, 2017.

[44] C. Bölük. Speculating The Entire X86-64 Instruction Set In Seconds With This One Weird Trick, 2021.

[45] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2007.

[46] N. Borisov, I. Goldberg, and D. Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *MobiCom*, 2001.

[47] P. Borrello, E. Coppa, D. C. D'Elia, and C. Demetrescu. The ROP needle: Hiding trigger-based injection vectors via code reuse. In *SAC*, 2019.

[48] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS*, 2021.

[49] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security Symposium*, 2022.

[50] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu. Memory models in symbolic execution: key ideas and new thoughts. *Soft. Testing, Verification and Reliability*, 29(8), 2019.

[51] P. Bosch. Intel LDAT notes, 2020.

[52] P. Bosch. Under the hood of a CPU: Reverse Engineering the P6 Microcode. In *Hardwear.io Netherlands*, 2020.

[53] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *S&P*, 2016.

[54] N. Brown. Smatch: pluggable static analysis for c. https://lwn.net/Articles/691882/.

[55] N. Brown. Sparse: a look under the hood. https://lwn.net/Articles/689907/.

[56] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *CGO*, 2011.

[57] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS*, 2014.

[58] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[59] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[60] C. Canella, S. M. Pudukotai Dinakarrao, D. Gruss, and K. N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI*, 2020.

[61] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019.

[62] T. Carter and J. Robertson. Radix-16 signed-digit division. *IEEE Transactions on Computers*, 1990.

[63] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. In *OOPSLA*, 2012.

[64] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new spectre era. In *PLDI*, 2020.

[65] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. Fact: A dsl for timing-sensitive computation. In *PLDI*, 2019.

[66] D. D. Chen and G.-J. Ahn. Security Analysis of x86 Processor Microcode. Bachelor's Thesis, Arizona State University, 2014.

[67] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *CCS*, 2018.

[68] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *S&P*, 2018.

[69] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing matlab through just-in-time specialization. In *Compiler Construction*, 2010.

[70] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[71] S. Chow, Y. X. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of the 4th International Conference on Information Security*, 2001.

[72] Christopher Domas. The m/o/vfuscator, 2015.

[73] Clang. LLVM's Control Flow Integrity, 2018.

[74] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4), 2012.

[75] T. Coe. Inside the pentium-fdiv bug. *Doctor Dobb's Journal*, 1995.

[76] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *ACSAC*, 2012.

[77] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[78] C. S. Collberg and J. Nagra. *Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley, 2010.

[79] R. R. Collins. The intel pentium f00f bug description and workarounds. *Doctor Dobb's Journal*, 1997.

[80] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *CCS*, 2011.

[81] K. Cook. Bounded flexible arrays in c. https://people.kernel.org/kees/bounded-flexible-arrays-in-c, 2023.

[82] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P*, 2009.

[83] J. Corbet. Moving the kernel to modern C. https://lwn.net/Articles/885941/, 2022.

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

[85] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *CCS*, 2017.

[86] V. Costan and S. Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[87] V. Costan, I. Lebedev, S. Devadas, et al. *Secure processors part II: Intel SGX security analysis and MIT sanctum architecture*. Now Publishers, 2017.

[88] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[89] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai. Smashex: Smashing sgx enclaves using exceptions. In *CCS*, 2021.

[90] L.-A. Daniel, S. Bardin, and T. Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *S&P*, 2020.

[91] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[92] J. Davis, A. Slobodova, and S. Swords. Microcode Verification - Another Piece of the Microprocessor Verification Puzzle. In *ITP*, 2014.

[93] A. C. De Melo. Profiling data structures. https://lpc.events/event/16/contributions/1200/attachments/1054/2013/Profiling%20Data%20Structures.pdf, 2022.

[94] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *ASIACCS*, 2019.

[95] D. C. D'Elia, E. Coppa, A. Salvati, and C. Demetrescu. Static Analysis of ROP Code. In *EuroSec*, 2019.

