Compiler-based Defenses against Code Execution Attacks

Dissertation

Markus Bauer

Saarland University

Department of Computer Science

2022

Saarland University
Department of Computer Science

Compiler-based Defenses against Code Execution Attacks

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Markus Bauer

Saarbrücken, 2022

Tag des Kolloquiums:	11.01.2024
Dekan:	Prof. Dr. Jürgen Steimle

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Sebastian Hack
Berichterstattende:	Prof. Dr. Christian Rossow
	Dr. Michael Schwarz
Akademischer Mitarbeiter:	Dr. Jordan Samhi

Zusammenfassung

Seit Jahrzehnten nutzen Angreifer Schwachstellen in der Speicherverwaltung nativer Programme aus. Mittels dieser Fehler korrumpieren sie Daten, führen beliebigen Code aus, und übernehmen angegriffene Systeme komplett. Besonders C- und C++-Anwendungen sind gefährdet.

In dieser Dissertation beschreiben wir Compiler-basierte Lösungen, die bestehende Anwendungen schützen, ohne dass Entwickler dafür Code umschreiben oder viel Zeit aufwenden müssen. Alle gefährdeten Funktionen werden bedacht: Rücksprünge, indirekte Sprünge (in C und C++) sowie unveränderbare Libraries. Erstens evaluieren wir bestehende Schutzmaßnahmen für Rücksprungaddressen. Wir zeigen, dass viele Argumente gegen den Einsatz dieser Techniken auf modernen Systemen nicht mehr relevant sind, und dass bestehende Lösungen bereits eingesetzt werden können. Zweitens schützen wir virtuelle Funktionsaufrufe in C++-Anwendungen. Wir nutzen eine Typ-basierte Analyse und eine Transformation im Compiler, um diese Aufrufe effizient und ohne Funktionspointer zu implementieren. Drittens schützen wir indirekte Aufrufe von Funktionsadressen in C. Wir nutzen eine neue, typ-basierte Analyse um mögliche Aufrufziele zu finden und die Menge gültiger Sprungziele zu minimieren. Zuletzt zeigen wir eine Methode, um möglicherweise verwundbaren Code, beispielsweise ungeschützte Libraries ohne zugänglichen Quellcode, in einer isolierten Umgebung auszuführen.

// Abstract

Memory corruption attacks have haunted computer systems for decades. Attackers abuse subtle bugs in an application's memory management, corrupting data and executing arbitrary code and, consequently, taking over systems. In particular, C and C++ applications are at risk, while developers often fail or lack time to identify or rewrite risky parts of their software.

In this thesis, we approach this problem with compilers that protect applications without requiring code changes or developer effort. We cover all treated aspects in legacy applications: returns, indirect forward jumps in both C and C++, and immutable libraries. First, we re-evaluate existing return address protections. In particular, we show that most adaption-preventing arguments have become less critical in the modern world and that already existing solutions can be deployable in production. Second, we protect virtual dispatch in C++ applications from hijacking. We employ a type analysis and a compiler transformation that implements virtual dispatch efficiently without hijackable pointers. Third, we protect indirect calls to function pointers in C applications. We use a new type-based analysis to find indirect call targets and transform indirect calls into a secure and fast version with limited targets. Finally, we propose a method to isolate potentially vulnerable code, particularly unprotected closed-source libraries, into compartments with restricted access to its environment.

Background of this Dissertation

This dissertation is based on three peer-reviewed papers that are published at IEEE EuroS&P 2021 [P1], ACM ACSAC 2022 [P2], and ACM Asia CCS 2021 [P3]. I contributed to all papers as the only author at PhD student level, and two of them [P1, P3] even as a single person next to faculty-level co-authors. These papers form Chapter 5 [P1], Chapter 6 [P2], and Chapter 7 [P3] of this thesis.

[P1] **M. Bauer** and C. Rossow. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In: *2021 IEEE European Symposium on Security and Privacy*. EuroS&P '21. Sept. 2021. DOI: 10.1109/EuroSP51992.2021.00049.

[P2] **M. Bauer**, I. Grishchenko, and C. Rossow. TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation. In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC '22. Dec. 2022. DOI: 10.1145/3564625.3564627.

[P3] **M. Bauer** and C. Rossow. Cali: Compiler Assisted Library Isolation. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. May 2021. DOI: 10.1145/3433210.3453111.

Further Contributions of the Author

I also contributed to one additional paper [S1] as one of the main authors. Switchpoline, the presented tool, protects C and C++ applications from Spectre-BTB attacks on ARM devices. To this end, it builds application binaries that do not contain any indirect branch instruction. It is based on NoVT [P1] and TyPro [P2]. The paper is not part of this thesis, Section 6.12.1 summarizes its content.

[S1] **M. Bauer**, L. Hetterich, C. Rossow, and M. Schwarz. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In: *Proceedings of the 2024 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. 2024.

Acknowledgments

This dissertation would have been impossible without the motivation, support, and company of many great people. I want to thank everybody who supported me during my studies, my work, and the preparation of my dissertation.

First and foremost, I want to thank my supervisor Christian Rossow for being a great mentor, for giving me the opportunity to do exciting projects, for the funding, for giving me the freedom to explore, and for having trust in me. I am deeply grateful for the introduction and support in academic writing and the great introductions in all our publications. Special thanks also go to Michael Schwarz for the time and effort in reviewing my dissertation. I want to give a special thank you to my co-authors, namely Ilya Grishchenko, Lorenz Hetterich, and Michael Schwarz. I am grateful for your ideas, the discussions, and the work you put in our joined publications. It has been a pleasure to work with you.

My deepest gratitude goes to my colleagues. First and foremost, I want to thank Fabian Schwarz for being such a nice office mate over the whole time, for all the talks, and for being able to share all the frustration that comes with technical work. I want to thank Johannes Krupp for his help in structuring and preparing this dissertation. Special thanks go to all members of our System Security Group: Benedikt Birtel, Michael Brengel, Jonas Bushart, Ahmad Ibrahim, Giorgi Maisuradze, Giancarlo Pellegrino, and Leon Trampert. I thank you for the endless discussions during lunch, the constant stream of ideas, for testing the CALI prototype, and for the company over the last six years. I also want to express my gratitude towards all CISPA employees for providing such a pleasant working atmosphere.

Solving new problems is impossible without a solid foundation and rich background knowledge. During my studies at Saarland University, I found many lecturers that put so much effort into their courses to give us students the best possible learning opportunity. Thank you all for providing such high-quality courses and teaching me that computer science is more than just programming. I also want to thank all members of our CTF team saarsec, particularly Ben Stock, who introduced me to the secrets of attack-defense competitions. I learned much about security and systems during the saarsec meetings, playing competitions, developing new tools, and during infrastructure preparation. The skills I learned here greatly supported my research and my future career. Thank you all for being such a fantastic team.

I want to thank my Bachelor's thesis advisor Sebastian Meiser who brought me on the path toward a doctoral degree. Finally, I want to thank Michelle Carnell from the Graduate School of Computer Science for rescuing my promotion studies twice.

I want to thank my family for their strong support, especially my parents, Karin and Manfred. I am grateful for all the times you helped me in many ways. You always had my back and supported me wherever possible. This dissertation would have been impossible without you. I also want to thank my brother Michael and my grandparents for their support and belief in me. I want to thank my friends for always supporting me, cheering me up, and for being there when I needed them.

Contents

1 Introduction **1**

2 Background **9**
2.1 C and C++ from a security perspective . . . 11
2.2 Memory Corruption Vulnerabilities and Attacks . . . 12
2.3 Attacker Model . . . 14
2.4 The LLVM Compiler Framework . . . 14
2.4.1 Clang . . . 15
2.4.2 Linkers and Link-Time-Optimization. . . . 15
2.4.3 Multi-Module Programs. . . . 17

3 Related Work **19**
3.1 Preconditions for Code Execution Attacks . . . 21
3.2 Separating Code from Data . . . 21
3.3 Preventing Memory Corruption . . . 22
3.4 Protecting Code Pointers (Code Pointer Integrity) . . . 23
3.5 Hiding Gadgets . . . 24
3.6 Control Flow Integrity . . . 25
3.6.1 Backward CFI . . . 25
3.6.2 Forward CFI . . . 26
3.7 Isolation and Compartmentalization . . . 26

4 SoK: Evaluation of Return Address Protections **29**
4.1 Motivation . . . 31
4.2 Problem Description . . . 31
4.3 Contributions . . . 31
4.4 Background: Attacks on Return Addresses . . . 32
4.5 Backward CFI schemes . . . 32
4.5.1 Stack Canaries . . . 33
4.5.2 Shadow Stacks . . . 33
4.5.3 Return Address Encryption . . . 34
4.6 The SPEC CPU Benchmark Suite . . . 35
4.7 Implementation . . . 36
4.7.1 Stack Canaries . . . 36
4.7.2 Shadow Stack . . . 36
4.7.3 Return address encryption . . . 37

4.7.4 Optimizations . . . 38
4.7.5 Evaluating Implementations . . . 39
4.7.6 Evaluating Security . . . 41
4.8 Performance Evaluation . . . 43
4.8.1 Methodology . . . 43
4.8.2 Measurement Soundness . . . 44
4.8.3 Performance Overhead on SPEC . . . 46
4.8.4 Overhead per programming language . . . 49
4.8.5 Overhead Comparison With Literature . . . 50
4.8.6 Binary Size Overhead . . . 52
4.9 Compatibility Evaluation . . . 52
4.10 Excursion: Return Address Protections on ARM . . . 54
4.11 Conclusion . . . 54

5 NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking 57
5.1 Motivation . . . 59
5.2 Problem Description . . . 59
5.3 Contributions . . . 60
5.4 Background . . . 61
5.4.1 C++ Inheritance and Vtables . . . 61
5.4.2 C++ Multiple Inheritance . . . 61
5.4.3 C++ Virtual Inheritance . . . 63
5.4.4 Vtable Hijacking . . . 64
5.5 Attacker Model . . . 65
5.6 Design and Implementation . . . 65
5.6.1 Class Hierarchy Analysis . . . 65
5.6.2 Class Identifiers . . . 66
5.6.3 Dispatch Function Generation . . . 67
5.6.4 Storing Class IDs and Removing Vtables . . . 69
5.6.5 Optimizations . . . 69
5.6.6 Implementation . . . 71
5.6.7 Compiler-Assisted Optimizations . . . 72
5.6.8 Usability . . . 73
5.7 Evaluation . . . 73
5.7.1 Security Evaluation . . . 73
5.7.2 Runtime Evaluation . . . 76
5.7.3 Generated Code Evaluation . . . 80
5.7.4 Binary Size and Memory Overhead . . . 81
5.8 Compatibility and Limitations . . . 82
5.9 Related Work . . . 83
5.9.1 Attacks on Vtables . . . 83
5.9.2 Vtable Protections . . . 83
5.9.3 Alternatives to Vtables . . . 85
5.9.4 Replacing Pointers with Identifiers . . . 86
5.10 Conclusion . . . 86

6 TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation 89
6.1 Motivation . . . 91
6.2 Problem Description . . . 91
6.3 Contributions . . . 92
6.4 Overview . . . 93
6.4.1 Attacker Model . . . 93
6.4.2 Challenges . . . 93
6.4.3 Methodology at a Glance . . . 94
6.4.4 Type Propagation vs. Data Flow . . . 96
6.5 Target Set Computation . . . 96
6.5.1 Analysis Input Generation . . . 98
6.5.2 Type Analysis . . . 101
6.6 Call Target Enforcement . . . 105
6.7 Dynamic Modules . . . 106
6.7.1 Additional Input Generation . . . 107
6.7.2 Additional Type Analysis . . . 107
6.7.3 Dynamic Call Target Enforcement . . . 108
6.8 Implementation . . . 109
6.8.1 C Standard Libraries . . . 110
6.8.2 Optimizations . . . 110
6.9 Evaluation . . . 111
6.9.1 Correctness . . . 112
6.9.2 Security . . . 112
6.9.3 Performance . . . 114
6.9.4 Dynamic Loading . . . 116
6.10 Limitations & Discussion . . . 117
6.11 Related Work . . . 118
6.12 Conclusion . . . 119
6.12.1 Future Work—Switchpoline . . . 119

7 Cali: Compiler-Assisted Library Isolation 121
7.1 Motivation . . . 123
7.2 Problem Description . . . 123
7.3 Contributions . . . 124
7.4 Background and Related Work . . . 125
7.4.1 Compartmentalization . . . 125
7.4.2 Isolation Primitives . . . 127
7.5 General Overview . . . 128
7.5.1 Compiler-Assisted Library Isolation . . . 128
7.5.2 Overview . . . 128
7.6 Shielding Compartments . . . 129
7.6.1 Basic Compartment Structure . . . 129
7.6.2 Shared Memory . . . 129
7.6.3 Library Calls . . . 130
7.6.4 Callbacks, Signals, and File Descriptors . . . 130

7.6.5 Isolation . . . 131
7.6.6 Threading, Forks, and Concurrency . . . 131
7.7 Compiler-Assisted Separation . . . 132
7.7.1 Background: Call Graphs and SCCs . . . 133
7.7.2 Analysis Phase: Overview . . . 133
7.7.3 PDG Construction . . . 135
7.7.4 Data Flow in PDGs . . . 137
7.7.5 Reachability Analysis . . . 138
7.7.6 Function Specialization . . . 141
7.7.7 Tracing File Descriptors . . . 142
7.7.8 Rewriting Memory Allocations . . . 142
7.7.9 Data-Transferring Call Graph Analysis . . . 142
7.8 Evaluation . . . 143
7.8.1 Correctness Evaluation . . . 143
7.8.2 Usability Evaluation . . . 144
7.8.3 Compilation and Size Overhead . . . 146
7.8.4 Security Evaluation . . . 146
7.8.5 Performance Evaluation . . . 149
7.9 Conclusion . . . 150

8 Conclusion **153**

Bibliography **159**

List of Tables

4.1 Overview of SPEC 2017 benchmarks, their size, the protection coverage of different canary modes, and coverage of our optimization strategy. Additional summary of code locations where scratch registers were detected. 40
4.2 Overhead of all protections on all benchmarks. 47
4.3 Mean overhead of all protections per language and summary of overheads. . 51
4.4 Mean binary size overhead of all protections per language and summary of additional binary size. 52

5.1 Number of protected operations per benchmark. 79
5.2 Summary of assembly constructs used to build virtual function dispatchers. . 80
5.3 Size of binaries before and after protection. 81
5.4 Related work to NoVT, grouped by binary- and source-based solutions. 84

6.1 Average number of call targets per indirect call on SPEC. TyPro compared to Clang CFI, IFCC, CFGuard and MCFI. 113
6.2 Average number of call targets per indirect call on various real-world server applications. 113

7.1 Remaining shared memory allocations and the number of specialized functions in the main program. 146

List of Algorithms

1 Class identifier assignment algorithm from NoVT. 71

List of Figures

2.1 Compilation and Linking of C/C++ programs with the Clang/LLVM toolchain. 16

4.1 Stack layout of C/C++ programs and the impact of buffer overflows. 32
4.2 Stack layout with canary. 33
4.3 Stack layout with shadow stack. 34
4.4 Stack layout with return address encryption. 35
4.5 Benchmark results and clusters of *omnetpp* and *xalancbmk* 45
4.6 Overhead of all protections on all benchmarks. 46
4.7 Overhead of all relaxed and optimized protections on all benchmarks. 48
4.8 Overhead per programming language. 50

5.1 C++ code as running example with multiple and virtual inheritance 62
5.2 Class hierarchy graph for our running example (Figure 5.1). 62
5.3 Memory layout of our example classes according to the Itanium ABI. 63
5.4 Class identifier graph including all possible construction identifiers. 68
5.5 Virtual dispatchers broken down by their number of possible call targets. . . . 74
5.6 Performance overhead of NOVT on all programs. 77
5.7 Performance overhead of NOVT on Chromium. 78
5.8 Virtual actions (calls, vbase offset, etc) per second, broken down by the number of `switch` cases in the NOVT dispatcher functions. 78

6.1 Code example showing different ways to transfer function pointers. 94
6.2 TYPRO's workflow. 95
6.3 Graphical representation of the collected and derived facts for "`scene1_a`" and "`scene1_b`". 97
6.4 Graphical representation of the collected and derived facts for "`scene2_a`" and "`scene2_b`". 97
6.5 Graphical representation of the collected and derived facts for "`scene3_a`" and "`scene3_b`". 98
6.6 Predicate signatures for the facts used by analysis. 98
6.7 Rules for computing the final result (all possible function types for each call). 103
6.8 Rules demonstrating the core of Clang CFI and IFCC computation of the final function types for each call. 103
6.9 Example showing a simple indirect call before and after transformation. . . . 106
6.10 Additional predicate definitions for dynamic module support, and additional rules for module summaries. 107
6.11 Runtime overhead of TYPRO on SPEC 2006 benchmarks. 115
6.12 Runtime overhead of TYPRO on real-world applications. 115
6.13 Additional size of SPEC and other example programs on x86 (in KB). 116
6.14 Runtime computation time for dynamic linking (SPEC / musl libc) 116

7.1 Overview of recent program isolation schemes. 127
7.2 Example program passing memory to a library. 134
7.3 Simplified LLVM code of the example in Figure 7.2. 135
7.4 Excerpt of the PDG from the example program. 136
7.5 Recursive algorithm to generate subnodes of a value node n with type t. . . . 136
7.6 Rules to determine data flow for LLVM instructions. 137
7.7 Full Program Dependence Graph with all analyses applied. 139
7.8 Configuration file for ImageMagick `convert`. 145
7.9 Compile time without and with CALI. 146
7.10 Performance impact of CALI. 148

1
Introduction

The most popular programs are still written in memory-unsafe languages like C and C++ [84]. From browsers over operating systems to end-user applications, today's computer systems rely on massive amounts of C and C++ code. Errors in this code can easily open up *memory corruption vulnerabilities* that can enable *code execution* attacks. Attackers can corrupt the entire system, executing arbitrary code, possibly even over the internet. And because C and C++ are low-level languages without any sophisticated memory protection, these programs are particularly prone to memory corruption attacks. In the real world, we see plenty of these attacks, for example, on operating systems [103, 72, 162], browsers [167], and servers. The underlying vulnerabilities are common—for example, 70% of all vulnerabilities in Google Chrome are memory safety problems [167], around 80% of all Android vulnerabilities are caused by memory issues [162], and use-after-free heap corruption bugs are the primary source of vulnerabilities in Microsoft Windows [103].

This dissertation will present and discuss methods to protect applications from code execution attacks. While we do not prevent memory corruption vulnerabilities, we can prevent the escalation to code execution and system corruption.

In an ideal world, developers would solve this problem by rewriting applications in memory-safe languages like Rust [147]. These languages should not have memory corruption vulnerabilities by design, preventing the root cause of code execution attacks. However, the sheer amount of code that is currently in use makes rewriting a tedious, longterm (if not impossible) task. While more and more developers consider Rust a potential language for new code, existing code is often left untouched [25]. Unsafe legacy C/C++ code will be around for the decades to come, and with it, the risk of code execution attacks.