[96] D. C. D'Elia and C. Demetrescu. Flexible On-stack Replacement in LLVM. In *CGO*, 2016.

[97] D. C. D'Elia and C. Demetrescu. On-stack replacement, distilled. In *PLDI*, 2018.

[98] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *PLDI*, 2011.

[99] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. *Software: Practice and Experience*, 46(8):1131–1152, 2016.

[100] J. D. DeMott and R. Enbody. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In *Black Hat USA*, 2007.

[101] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran. HardFails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Security Symposium*, 2019.

[102] M. Dillon. Some Ryzen Linux Users Are Facing Issues With Heavy Compilation Loads, 2017.

[103] S. Dinesh, N. Burow, D. Xu, and M. Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *S&P*, 2020.

[104] Z. Y. Ding and C. L. Goues. An empirical study of oss-fuzz bugs. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142, 2021.

[105] C. Domas. The Memory Sinkhole. In *BlackHat USA*, 2015.

[106] C. Domas. Breaking the x86 ISA. In *BlackHat USA*, 2017.

[107] C. Domas. God Mode Unlocked: Hardware Backdoors in x86 CPUs. In *BlackHat USA*, 2018.

[108] C. Domas. Hardware Backdoors in x86 CPUs. *Black Hat US*, 2018.

[109] J. Dougall. Apple M1 Microarchitecture Research, 2021.

[110] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[111] T. Downs. Hardware store elimination. https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html, 2020.

[112] T. Downs. Ice Lake Store Elimination, 2020.

[113] G. J. Duck and R. H. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. In *PLDI*, 2018.

[114] G. J. Duck, R. H. Yap, and L. Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS*, 2017.

[115] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU system management mode to circumvent operating system security functions. *CanSecWest*, 2006.

[116] V. Duta, C. Giuffrida, H. Bos, and E. van der Kouwe. Pibe: Practical kernel control-flow hardening with profile-guided indirect branch elimination. In *ASPLOS*, 2021.

[117] C. Eagle and K. Nance. *The Ghidra Book: The Definitive Guide*. No Starch Press, 2020.

[118] C. Easdon. Undocumented CPU Behavior: Analyzing Undocumented Opcodes on Intel x86-64. Talk, 2018.

[119] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss. Rapid Prototyping for Microarchitectural Attacks. In *USENIX Security Symposium*, 2022.

[120] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou. Detecting ROP with statistical learning of program characteristics. In *CODASPY*, 2017.

[121] A. S. Engineering and Architecture. Towards the next generation of xnu memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/, 2022.

[122] M. Ermolov, D. Sklyarov, and M. Goryachy. glm-ucode, 2020.

[123] M. Ermolov, D. Sklyarov, and M. Goryachy. Chip Red Pill: How we Achieved the Arbitrary [micro]Code Execution inside Intel Atom CPUs. In *OffensiveCon 22*, 2022.

[124] M. Ermolov, D. Sklyarov, and M. Goryachy. MicrocodeDecryptor, 2022.

[125] M. Ermolov, D. Sklyarov, and M. Goryachy. uCodeDisasm, 2022.

[126] M. Ermolov, D. Sklyarov, and M. Goryachy. Undocumented x86 Instructions to Control the CPU at the Microarchitecture Level in Modern Intel Processors. *Journal of Computer Virology and Hacking Techniques*, 2022.

[127] B. Fahim, Y.-C. Liu, C.-C. Wang, D. C. Soltis, T. C. Huang, T. Singh, B. Jung, and N. Haider. Shared mesh, 2017. US Patent 2017/0019350 A1.

[128] Fail0verflow. Console hacking 2010, 2010.

[129] Falk, Brandon. Mesos, 2020.

[130] A. Fioraldi, D. C. D'Elia, and D. Balzarotti. The use of likely invariants as feedback for fuzzers. In *USENIX Security Symposium*, 2021.

[131] A. Fioraldi, D. C. D'Elia, and E. Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *ISSTA*, 2020.

[132] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining incremental steps of fuzzing research. In *WOOT*, 2020.

[133] FireEye. The Number of the Beast. https://www.fireeye.com/blog/threat-research/2013/02/the-number-of-the-beast.html, 2013. Online; accessed 11 June 2020.

[134] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, 2012.

[135] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, 2014.

[136] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. GREYONE: Data flow sensitive fuzzing. In *USENIX Security Symposium*, 2020.

[137] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *S&P*, 2018.

[138] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, 2009.

[139] GitHub. CodeQL. https://codeql.github.com/.

[140] GNU. GDB: The GNU Project Debugger, 2022.

[141] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[142] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[143] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431?473, 1996.

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

[145] Google. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz, 2016. [Online; accessed 10 Sep. 2021].

[146] I. Gouy. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/, 2018. Online; accessed 20 March 2021.

[147] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[148] M. Graziano, D. Balzarotti, and A. Zidouemba. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *ASIACCS*, 2016.

[149] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, 2017.

[150] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[151] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*, 2015.

[152] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. Van Der Kouwe. TypeSan: Practical type confusion detection. In *CCS*, 2016.

[153] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani. Demystifying the dependency challenge in kernel fuzzing. In *ICSE*, 2022.

[154] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(03):243–250, 1977.

[155] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *TOPLAS*, 16(2):175–204, 1994.

[156] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *CGO*, 2012.

[157] R. Hastings. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter Conference*, 1992.

[158] B. Hawkes. Notes on Intel Microcode Updates, 2012.

[159] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *CGO*, 2003.

[160] S. He, M. Emmi, and G. Ciocarlie. ct-fuzz: Fuzzing for timing leaks. In *ICST*, 2020.

[161] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 2011.

[162] A. Herrera, M. Payer, and A. Hosking. datAFLow: Towards a data-flow-guided fuzzer. In *FUZZING*, 2022.

[163] L. Hetterich and M. Schwarz. Branch Different - Spectre Attacks on Apple Silicon. In *DIMVA*, 2022.

[164] M. 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*, 2001.

[165] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, 2012.

[166] J. Horn. speculative execution, variant 4: speculative store bypass, 2018.

[167] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. Enforcing unique code target property for control-flow integrity. In *CCS*, 2018.

[168] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *S&P*, 2016.

[169] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, 2009.

[170] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[171] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. Understanding contention-based channels and using them for defense. In *HPCA*, 2015.

[172] N. Hussein. Randomizing structure layout. https://lwn.net/Articles/722293/, 2017.

[173] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE*, 2014.

[174] Intel. Application Note AP-526: Optimizations for Intel's 32-Bit Processors, 1995.

[175] Intel. Intel 64 Architecture x2APIC Specification, 2008.

[176] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, 2016. Rev 1.5.

[177] Intel. L1 Terminal Fault SA-00161, 2018.

[178] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[179] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[180] Intel. Developer Reference for Intel Integrated Performance Primitives Cryptography, 2019.

[181] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[182] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[183] Intel. INTEL-SA-00219, 2019.

[184] Intel. Machine Check Error Avoidance on Page Size Change, 2019.

[185] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. *Developer Zone - Secure Coding*, 2020.

[186] Intel. Intel Debug Technology, 2021.

[187] Intel. Intel Trust Domain Extensions, 2021.

[188] Intel. 12th Generation Intel Core Processor Family, 2022.

[189] Intel. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance, 2022.

[190] Intel. Intel Integrated Performance Primitives Cryptography, 2022.

[191] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2022.

[192] Intel. Microarchitectural Data Sampling, 2022.

[193] T. Jaeger. Operating system security. *Synthesis Lectures on Information Security, Privacy and Trust*, 2008.