For years, researchers have identified the need for protection against this problem, leading to an arms race between attackers and defenders. Automatic, program-agnostic protections have been proposed; they try to protect legacy applications from different types of code execution attacks. For example, *data execution prevention* (DEP) [102] or W⊕X [188] prevent *code injection* attacks, *address space layout randomization* (ASLR) [137] hides crucial information from the attacker, and *stack canaries* [37] prevent attacks based on memory corruption from stack buffer overflows. Built into major C/C++ compilers, applying these defenses is simple for programmers; they require no changes in the source code. Following, these defenses have seen widespread adoption in real-world programs. By now, they are even enabled by default in many popular compilers.

But attackers have reacted to the emerging spread of these defenses, and have developed newer, stronger attacks countering many existing defenses: *code-reuse attacks* have replaced code injection attacks, *information leaks* defeat ASLR, and heap-based memory corruption evades stack-based protections like stack canaries. Stronger protections are necessary that cover the missing parts of C/C++ applications.

This dissertation will present such stronger protections and how we can isolate the effects of code execution attacks. To this end, we will answer two research questions (RQ):

(RQ1) How can we prevent the escalation from memory corruption to attacks on an application's control flow? Attacks on the control flow are the crucial step of code execution attacks. By bending an application's control flow, attackers can ultimately determine which code will execute next, which usually results in a complete take-over of the application process. In Chapter 2, we will detail the background of these attacks more. In Chapter 3, we will review the plethora of existing work on this topic. We will see that

many defenses have been proposed, but only a few have been actually deployed in practice, and clever attackers can circumvent most deployed solutions. From the related work, we conclude that *control flow integrity* (CFI) systems are the best possibility to prevent code execution attacks today, either in software or hardware.

CFI schemes must protect backward control flow (returns) and forward control flow (indirect calls). So far, backward CFI schemes exist, but they are considered slow or weak. We re-evaluate three of these schemes in Chapter 4 in a modern setting and show that their performance overhead is much lower than expected. In particular, we show that strong and fast schemes exist, and they might see real-world adoption in the coming years.

Protecting forward control flow is still an ongoing topic in research. Current schemes are either complex, slow or require code modifications. In Chapter 5, we present NoVT, a forward CFI scheme for C++ applications. NoVT protects existing code without any modifications, additional complexity, or performance overhead. In Chapter 6, we present a similar system for C applications: TyPro. TyPro protects C-style indirect calls with little to no performance cost. Together, NoVT and TyPro cover forward control flow completely.

However, even a perfect CFI scheme might not be enough to prevent code execution attacks completely in every scenario. In particular, we identified third-party libraries as a threat to security. Not only have bugs in application dependencies been the root cause for many attacks in the real world, but state-level attackers have successfully executed *supply chain attacks*. Attackers have submitted intentionally vulnerable or malicious code to third-party libraries used by their victims. If the victim does not detect the evil nature of these contributions, it includes malicious code in the application, which becomes an easy target for the attackers. But libraries impose a security risk even if they do not contain malicious code. If source code is unavailable, the proposed CFI schemes cannot protect the library, and memory corruption attacks on the library's code can take over the whole application process. And some libraries might be old, unmaintained, or known for the regular discovery of vulnerabilities.

Application maintainers can hinder but hardly prevent code execution attacks originating in library code. In these scenarios, there is a need for additional protection, which we will research in our second research question:

(RQ2) How can we isolate application memory and underlying system from attackers that exploit a code execution vulnerability in a library? While the operating system's user and permission system can already limit system corruption to a single user account, we want to limit libraries further: In Chapter 7, we will isolate parts of the application, e.g., libraries, minimizing their system privileges and, thus, possible corruptions in case of code execution attacks. Developers can use Cali, our protection, with minimal effort to isolate third-party libraries or other untrusted components from their main application. The isolated code can only access files, networks, or other resources if explicitly permitted by the developer; thus, the impact of an attack on the system is minimized. With a proper isolation policy, developers can even limit the damage of potential supply chain attacks in their dependencies. Isolation can also include closed-source libraries where CFI is not applicable. Cali thus completes the protection of CFI-protected applications with unprotected modules.

We publish all our protections as open-source software.

Contributions

This dissertation comprises three publications [P1, P2, P3] and one still unpublished report, outlined below.

Evaluation of Existing Return Address Protections (RQ1)

In "SoK: Evaluation of Return Address Protections" (see Chapter 4), we evaluate the performance, security, and compatibility of three return address protection schemes in a modern setting. So far, researchers have believed that these protections are either weak or too slow for widespread real-world adoption.

We re-implement stack canaries, shadow stacks, and return address encryption and re-evaluate these protections on modern hardware, software, and compilers. We show that their performance overhead has degraded well over time—up to a point where it might not prevent wider real-world adoption. In particular, a software-only shadow stack has a mean performance overhead of 2.7% if optimized for speed, so it is faster and more secure than stack canaries on every function. We further highlight the impact of programming languages on the expected performance overhead—C, C++, and Fortran are differently affected.

Protecting Virtual Dispatch in C++ Programs (RQ1)

In "NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking" [P1] (see Chapter 5), we protect C++ applications against vtable hijacking attacks.

All major C++ compilers use virtual function tables (vtables) to implement class inheritance and dispatch of overridden methods. Attackers can corrupt pointers to these vtables in memory to get code execution. Our solution replaces vtables with switch-case constructs that are inherently control-flow safe, thus preserving the control flow integrity of virtual dispatch.

To this end, NoVT extends the Clang C++ compiler to perform a class hierarchy analysis on C++ source code. This lightweight static analysis can infer which method implementations are valid targets for a virtual call. Instead of a vtable, we give each class a unique identifier number that is used to dispatch the correct method implementation. As a bonus, NoVT inherently protects all usages of a vtable, not just virtual dispatch.

We evaluate NoVT on the SPEC CPU 2006 benchmark and real-world programs, including Chromium. Despite its strong security guarantees, NoVT improves runtime performance of most programs (mean overhead -0.5%, -3.7% min, 2% max). In addition, protected binaries are slightly smaller than unprotected ones. NoVT works on different CPU architectures. It protects complex C++ programs against attacks that have been used to defeat previous, weaker solutions.

Protecting Indirect Calls in Legacy C Programs (RQ1)

In "TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation" [P2] (see Chapter 6), we focus on legacy C applications: We protect function pointers from arbitrary modifications, thus enforcing forward control flow integrity over the whole program, including libraries.

To protect indirect calls to function pointers, we must know which functions a pointer might point to at runtime. Then we can restrict indirect calls to target these functions only. Existing CFI schemes are either too permissive (thus weakening security guarantees) or too strict (thus breaking compatibility and introducing crashes). Furthermore, many existing schemes depend on specific hardware capabilities that are not generally available.

Our solution TYPRO solves both issues. It uses a new static analysis, called type propagation, to follow function pointer types through C programs. Type propagation can determine the possible target functions for indirect calls at compile time with high precision. We will show that TYPRO does not underestimate possible targets and does not break real-world programs, including those relying on dynamically-loaded code.

Instead of checking indirect call targets at runtime, TYPRO converts indirect calls into a set of direct calls, similar to NOVT. Instead of function pointers, TYPRO uses unique IDs to trigger the correct direct call. This transformation is implemented in the compiler; it is software-only and fast: TYPRO has no runtime overhead on average, both in benchmarks and real-world applications. It does not depend on special hardware features and works on different CPU architectures.

So far, TYPRO provides a good trade-off between compatibility, reliability, and security. These properties make TYPRO ideal for real-world adoption.

Defending Applications against Vulnerabilities in Libraries (RQ2)

Even though the proposed protections counter code execution attacks in many legacy programs, an attacker might find ways to break through: Most prominently, code not covered by the proposed protections could be exploited, like libraries without accessible source code. Software libraries can freely access the program's entire address space and inherit its system-level privileges. This lack of separation regularly leads to security-critical incidents once libraries contain vulnerabilities or turn rogue.

In "CALI: Compiler-Assisted Library Isolation" [P3] (see Chapter 7), we propose library isolation to solve this problem: CALI can *automatically* isolate unprotected or potentially risky parts of the overall application into restricted compartments. While this does not prevent the actual exploit, it prevents attackers from damaging the system. CALI is fully compatible with mainline Linux and does not require supervisor privileges to execute. We compartmentalize libraries into their own process and kernel namespace context with well-defined security policies. To preserve the functionality of the interactions between program and library, CALI uses a Program Dependence Graph to track data flow between the program and the library during link time.

We evaluate our open-source prototype against three popular libraries: *Ghostscript*, *OpenSSL*, and *SQLite*. CALI successfully reduced the amount of memory that is shared between the program and library to 0.08%–0.4% and retained an acceptable program performance.

Outline

The remainder of this dissertation is structured as follows: We first introduce the concepts we based our work on in Chapter 2: the C/C++ languages, memory corruption vulnerabilities, and the LLVM compiler framework. In Chapter 3, we summarize the most important related work on these topics. We will discuss topic-specific related work in the appropriate chapter. In Chapter 4, we re-evaluate existing protections for return addresses. Next, we discuss our own three protections: the C++ protection NoVT in Chapter 5, the function pointer protection TyPro in Chapter 6 and the library isolation tool CALI in Chapter 7. Finally, in Chapter 8, we summarize our results and give an outlook on future work.

2
Background

This chapter will provide the necessary background information about concepts, programming languages, and tools that we will use in later chapters. This thesis considers applications written in C/C++ (Section 2.1), some of the most common programming languages. Subtle programming errors in C/C++ can easily impose memory corruption vulnerabilities (Section 2.2) that attackers can abuse to run arbitrary code. We define the capabilities of such an attacker in Section 2.3. Our defenses against this threat rely on the LLVM compiler framework and its Clang compiler, which we introduce in Section 2.4.