[194] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with Intel TSX. In *CCS*, 2016.

[195] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. HexType: Efficient detection of type confusion errors for C++. In *CCS*, 2017.

[196] Y. Jeon, W. Han, N. Burow, and M. Payer. Fuzzan: Efficient sanitizer metadata design for fuzzing. In *USENIX ATC*, 2020.

[197] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the Linux kernel. In *NDSS*, 2022.

[198] B. Johansson, P. Lantz, and M. Liljenstam. Lightweight dispatcher constructions for control flow flattening. In *SSPREW*, 2017.

[199] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. In *CAV*, 2009.

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

[201] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A. Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, 2020.

[202] H. Khattri, N. K. V. Mangipudi, and S. Mandujano. Hsdl: A security development lifecycle for hardware technologies. In *HOST*, 2012.

[203] T. Kim, M. Peinado, and G. Mainar-Ruiz. {STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*, 2012.

[204] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *WCRE*, 2012.

[205] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[206] J. Kirsch, C. Jonischkeit, T. Kittel, A. Zarras, and C. Eckert. Combating control flow linearization. In *ICT Systems Security and Privacy Protection*, 2017.

[207] O. Kirzner and A. Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security Symposium*, 2021.

[208] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *CCS*, 2018.

[209] A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3), 2008.

[210] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[211] A. Kogler, D. Weber, M. Haubenwallner, M. Lipp, D. Gruss, and M. Schwarz. Finding and Exploiting CPU Features using MSR Templating. In *S&P*, 2022.

[212] A. Koker, T. A. Piazza, and M. Sundaresan. Scatter/gather capable system coherent cache, 2016. US Patent 9,471,492 B2.

[213] B. Kollenda, P. Koppe, M. Fyrbiak, C. Kison, C. Paar, and T. Holz. An Exploratory Analysis of Microcode as a Building Block for System Defenses. In *ACM CCS*, 2018.

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

[215] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz. Reverse Engineering x86 Processor Microcode. In *USENIX Security Symposium*, 2017.

[216] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[217] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi. TagBleed: breaking KASLR on the isolated kernel address space using tagged TLBs. In *EuroS&P*, 2020.

[218] M. Krause. Watch Your Step(ping): Atoms Breaking Apart, 2021.

[219] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *EuroSys*, 2018.

[220] J. Kuderski, J. A. Navas, and A. Gurfinkel. Unification-based pointer analysis without oversharing. In *FMCAD*, 2019.

[221] T. Kurts, Z. Wayner, and T. Bojan. Apparatus and method for bus signal termination compensation during detected quiet cycle, 2007. US Patent 7,227,377 B2.

[222] D. Kwan, K. Shtoyk, K. Serebryany, M. L. Lifantsev, and P. Hochschild. SiliFuzz: Fuzzing CPUs by proxy. Google Research, 2021.

[223] A. Kwon, U. Dhawan, J. M. Smith, T. F. K. Jr, and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *CCS*, 2013.

[224] lafintel. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016. [Online; accessed 10 Sep. 2021].

[225] P. Larsen, S. Brunthaler, and M. Franz. Automatic software diversity. *IEEE Sec. and Priv.*, 2015.

[226] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *S&P*, 2014.

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

[228] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[229] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium*, 2015.

[230] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*, 2017.

[231] X. Leroy. How I found a bug in Intel Skylake processors, 2017.

[232] D. Levinthal. Performance analysis guide for intel core i7 processor and intel® xeon 5500 processors, 2009.

[233] W. Li. System-on-chip devices and methods for testing system-on-chip devices. Patent WO2017164872A1, 2017.

[234] M. Lipp, D. Gruss, and M. Schwarz. {AMD} prefetch attacks through power and time. In *USENIX Security Symposium*, 2022.

[235] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2021.

[236] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[237] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[238] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *CSF*, 2013.