2.1 C and C++ from a security perspective

C is a low-level, imperative, statically typed language focusing on efficiency. Since 1972, C has been widely used in systems programming, including operating systems. C requires programmers to manage the application's memory manually. Memory addresses are an essential concept of the C language in the form of typed *pointers*. These pointers contain only an address of data or code but no boundary or allocation information, which programmers must manage themselves. In all typical implementations, the referenced memory can come from three different pools with unique characteristics:

- **the data segment:** Global variables are located in a preallocated piece of memory at a fixed address, which is available for the whole runtime of the application.
- **the stack:** For each called function, a C program automatically reserves a piece of memory on a stack; this piece of memory is called the *stack frame*. It contains mainly the function's local variables and some information relevant to the machine's calling convention. It is automatically released when the function returns.
- **the heap:** Data that is not function-local can be stored on the heap, which the C runtime provides. Programmers can reserve variable-sized chunks of memory from the heap at any time, which remain allocated until the programmer manually releases them. The C runtime then allocates the necessary memory from the operating system and manages the released chunks of memory.

This example code shows the different types of memory. For each type, it contains one memory allocation:

```
1  int a = 0;                   // a 4-byte number in data segment
2  void main() {
3      char buffer[16];         // a 16-byte buffer on the stack
4      char *c = malloc(32);    // a 32-byte chunk on the heap
5      // ... c is now a pointer to the heap chunk ...
6
7      printf("%p %p %p\n", &a, buffer, c);  // show pointers
8
9      free(c);                 // manually release heap memory ("c")
10     return;                  // stack var "buffer" is released here
11 }
```

C++ is an object-oriented language based on C. It extends pure C with many new language features, with *classes* and *inheritance* being the most prominent ones. A class is a structural type combining data with functions operating on it (called *methods*). Classes can inherit from others, adding additional data or methods. An inheriting class can *override* methods it inherited from its parent class. Similar to C, memory management is done manually using pointers. Newer versions of the C++ runtime provide constructs to make memory management easier, preventing some but not all possible memory corruptions when used correctly. On the other hand, inheritance and overridden methods are good targets for memory corruption from an attacker's perspective.

2.2 Memory Corruption Vulnerabilities and Attacks

As of the C/C++ specifications [67, 68], any memory or pointer management error like out-of-bounds access is *undefined behavior*—it is up to the compiler, hardware, and operating system how the application behaves in these cases. In particular, a pointer handling error does not necessarily terminate the application, leaving it in a corrupted state. This section shows how attackers can trigger undefined behavior, exploit the underlying implementation and bend an application's actual behavior to their needs.

Attacks on C/C++ programs usually start with an error in the memory-managing code, producing an invalid pointer. Such a pointer can either be a *spatial* violation (e.g., go out of bounds) or a *temporal* violation (e.g., point to memory that is no longer valid). When such a pointer is used to read from or write to the pointed memory, the application's behavior starts deviating from the expected behavior—operations on invalid pointers do not crash the program but access memory that the program has in use for other purposes. The actual attack and the possible impact depend on the memory type the invalid pointer is referencing.

- **Data segment** pointers can only violate spatial safety because the data segment has an unbounded lifetime. An attacker with an out-of-bounds pointer to the data segment can tamper with global variables. In particular, global variables might contain data structures with function pointers. Overriding a function pointer can divert control flow and be the start of arbitrary code execution. Similarly, global variables might contain C++ objects or pointers to C++ objects. These objects might have an associated virtual function table, which attackers can override to divert control flow (see Section 5.4.4 for details). Depending on the operating system and binary layout, a *global offset table* might be located next to the data segment. It contains plenty of function pointers necessary for the interaction with libraries and therefore is a viable target for attackers.

- **Stack** pointers have been a primary target for a long time. The most common error source were *stack buffer overflow* bugs, where user input is too long to fit into a buffer, consequently overwriting subsequent memory chunks. Because stacks grow backward (in terms of addresses) on most architectures, such a (forward) overflow can corrupt everything allocated earlier on the stack. Next to overflows, violations in the calling convention can produce spatially invalid stack pointers. For example, functions accepting a variable number of arguments might be tricked into reading more arguments than actually passed, as in *format string* attacks. A temporal violation

of stack pointers is possible: A pointer to a local variable might be moved outside of the function containing the local variable. If this pointer is used after the function has returned, it is invalid: the local variable is no longer allocated, and another function's stack frame might have taken its place.

For an attacker, the stack usually contains plenty of interesting targets: Next to local variables and pointers to almost any other data structure, it includes the *return address* of each function—the address in code that the program should jump to once the function returns. This address has been the primary target in attacks: overwriting this return address using an invalid stack pointer directly changes the control flow. It gives an attacker the possibility to execute almost arbitrary code.

- **Heap** pointers can violate spatial safety similar to stack pointers: a *heap buffer overflow* might write into the following chunks. Furthermore, a heap pointer going out-of-bounds to a lower address (for example, as the result of an *integer overflow*) can write into previous chunks or the control data of the heap's implementation. But more prominent are temporal safety violations that occur as a result of faulty memory management: A *double-free* attack gets possible when the same chunk of memory is reserved once but released multiple times. As a result, the heap implementation might allocate the same memory for multiple chunks, producing more invalid pointers. Most prominent are *use-after-free* bugs, where a chunk is released, but its pointer is still used afterwards, accessing either the heap's implementation control data or another unrelated chunk of memory. Use-after-free bugs are the single most common source of errors in Google Chrome [167], the most popular web browser today.

  For attackers, the heap often stores C++ objects. Objects with virtual function tables are a viable target for corruption. The heap might also store function pointers in complex data structures in C programs.

From an initial safety violation, the attacker can chain attack vectors. For example, the attacker can use an initial stack pointer going slightly out of bounds to corrupt a heap pointer. The attacker can then use this heap pointer to access the data segment and overwrite one of the function addresses stored there.

To get from a memory corruption to code execution, the attacker always triggers the vulnerability to finally corrupt a code address in memory, be it a return address, a function pointer, or a global offset table entry. In earlier times, it was easily possible to execute arbitrary code at this point: The attacker could set the code address to an attacker-controlled buffer and place machine instructions there.

Today, all major architectures and operating systems employ a hardware-based defense called `W⊕X` or *data execution prevention*, where writeable memory is not generally executable (see Section 3.2 for details). Consequently, attackers rely on *code-reuse attacks*: Instead of bringing their own machine instructions, they reuse existing instruction snippets in a different order. To this end, the attacker must find suitable *gadgets* in the program's machine code. A gadget is a series of instructions that end with a control transfer instruction with an attacker-controllable target. The attacker can use the final instruction to chain these gadgets together, building new program behavior out of existing code. The most well-known method is *return-oriented programming* (ROP) [15]: Attackers search for gadgets ending

with a `ret` instruction. This instruction pops an address off the stack and jumps to it. If the attacker places the addresses of multiple gadgets next to each other on the stack, they will be executed one after another. A modern alternative is *call-oriented programming* (COP) [19] based on indirect jump and call instructions, which do not rely on the stack. For C++, researchers have developed *counterfeit object-oriented programming* (COOP) [151] based solely on C++ virtual methods. For all these methods, the attacker has to rely on the presence of enough suitable gadgets for his intention. Given these gadgets and an initial code address, he can build a code execution attack from a memory corruption vulnerability.

Having achieved code execution, the attacker has numerous possibilities to proceed: In the real world, the deployed code would, for example, install malware to get persistent control over the attacked system, leak sensitive data or make the system inoperable.

2.3 Attacker Model

In this thesis, we will consider attacks on C or C++ programs. These programs include large programs, legacy programs not maintained well, and programs where the original developers are unavailable. In this scenario, we do not assume that the users of our tools have much knowledge about these programs. In general, we can assume that language, the build scripts, library dependencies, and the expected functionality of the program are known, but no profound knowledge of the code is given. A typical real-world incarnation of this scenario is a Linux package maintainer, who is building and distributing software, but rarely writing code. In this scenario, solutions must be *automatic*: apart from the details outlined above, our solution must infer all necessary knowledge independently without requiring user interaction.

In our model, this program is the target of an *attacker*. We assume that the attacker has found memory corruption vulnerabilities in the program. The attacker can trigger them to change the content of writeable memory in the target process, i.e., change all data but not the (read-only) code. We have no further assumptions on timing, memory location or content the attacker writes. Our attacker's goal is first to reach code execution, then change the underlying system. Our solutions aim to stop relevant changes by preventing the attacker's code execution or limiting the impact of the attacked code on the system.

We give more details on the exact goals in the respective chapters: Section 5.5, Section 6.4.1, and Section 7.5.1. Each of our solutions covers a specific kind of attack. In the respective attacker models, we do not limit the memory corruptions but assume other orthogonal protections are in place. For example, we assume a return address verification scheme if return addresses are out of scope.

2.4 The LLVM Compiler Framework

Before a computer can execute C and C++ programs, they must be translated into machine instructions. A program named *compiler* does this translation. A compiler must check if a program is syntactically and semantically valid, it must combine all source code files, and translate them into an executable file. There are many C/C++ compilers used in practice, with the Gnu Compiler Collection (gcc), Visual C++, and Clang being most common.

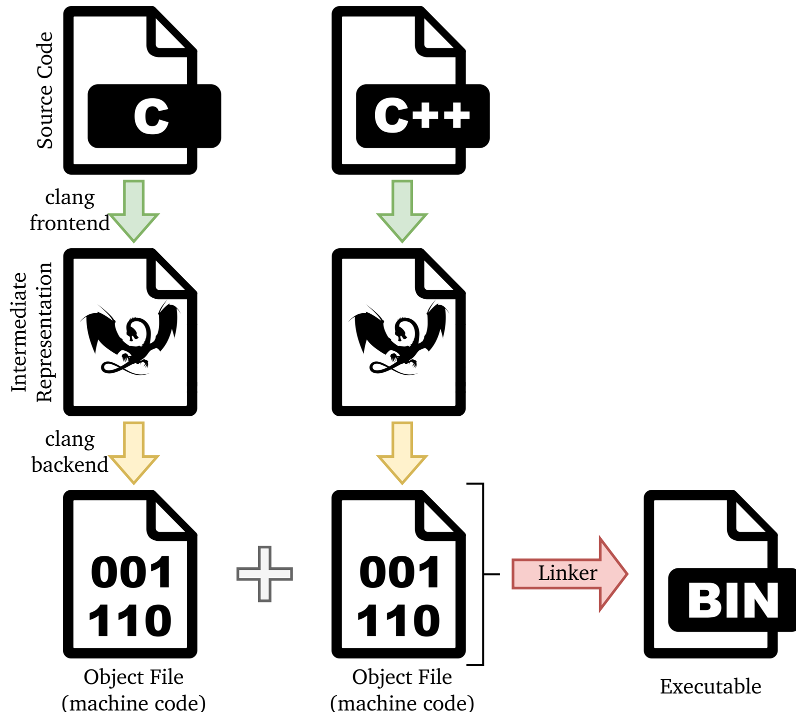
2.4.1 Clang

In this dissertation, we focus on the Clang compiler because it is open source, well-maintained, and easily extendable. Clang is part of the LLVM ("low-level virtual machine") framework, an *intermediate language* (IR) with associated tooling. Compiling programs with LLVM is a four-step process: First, different *frontends* like Clang translate different programming languages into the LLVM intermediate representation (IR). Second, this representation can be transformed, processed, and optimized by language-agnostic and hardware-agnostic methods, so-called *passes*. Third, different *backends* translate the intermediate representation to hardware-specific assembly or machine code. Finally, the *linker* combines one or multiple machine code files into a single executable file. This separation of concerns makes LLVM pretty modular—in particular, it is possible to add new transformations without considering source languages or target hardware. Our solutions mainly operate in the optimization stage of LLVM, transforming the intermediate representation into a more secure program representation. Furthermore, some of our solutions rely on source code information collected in the frontend, and some solutions add additional code (runtime libraries) to the final linking step. Thus, apart from the runtime libraries, our solutions are independent of the concrete backend, i.e., the target hardware.

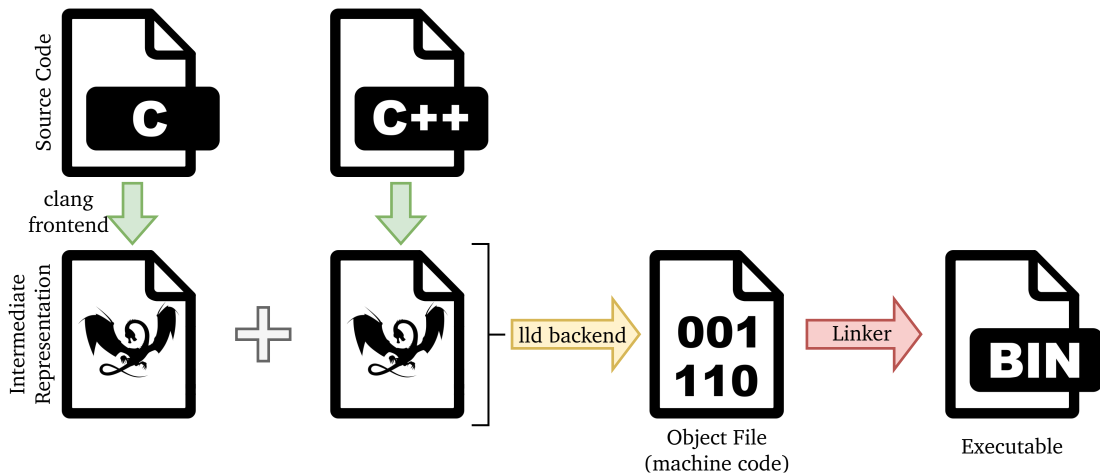
2.4.2 Linkers and Link-Time-Optimization.

Most C and C++ programs consist of many source code files. Figure 2.1a shows how all major build systems handle programs with multiple files. The compiler processes each file independently, from the frontend over the optimization stage to the backend. An *object file* is produced for each source code, containing the generated machine code. When all files have been converted to machine code, they are combined in the fourth step: The *linker* concatenates all object files, resolves any links between them, adds boilerplate code if necessary, and combines everything into a single executable file. This scheme is very efficient in terms of compilation time because it allows parallel compilation, and tools can cache the compilation result of unchanged source files. However, this scheme is not well suited for passes that benefit from knowledge about other source code files: For example, "dead code elimination" can only remove a function after ensuring that no other source file calls it, and function inlining can only integrate functions if both caller and callee are present in the same IR file. In particular, advanced security schemes working on the architecture-independent IR are affected: When IR is processed, only the source information from one single source code file is available. But many schemes require knowledge about the whole program, making the traditional compilation mode an obstacle.

Compiler designers have solved this problem by introducing *link-time optimization* (LTO). Figure 2.1b shows the difference from the traditional build system. Each source file is compiled into a file containing IR, not an object file containing machine code. The backend does not execute as part of the source file compilation process but is moved into the linker. When the program is linked, the linker merges all IR files into one big IR unit, in contrast to one big machine code unit. Additional more powerful compiler passes inside the linker can optimize this single unit. Finally, the compiler's backend is invoked, transforming the IR unit to a single machine code file, which is then linked. Reordering the four compilation phases increases the compilation time but can result in better-optimized programs. So far, link-time



(a) Traditional compilation steps. Each file is compiled and optimized independently to machine code, then object files are linked together. The dragon is LLVM's logo.



(b) Compilation steps with link-time optimization. Each file is compiled to the intermediate representation. All files are first combined, then optimized and converted to machine code.

Figure 2.1: Compilation and Linking of C/C++ programs with the Clang/LLVM toolchain.

optimization has been integrated in all major compilers. Almost any C or C++ program supports link-time optimization. Even massive programs such as Chromium compile with link-time optimization in feasible time.

With link-time optimization enabled, LLVM produces *bitcode files* containing pre-optimized IR instead of object files. LLVM's linker `lld` is aware of these bitcode files and runs a set of additional optimization passes on the given IR. `lld` also contains the compiler backends, transforming IR into machine code. As an alternative, LLVM offers a plugin for traditional GNU linkers, doing the same job.

Our solutions rely on link-time optimization and put additional IR passes in `lld`. At this point in the compilation process, our solutions see all information collected in the frontend, can collect additional information from the whole program's IR, and can transform the whole program at once. Furthermore, extending the linker `lld` makes it easy to add custom runtime libraries into the program. None of these changes requires additional input from or modifications to the program or build system, making the solutions easily applicable.

2.4.3 Multi-Module Programs.

C and C++ applications often reuse code from other projects packaged in libraries. A library is a single file containing reusable, binary machine code. Applications can load this file into their process at runtime; the loaded code executes within the application's process. The process of loading other executable files at startup is called *dynamic linking*. Dynamically linked libraries might be shipped as a pre-compiled file without source code. So the code to be loaded is often not known in advance, nor can it be analyzed together with the application. A C/C++ application only needs a library's *header files* to interact with it. They contain names and types of the library's exported functions and global symbols, its *interface*, which our solutions can use for reasoning. In Chapter 7, we will present CALI, a system dealing with the risks of libraries and dynamic linking.

Some applications can even load libraries after startup that have not been linked. For example, applications can have a plugin system where users can extend it with custom functionality. *Dynamic loading* allows applications to load any library at any time into a process and access its interface. Neither the libraries' names, interface, or header files are known at compilation time. Therefore, no whole-program analysis is possible, and compile-time solutions cannot infer any knowledge about this application, except that it might load code in the future. The function pointer protection TYPRO, presented in Chapter 6, supports dynamic linking and dynamic loading, despite relying on compile-time type analysis.

In contrast to dynamic linking and loading, *static linking* allows a developer to combine multiple modules in a single executable; application and libraries are combined at build time. Therefore, statically linked applications are more independent of the underlying operating system and the libraries installed there, e.g., avoiding problems with incompatible versions. However, static linking increases file size and has potential licensing problems, e.g., when combining LGPL [42] libraries with proprietary applications. In practice, applications can use static and dynamic linking together—some system-dependent libraries might use dynamic linking, while other version-dependent libraries might use static linking. In Chapter 5, we present a system (NOVT) relying on static linking for whole-program analysis and hardening.

3
Related Work

3.1 Preconditions for Code Execution Attacks

Memory corruption vulnerabilities and code execution attacks are long-known problems and have been under research for some time.

Researchers have proposed many defenses, while other researchers and real-world attackers have invented new ways to circumvent these defenses. Szekeres et al. summarize both attacks and defenses in their paper "SoK: Eternal War in Memory" [164]: In their attack model summary, an attacker with a memory corruption vulnerability has two ways to reach code execution, with different preconditions:

1. *code corruption attack*: the attacker can modify existing code and replace it with the desired functionality. According to Szekeres et al., the attacker must be able to:
   (a) modify existing, executable code and
   (b) write machine-specific instructions.
2. *control-flow hijack attack*: the attacker can re-purpose existing control-flow instructions to re-use existing code. This is today's primary attack method, which we already described in Section 2.2. To perform this attack, an attacker must be able to:
   (a) modify a code pointer
   (b) know the addresses of useful code or gadgets and
   (c) trigger an indirect control flow transfer using this pointer.