[239] LLVM. UndefinedBehaviorSanitizer - Clang documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[240] LLVM. SanitizerCoverage - Edge coverage. https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage, 2021. [Online; accessed 10 Sep. 2021].

[241] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing., 2018.

[242] K. Lu, S. Xiong, and D. Gao. RopSteg: Program steganography with return oriented programming. In *CODASPY*, 2014.

[243] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[244] S. Luo. trapfuzzer: Coverage-guided Binary Fuzzing with Breakpoints. In *HITB SecConf*, 2021.

[245] A. Lutas and D. Lutas. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. In *BlackHat Europe*, 2019.

[246] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao. Software watermarking using return-oriented programming. In *ASIACCS*, 2015.

[247] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[248] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[249] V. J. M. Manès, S. Kim, and S. K. Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *ICSE*, 2020.

[250] H. Mantel and A. Starostin. Transforming out timing leaks, more or less. In *ESORICS*, 2015.

[251] A. Mantovani, A. Fioraldi, and D. Balzarotti. Fuzzing with data dependency information. In *EuroS&P*, 2022.

[252] M. Marlinspike. Technology preview: Private contact discovery for signal, 2017.

[253] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *ISCA*, 2012.

[254] I. A. Mason. Whole Program LLVM in Go. https://github.com/SRI-CSL/gllvm, 2021. [Online; accessed 2 Sep. 2021].

[255] P. Mavropoulos. CPUMicrocodes: Intel, AMD, VIA & Freescale CPU Microcode Repositories, 2022.

[256] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG*, 2001.

[257] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.

[258] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *FSE*, 2021.

[259] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA*, 2002.

[260] J. F. Misarsky. How (not) to Design RSA Signature Schemes. In *Public Key Cryptography*, 1998.

[261] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[262] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.*, 47(4):538?570, 2019.

[263] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar. Copycat: Controlled instruction-level attacks on enclaves. In *USENIX Security Symposium*, 2020.

[264] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*, 2020.

[265] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICASE*, 2005.

[266] M. Morbitzer, S. Proskurin, M. Radev, and M. Dorfhuber. Severity: Code injection attacks against encrypted virtual machines. In *WOOT*, 2021.

[267] D. Mu, J. Guo, W. Ding, Z. Wang, B. Mao, and L. Shi. ROPOB: Obfuscating Binary Code via Return Oriented Programming. In *Security and Privacy in Communication Networks*, 2018.

[268] T. Müller, F. C. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium*, 2011.

[269] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert. CastSan: Efficient detection of polymorphic C++ object type confusions with LLVM. In *ESORICS*, 2018.

[270] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[271] R. Muth, S. Watterson, and S. Debray. Code specialization based on value profiles. In *Static Analysis*, 2000.

[272] S. Nagy and M. Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *S&P*, 2019.

[273] Z. L. Nemeth. Modern binary attacks and defences in the Windows environment – fighting against Microsoft EMET in seven rounds. In *SISY*, 2015.

[274] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[275] NSA. Ghidra. https://ghidra-sre.org/. Online; accessed 11 June 2020.

[276] C. Ntantogian, G. Poulios, G. Karopoulos, and C. Xenakis. Transforming malicious code to rop gadgets for antivirus evasion. *IET Inform. Security*, 13(6):570–578, 2019.

[277] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis*, 2004.

[278] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion. How to kill symbolic deobfuscation for free (or: Unleashing the potential of path-oriented protections). In *ACSAC*, 2019.

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

[280] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium*, 2020.

[281] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[282] R. Paccagnella, L. Luo, and C. W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security Symposium*, 2021.

[283] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[284] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. *ACM SIGOPS Operating Systems Review*, 2008.

[285] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *USENIX Security Symposium*, 2018.

[286] T. Palit, J. Firose Moon, F. Monrose, and M. Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *S&P*, 2021.