The model of [164] contains two more attacks (information leaks and data-only attacks), but they are not relevant for code execution attacks or this thesis. Similar, attacks can be chained together, e.g., attackers can use a weak control-flow hijack attack to invoke `mprotect`, which in turn enables a more powerful code corruption attack. This problem is solved if both attack types are mitigated—stopping the first of a chain of attacks is enough to protect the application.

In this chapter, we will introduce existing defenses against code execution attacks. Some of these defenses are already deployed in real-world programs; we will show which attack conditions they target. We will also show the covered parts and the weaknesses these defenses might have; and we will show where our protections from the following chapters will fit in.

3.2 Separating Code from Data

As a basic defense, memory corruption attacks must not be able to write code. This policy is named W⊕X [188] or *data execution prevention* (DEP) [102]. No part of a program's memory can be writeable and executable simultaneously. In some way, W⊕X creates a runtime distinction between code and data: Code is executable but must be read-only (rx), i.e., attackers cannot overwrite code. Data can be writeable but must not be executable (rw), i.e., attackers cannot write new code in data memory.

The strict enforcement of W⊕X prevents code corruption attacks completely by invalidating condition 1(a). Instead, attackers can only try to mount *code-reuse attacks* [15]: Their

code execution capabilities are limited to the gadgets in the original program—the attacker cannot create new gadgets. Furthermore, the number of gadgets is reduced considerably because all data is not executable anymore. Thus, condition 2(b) is harder to fulfill for attackers because only existing gadgets from the existing code can be reused.

All major compilers and operating systems have adopted this policy and produce protected binaries by default. Existing code did not need changes, apart from just-in-time compilers, which required little patches. The permissions of memory pages are implemented in the system's pagetables and are enforced by the CPU. Therefore, this defense has no performance overhead. W⊕X is widely adopted; we consider it active in all our threat models. All further presented defenses can also focus on the remaining threat: control-flow hijack attacks.

3.3 Preventing Memory Corruption

The root cause of most code execution attacks is the existence of memory corruption vulnerabilities. Thus, researchers have developed defenses against memory corruption in general, denying condition 1(a) and 2(a) of the considered attacks. So far, no defense has seen relevant adoption in practice, in particular, because they impose high runtime overhead or are incompatible with many legacy programs.

Memory Corruption Checkers in Academic Research. CCured [125], SoftBound [122], WIT [2] and the tool from [204] track potentially insecure pointers and insert runtime checks around their usages. BinArmor [157] stops buffer overflows even in closed-source binaries. CRED [148], "backwards-compatible array bounds checking" [33], "baggy bounds checking" [3], and PAriCheck [205] detect buffer overflows at runtime, at different precision and performance penalties. [33] and [82] propose memory partitions to reduce the impact of memory corruptions. PARTS [86] uses ARM's pointer authentication [142] instructions to detect modified pointers at runtime. Data-Flow Integrity [21] statically determines which data can be written to certain memory regions and enforces this at runtime. Cling [4] or CETS [123] counter temporal attacks like use-after-free: Cling is a custom memory allocator that does not reuse addresses for incompatible types. At the same time, CETS detects reused pointers at runtime. Cyclone [70] proposes a safe dialect of C, preventing memory corruption errors.

Valgrind Memcheck [126, 154] and AddressSanitizer [153, 171] (with its spin-offs MemorySanitizer [174], ThreadSanitizer [177] and UndefinedBehaviorSanitizer [178]) detect common memory errors at runtime with performance overhead starting at 2×. Developers often use these sanitizers to check for subtle memory errors during application testing, but not in production.

In the meantime, hardware features like ARM's Memory Tagging Extensions [7] have been introduced to catch invalid memory access with little performance penalty. However, the implementable policies can only catch few memory corruptions because only few (16 for ARM) memory classes are possible. For example, a typical implementation might catch a heap-based out-of-bounds memory access only if it overflows from an allocated heap chunk into another one. Overflows within data structures or between stack variables of the same function are undetectable. As soon as attackers can corrupt a pointer within the same data structure, memory tagging can be completely broken with some non-negligible probability.

Safe Languages. Memory corruption vulnerabilities occur because C and C++ are inherently memory-unsafe. To this end, developers have started adopting new, memory-safe languages. In these languages, the language design and compiler-enforced rules forbid constructs that might not be memory-safe, preventing accidental memory corruption bugs. In particular, Rust [147] is a low-level system programming language that uses extensive compiler-based verification and runtime checks to generate safe executables. It can be as fast as C/C++ and is compatible with legacy C libraries. However, a major effort is necessary for adoption in legacy projects, as developers must rewrite all source code in the new language. For new projects, this is not a problem: For example, new parts of the Firefox engine are written in Rust [116].

3.4 Protecting Code Pointers (Code Pointer Integrity)

As outlined in the previous section, there is no accepted solution to protect an existing program's memory from attacker-controlled memory corruptions. Attackers can still modify a code pointer (conditions 2(a)) as the first step toward control-flow hijack attacks. Researchers have proposed schemes like Code Pointer Integrity (CPI) [79] that protect these code pointers in particular, but not the remainder of a program's memory. Protection of all code pointers would stop code-reuse attacks at the very first step. However, such protections are very complicated because any pointer that might point to code directly or indirectly must be moved to specially secured memory, and every write access to this protected memory must be checked. This is inefficient for real-world deployment. So, in general, the integrity of code pointers in applications is not protected, with two exceptions: SafeStack and RELRO.
SafeStack. As a weak form of code pointer integrity, SafeStack [175] is part of the Clang compiler. Programs with SafeStack use two stacks: the safe stack contains return addresses, spilled registers, and "safe" local variables, while the unsafe stack contains "unsafe" local variables. Local variables are unsafe if the program accesses them in a way that involves pointer arithmetic or if the function might leak their address. A stack-based attack can only occur from variables on the unsafe stack. Attackers can only target other variables on the unsafe stack; the return addresses on the safe stack are not located nearby. Furthermore, the address of the safe stack is kept secret and never stored in the unsafe stack or heap. Therefore attackers with strong memory corruption abilities cannot target the safe stack.

SafeStack has little to no performance overhead but is not fully compatible with all application designs: it cannot be applied to dynamic libraries and requires a specific compiler runtime (LLVM's compiler-rt) bundled with the application.
RELRO. The C runtime and the linker also introduce some code pointers into applications, invisible to the programmer. Most prominent, *relocations* [182] are necessary to connect an application with its runtime-loaded libraries. To this end, a program or library contains one code pointer for each function it imports from other modules. When this function is first called, the dynamic linker sets the code pointer to the actual address of the desired function.

Newer operating systems support *relocation read-only* (RELRO) [156] mode to prevent attacks on relocations. When enabled, the dynamic linker sets all relocations to the corresponding addresses at startup, and lazy evaluation is disabled. Then the whole section containing relocations is read-only—attackers cannot tamper with it anymore. RELRO prevents attacks on code pointers outside of the application scope, and precondition 2(a) is

invalidated for these pointers.

Today, RELRO is available in major compilers and linkers but not enabled by default. While RELRO has no compatibility problems, it can considerably increase the startup time of large applications.

Remaining Code Pointers. Even if both SafeStack and RELRO are active, many code pointers remain for attackers to target. In particular, function pointers in C programs and vtable pointers in C++ programs remain accessible to attackers. If an application faces compatibility problems with SafeStack, even return addresses on the stack are accessible again. With the current state of the art, protecting code pointers (condition 2(a)) is impossible under the harsh conditions for real-world adaption.

3.5 Hiding Gadgets

Attackers can only perform code-reuse attacks if the application contains enough valuable gadgets and the attacker knows their address (condition 2(b)). All major operating systems try to hide these addresses from attackers using randomization. All deployed or experimental randomization-based schemes have in common that sufficient information about a program's memory can defeat its protection.

Address Space Layout Randomization (ASLR). ASLR [137] aims to hide the program's memory layout from attackers. All libraries are loaded to a random memory location each time the application is started. Attackers can use gadgets from this library only if they know the random memory locations and can adapt their attack to the current addresses. Thus, precondition 2(b) is not given anymore. Later, *position-independent executables* [144] extended the randomization to the program itself—in these programs, attackers cannot know any address in advance. To defeat ASLR, attackers need an additional vulnerability that leaks at least one address from a library. To defeat position-independent executables, attackers must leak a code address from the program. After that leak, attackers must be able to trigger their actual code execution attack before the application terminates or restarts. This adaptive attack schema is hardly possible for offline targets, e.g., applications that are attacked by malicious input files.

Today, all major compilers and operating systems support randomization and enable it by default for libraries. Some compilers (like gcc on Ubuntu or Debian) produce position-independent executables by default. On the application side, compilers can add ASLR support alone—no source code changes are required. Programs and operating systems with and without ASLR support are compatible. Specific hardware support is not required, and performance impact is minimal: we measured a median runtime overhead of 0.1% of position-independent executables on the SPEC CPU 2017 benchmark.

In our thread models, we usually assume that attackers have already defeated ASLR; our solutions do not rely on randomization. Our solutions remain compatible with ASLR.

Hiding Gadgets and Information by Randomization. Substantial research has been done to hide addresses, useful gadgets, and the program's memory layout from attackers beyond ASLR. All these defenses randomize different parts of the application at different granularity. Attackers can defeat these schemes by leaking enough information about the randomization; the necessary amount varies from single addresses to complete memory dumps. None of these defenses has seen real-world adoption.

Address Space Layout Permutation [75] randomizes all mapped addresses and the order of functions and variables. Binary Stirring [196], Xifer [30], and ILR [59] further randomize addresses of individual basic blocks or instructions. Software Diversity approaches [81] randomize the whole code so attackers cannot search for gadgets in advance. Data Space Randomization [9] and CoDaRR [143] randomize the addresses and layout of data in the program, thwarting not only code execution but also data-only attacks. Address space randomization has been further extended to shared libraries [8], just-in-time compiled code [8] operating system kernels [48], while runtime re-randomization after forks [94] or any output [10] defeats certain information leaks. As additional protection on top of randomization-based solutions, PointGuard [26] encrypts pointers in memory—information leaks cannot reveal randomized addresses anymore.

3.6 Control Flow Integrity

So far, we have seen defenses trying to invalidate conditions 2(a) and 2(b). In many cases, attackers can still modify code pointers and find suitable gadgets in existing code. In the final step of a control-flow hijack attack, attackers trigger an indirect control flow transfer to a modified code pointer.

Control flow integrity (CFI) [1] tries to prevent this step. A CFI scheme checks each code pointer for validity before allowing its use in a control flow transfer. CFI schemes terminate the application under attack if a modified code pointer is detected, preventing the code execution and further damage. Thus, precondition 2(c) is invalidated. Code pointers in applications are manifold; most CFI schemes target only specific classes of code pointers. We can categorize these code pointers into four classes: return addresses, indirect call targets (including C++ virtual dispatch), relocations, and jumps in hand-written assembly. Developers can protect relocation code pointers with RELRO, and jumps in hand-written assembly are very rare: we found only two in musl libc, and none in any of the reviewed applications. Thus, we must focus on return addresses and indirect call targets.

3.6.1 Backward CFI

C and C++ applications store a code pointer on the stack for each function call—the return address. When the called function has completed its execution, this code pointer is used to resume the execution of the calling function. While function calls go forward in the program, function returns go back to the calling function; therefore, returns are named "backward control flow." Return addresses have been a primary target for attackers for years. Different CFI-based defenses with different strengths have been proposed, for example, *stack canaries* [37], *shadow stacks* [29], and *return address encryption* [133]. We explain these schemes in more detail in Chapter 4, where we evaluate these defenses.

In the real world, only stack canaries have seen widespread adoption due to their compatibility and performance (around 1% overhead). We will show in Chapter 4 that shadow stacks can be implemented with acceptable performance overhead on modern systems. Furthermore, recent x86 CPUs have gained hardware support for shadow stacks as part of Intel CET [155, 136], removing their performance penalty. Recent ARM CPUs have hardware support for Pointer Authentication [142] which compilers can use to implement a

stronger version of return address encryption. We therefore expect increasing adoption of return address protections in the years to come.

3.6.2 Forward CFI

Forward CFI protects all code pointers explicitly managed by C and C++ applications, namely C-style function pointers and C++ virtual dispatch. Programmers use this functionality to build modular software—modules can be extended with additional functionality, i.e., additional code referenced by code pointers. On the hardware level, these features translate into indirect calls to code pointers loaded from mutable memory, which an attacker could have modified. So far, numerous defenses are trying to enforce CFI. However, many of these schemes have limitations regarding CFI policy, compatibility, hardware requirements, or performance. In Chapter 5 and Chapter 6, we present two forward CFI solutions that tackle many of these problems. We present existing C-based CFI solutions in Section 6.9 and Section 6.11, where we compare them to our solution TYPRO from Chapter 6. We present related C++-based CFI solutions in Section 5.9.2, where we compare them to our solution NOVT from Chapter 5.

In the real world, no forward CFI scheme has seen widespread adoption. Some major compilers already contain CFI schemes, such as Clang CFI [170] or CFGuard [101] in the Microsoft Visual C++ compiler, but applications rarely use these defenses. Again, recent x86 CPU's have gained hardware support (Intel CET indirect branch tracking [136]) to improve the performance of forward CFI checks, which could boost the adoption of a weaker but compatible CFI scheme. ARM CPUs feature Pointer Authentication [142] which can be used by compilers to implement CFI policies of different strength.

3.7 Isolation and Compartmentalization

If code execution attacks succeed, attackers can use all of the program's privileges to damage the underlying system. For example, attackers can leak sensitive files, install backdoors, misuse computational resources for their own purpose, or delete important data. As a last line of defense, developers of high-risk applications try to isolate potentially dangerous code from the system. This code runs in special *compartments* that cannot have any meaningful impact on the system. These compartments are either processes with extremely low privileges or software-based execution environments like NaCl [203] or WebAssembly [54].

For example, the popular browser Google Chrome has separate processes for interpreters, parsers, and rendering code [168]. These "sandboxed" processes have no network or disk access; only a central, privileged process can access these resources. It contains the I/O code only and sends all received data to the restricted processes for further processing. If a memory corruption or code execution vulnerability occurs, it is likely in the restricted code. Apart from crashing the application, it cannot have any malicious impact. Furthermore, Chrome's approach provides some level of *memory compartmentalization*: Using multiple processes for different websites (origins) and tabs, Chrome can isolate all sensitive data between different websites. An attack from one website cannot use memory corruption vulnerabilities to access the sensitive data in another website's memory. Thus, isolation targets not only code execution attacks but also memory corruption vulnerabilities themselves.

While other high-risk applications such as Mozilla Firefox [117] or OpenSSH [43] implement similar schemes, defenses based on program isolation are not widely deployed. The primary reason is the developer's effort to implement, test, and maintain proper isolation—as of today, software isolation is mostly performed manually. In Section 7.4, we show how researchers and other developers have approached isolation and compare these approaches to our solution for automated library isolation—CALI.

4
SoK: Evaluation of Return Address Protections

4.1 Motivation

In the past two decades, stack-based attacks targeting return addresses have been a major threat to computer systems. Researchers have proposed countermeasures that detect and prevent these attacks at runtime. However, most of these protections have been reported to be too slow for wide-spread real-world adoption, i.e., slow down programs by 5%–10%. Only one countermeasure received more wide-spread adoption in a weakened version—in many cases, attackers can circumvent the provided protection with improved attacks.

In this chapter, we re-implement three well-known return address protections: stack canaries, shadow stacks and return address encryption. We re-evaluate the protections on modern hardware, software, and compilers. We show that their performance overhead has degraded well over time—up to a point where it might not prevent wider real-world adoption: In particular, we show that a speed-optimized shadow stack has a mean performance overhead of 2.7% only—it is faster and more secure than using stack canaries in every function (4.1% overhead). We further highlight the impact of the chosen programming languages on the expected performance overhead—C, C++ and Fortran are differently affected.

4.2 Problem Description

For decades, attackers have been using stack-based memory corruptions to take over entire applications and systems. Since the initial invention of stack smashing attacks [134] in 1996, attackers have targeted an uncountable number of applications. Information leakages and return-oriented programming (ROP) techniques worsen this, as they allow attackers to evade most deployed countermeasures like `W⊕X` or ASLR, as outlined in Section 3.2 and Section 3.5.

Shortly after stack-based attacks became common, researchers developed countermeasures: Stack Canaries [37] made it into major compilers, while more robust protections like Shadow Stacks [189] have not seen any relevant adoption. Two decades later, hardware architectures, processors, and software have changed considerably. However, little work has been done to re-evaluate the existing protection techniques after their initial publication and the initial reservation against stronger protections has not vanished. Shadow stacks are still mainly used in research prototypes only. While Intel CET [155, 136] will provide hardware-assisted shadow stacks, and ARM Pointer Authentication [142] signs valid return addresses, the supporting hardware is still rare—the need for fast, software-based return address protections is still present.

4.3 Contributions

In this chapter, we want to evaluate the state of software-only return address protection in 2022. We re-implemented three protections, namely *stack canaries* [37], *shadow stacks* [29, 189] and *return address encryption* [133]; our implementations are open-source[1]. We measure the performance overhead on a modern system with modern hardware and a modern

[1] `https://github.com/MarkusBauer/return-address-protections`

benchmark, and further, we evaluate how this overhead changes for different programming languages. We will show that most protections have much lower performance overhead in these modern scenarios than initially assumed. We will also introduce benchmarks, their usage, and methods to check the significance of results, which we will use in later chapters. Finally, we conclude that software-only return address protections have matured well. Considering backward CFI a solved problem, we can focus more on forward CFI in the following chapters.

4.4 Background: Attacks on Return Addresses

Backward CFI schemes defend against stack-based memory corruption attacks, as explained in Section 2.2. Figure 4.1 shows the typical stack layout on 64-bit x86 systems. The stack grows downwards and contains each function's local variables, plus an address of its calling function (the return address). This allows for the exploitation of a stack-based memory error, often a buffer overflow [134]. By exploiting this error, attackers can change memory from a local array upwards, thus reaching the return address. Overwriting any return address allows an attacker to change the program's control flow arbitrary as soon as the stack frame is released and the function returns.