[287] C. Pang, Y. Du, B. Mao, and S. Guo. Mapping to bits: Efficiently detecting type confusion errors. In *ACSAC*, 2018.

[288] PaX Team. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt, 2016. Online; accessed 11 May 2020.

[289] M. Payer. The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes. *IEEE Security and Privacy*, 2019.

[290] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-executing binary programs for security applications. In *USENIX Security Symposium*, 2014.

[291] H. Peng and M. Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *USENIX Security Symposium*, 2020.

[292] Phoronix. Google publishes latest linux core scheduling patches so only trusted tasks share a core, 2020.

[293] S. Poeplau and A. Francillon. Symbolic execution with symcc: Don't interpret, compile! In *USENIX Security Symposium*, 2020.

[294] M. Polychronakis and A. D. Keromytis. ROP payload detection using speculative code execution. In *MALWARE*, 2011.

[295] M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *WCRE*, 2002.

[296] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *AsianHOST*, 2019.

[297] Qualcomm Technologies Inc. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions, 2017.

[298] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security Symposium*, 2021.

[299] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, 2021.

[300] G. Ramalingam. The undecidability of aliasing. *ACM Trans. on Prog. Lang. and Sys.*, 16(5):1467–1471, 1994.

[301] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, 2015.

[302] J. Ravichandran, W. T. Na, J. Lang, and M. Yan. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *ISCA*, 2022.

[303] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[304] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet. Get rid of inline assembly through verification-oriented lifting. In *ICASE*, 2019.

[305] RISC-V. RISC-V "V" vector extension, 2019.

[306] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *CC*, 2016.

[307] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. on Inf. and Sys. Sec.*, 15(1), 2012.

[308] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL*, 1988.

[309] J. Salwan, S. Bardin, and M. Potet. Symbolic deobfuscation: From virtualized code back to the original. In *MALWARE*, 2018.

[310] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computer Surveys*, 49(1), 2016.

[311] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium*, 2017.

[312] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *S&P*, 2015.

[313] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[314] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[315] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[316] M. Schwarz, S. Weiser, and D. Gruss. Practical enclave malware with Intel SGX. In *MALWARE*, 2019.

[317] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[318] M. Schwarzl, C. Canella, D. Gruss, and M. Schwarz. Specfuscator: Evaluating Branch Removal as a Spectre Mitigation. In *FC*, 2021.

[319] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[320] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[321] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *WBIA*, 2009.

[322] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX ATC*, 2005.

[323] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[324] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[325] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *S&P*, 2009.

[326] Z. Shen, R. Roongta, and B. Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *USENIX Security Symposium*, 2022.

[327] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with o((logN)3) worst-case cost. In *In: Lee D.H., Wang X. (eds) Advances in Cryptology – ASIACRYPT 2011. Lecture Notes in Computer Science, vol 7073*. Springer Berlin Heidelberg, 2011.

[328] O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages of Taming Lambda*. Carnegie Mellon University, 1991.

[329] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *S&P*, 2016.

[330] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, 2015.

[331] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2011.

[332] L. Soares and F. M. Q. Pereira. Memory-safe elimination of side channels. In *CGO*, 2021.

[333] J. Somorovsky. Systematic fuzzing and testing of tls libraries. In *CCS*, 2016.

[334] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-OS boundary. In *NDSS*, 2019.

[335] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *USENIX Security Symposium*, 2020.

[336] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *S&P*, 2019.

[337] M. Sridharan and S. J. Fink. The complexity of andersen's analysis in practice. In *Static Analysis*, 2009.

[338] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[339] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS*, 2013.

[340] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[341] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[342] S. R. Subramanya and B. K. Yi. Digital rights management. *IEEE Potentials*, 25(2):31–34, 2006.

[343] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, 2003.

[344] Y. Sui and J. Xue. SVF: interprocedural static value-flow analysis in LLVM. In *CC*, 2016.

[345] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *ICSE*, 2010.

[346] J. Sutherland, N. Coull, and A. MacLeod. CPU covert channel accessible from JavaScript. *CyberForensics*, 2014.

[347] J. A. Sutton. Microcode Patch Authentication. Patent US20030196096A1, 2003.

[348] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[349] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *USENIX Security Symposium*, 2018.

[350] Terenceli. Local APIC virtualization, 2018.

[351] The MITRE Corporation. Cwe version 4.6, 2021.

[352] D. Thomas and P. Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[353] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *USENIX Security Symposium*, 2014.

[354] B. Trower. base64. http://base64.sourceforge.net/b64.c, 2001.

[355] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *WCRE*, 2005.

[356] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Rambo: Run-time packer analysis with multiple branch observation. In *MALWARE*, 2016.

[357] U.S. National Security Agency. Commercial national security algorithm suite and quantum computing FAQ, 2016.

[358] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[359] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[360] J. Van Bulck, F. Piessens, and R. Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX*, 2017.

[361] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[362] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[363] E. Van Der Kouwe, V. Nigade, and C. Giuffrida. Dangsan: Scalable use-after-free detection. In *EuroSys*, 2017.

[364] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *CCS*, 2015.

[365] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *RAID*, 2012.

[366] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[367] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. Sgaxe: How sgx fails in practice, 2020.

[368] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[369] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on intel cpus via cache evictions, 2020.

[370] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, 2011.

[371] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *NDSS*, 2014.

[372] Voss, Nathan. afl-unicorn: Fuzzing Arbitrary Binary Code, 2017.

[373] D. Vyukov. syzkaller - kernel fuzzer. [Online; accessed 10 Sep. 2021].

[374] A. Wailly, A. Souchet, J. Salwan, A. Verez, and T. Romand. Automated Return-Oriented Programming Chaining. https://github.com/awailly/nrop, 2014. Online; accessed 11 June 2020.

[375] J. Wampler, I. Martiny, and E. Wustrow. Exspectre: Hiding malware in speculative execution. In *NDSS*, 2019.

[376] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of software-based survivability mechanisms. In *DSN*, 2001.

[377] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *RAID*, 2019.

[378] S. Wang, P. Wang, and D. Wu. UROBOROS: instrumenting stripped binaries with static reassembling. In *IEEE 23rd Int. Conf. on Soft. Analysis, Evol., and Reengineering*, 2016.

[379] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When benign apps become evil. In *USENIX Security Symposium*, 2013.

[380] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, 2013.

[381] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.

[382] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *MICRO*, 2008.

[383] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[384] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*, 2016.

[385] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Security Symposium*, 2018.

[386] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018.

[387] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[388] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *ASPLOS*, 2020.

[389] M. Wu, S. Guo, P. Schaumont, and C. Wang. Eliminating timing side-channel leaks using program repair. In *ISSTA*, 2018.

[390] D. Xu, J. Ming, Y. Fu, and D. Wu. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *CCS*, 2018.

[391] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *CCS*, 2015.

[392] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *S&P*, 2015.

[393] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[394] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[395] Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: A timing attack on openssl constant time rsa. In *CHES*, 2016.

[396] Y. Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[397] T. Yu and O. Kaser. A note on "on the conversion of indirect to direct recursion". *ACM Trans. Program. Lang. Syst.*, 19(6):1085–1087, 1997.

[398] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, 2018.

[399] M. Zalewski. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed 10 Sep. 2021].

[400] M. Zalewski. American Fuzzy Lop, 2021.

[401] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *CCS*, 2011.

[402] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[403] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *S&P*, 2011.

[404] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

[405] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang. StateFuzz: System call-based state-aware Linux driver fuzzing. In *USENIX Security Symposium*, 2022.

[406] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *ASE*, 2020.

[407] C. Zou, Y. Sui, H. Yan, and J. Xue. TCD: Statically detecting type confusion errors in C++ programs. In *ISSRE*, 2019.