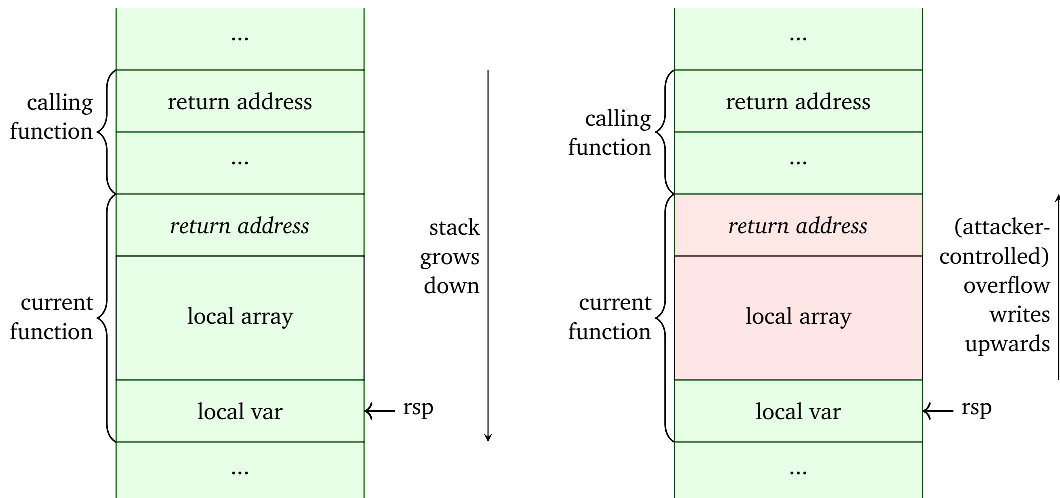


Figure 4.1: Stack layout of C/C++ programs. Left: typical layout with two functions. Right: Impact of a buffer overflow memory corruption (red).

4.5 Backward CFI schemes

We have chosen three backward CFI schemes of similar complexity: Stack Canaries, Shadow Stacks, and Return Address Encryption. All three backward CFI schemes have a similar functionality—they check if the return address on the stack has changed between a function

call and the following function return. If the address has been altered, the program is stopped and an error is reported. Attacker-overwritten return addresses do not pass this check and will never be used. Thus, no attacker-chosen code is executed.

4.5.1 Stack Canaries

Stack canaries [27] insert a secret value called the canary in each function's stack frame, i.e., the canary is stored just below the saved return address on the stack, see Figure 4.2. Before using the return address upon function exit, the function epilogue checks the secret value for changes, and if the canary has changed (e.g., due to a stack-corrupting attack), the program terminates.

Stack canaries mainly detect buffer overflow attacks. An overflow overwriting the return address also overwrites everything between buffer and return address, including the canary. The original canary value is secret and random; thus, the overflow will have to change the canary's content.

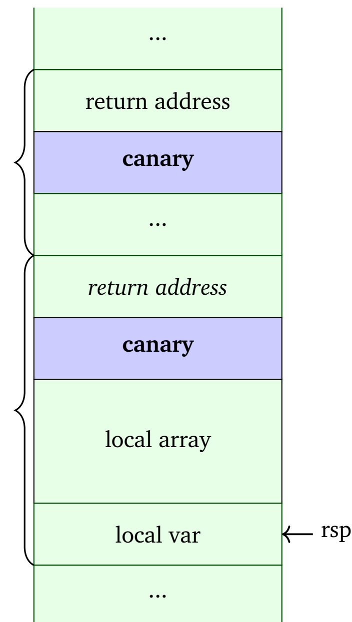


Figure 4.2: Stack layout with canary.

Security. A stack canary does not protect against attacks that overwrite the return address directly, i.e., without affecting other memory locations. For example, format string attacks can circumvent stack canaries. Furthermore, canaries can be defeated with a sufficient information leak because each process has only one fixed canary value shared by all functions. If an attacker can leak this canary value, the process is vulnerable to buffer overflows again.
History. Stack Canaries were suggested in 1998 [27] and have seen wide-spread adoption. By now, all major compilers implement Stack Canaries. Many compilers enable them by default on vulnerable functions.

4.5.2 Shadow Stacks

A shadow stack [189] acts as a second stack to protect return addresses. After calling a function, the return address is pushed to *both* stacks (the original stack and the shadow stack). Before returning, the program compares the addresses on top of the original stack and on top of the shadow stack. The basic idea is that buffer overflows allow attackers to modify the original stack only, not the shadow stack. If the addresses stored on both stacks differ, the return address on the original stack has changed, and the application terminates.

Dang et al. [29] proposed a computationally more efficient variant of this traditional shadow stack scheme: the parallel shadow stack, as shown in Figure 4.3. In this variant, the shadow stack grows parallel with the original stack, leaving all memory but the return address locations unused. This way, one can replace the second stack pointer with the

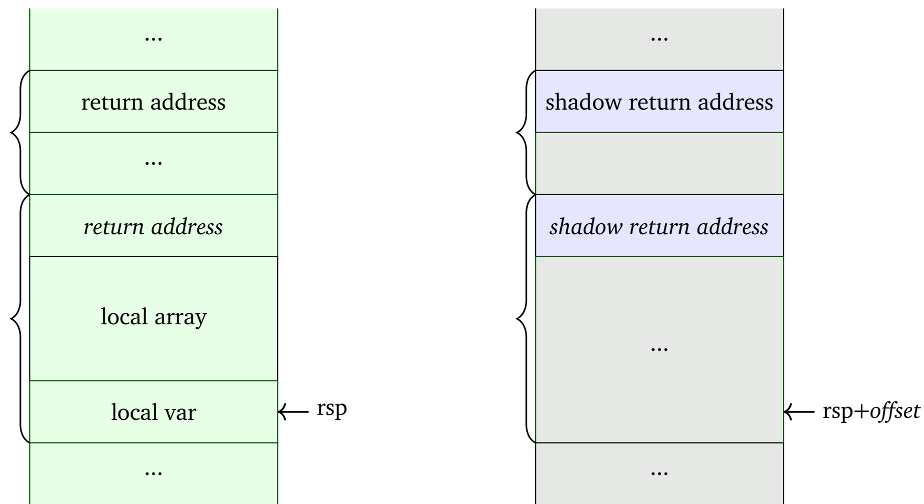


Figure 4.3: Stack layout with shadow stack. Left: regular stack. Right: shadow stack.

standard stack pointer and a constant but possibly random offset. If the offset is constant and fixed, the second shadow stack address can now be computed using one *add* instruction and the stack pointer register—no memory access is needed to compute the address of the respective item on the shadow stack. While this scheme is faster than the traditional shadow stack, it requires more memory.

Security. A shadow stack protects against attacks that overwrite the return address pointer, as long as the attacker cannot overwrite the address on the shadow stack before the function returns. To do so, an attacker requires a vulnerability that allows multiple arbitrary memory writes. Most shadow stack implementations place the shadow stack at a random memory address. In this case, overwriting the address on the shadow stack is only possible if the attacker has leaked the shadow stack's address before.

History. Shadow Stacks have been around for more than 20 years [189]. So far, they have been used in different research prototypes but not in production. No major compiler implements a general shadow stack by today—only Clang provides an implementation for 64-bit ARM [176]. This situation might change with the upcoming Intel CET hardware extensions, improving shadow stack performance considerably.

4.5.3 Return Address Encryption

Return Address Encryption [133] uses a secret key k of size equal to the return address size to *xor*-encrypt the return addresses, as depicted in Figure 4.4. After calling a function, the saved return address is replaced with $\mathsf{returnaddress} \oplus k$. Before returning, the same operation is applied again, which restores the old address. If an attack overwrites the saved return address, the overwritten value is clobbered with k before usage. If the attacker cannot guess k, the attacker-controlled but xor-decrypted return address is invalid. This way, an attacker cannot predict the jump target of his attack, and the program will likely crash due to an invalid jump target.

Security. Return Address encryption protects against any overwrites of the return address, as long as the secret value does not leak. Alternatively, an attacker can leak an encrypted return address and an arbitrary unencrypted code address to recover the secret key. Return address encryption works best when the address space is much larger than the executable memory; otherwise, spraying attacks are possible.
History. Onarlioglu et al. introduced Return Address Encryption in 2010 as part of the gadget removal tool GFree [133]. So far, it has not been implemented in a major compiler and has not been used in practice. ARM Pointer Authentication [142] proposes a similar concept which requires special hardware. Instead of a simple xor, ARM Pointer Authentication uses a cryptographic signature that is stored in the unused bits of the return address.

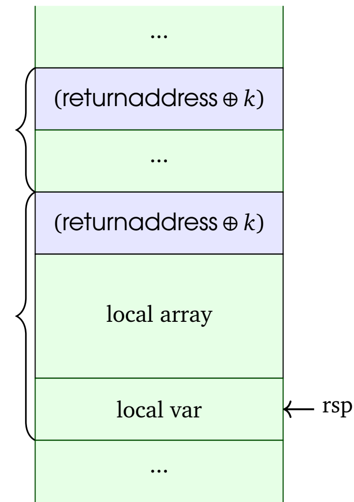


Figure 4.4: Stack layout with return address encryption.

4.6 The SPEC CPU Benchmark Suite

To perform our measurements, we focus on the SPEC CPU 2017 benchmark [160], the modern successor of the well-adapted SPEC CPU 2006 [58]. SPEC CPU has several advantages: It is a well-established industry standard focusing on CPU performance only and contains several common, real-world programs written in different languages (C, C++, and Fortran). In contrast to the many closed-source benchmarks, SPEC CPU builds its programs from source code. It is possible to adapt this benchmark to almost any system, and adding compiler-based security mechanisms at build is easy.

We use SPEC CPU 2017's "rate" benchmark, which includes 23 programs. These programs are separated into two groups: 10 integer benchmarks and 13 floating-point benchmarks. The integer benchmark suite contains, for example, compilers, compression tools, and artificial intelligence algorithms, while the floating point benchmark suite contains mainly simulations for physical or chemical systems. Table 4.1 summarizes the benchmarks in SPEC CPU 2017; additional details can be found in [135]. The program size varies greatly, from programs with 1300 lines of code and only six functions up to 1.7 million lines and more than 37000 functions. Nine programs are plain C, seven include C++, and seven include Fortran code. Some programs are written in more than one language—we have analyzed these programs and report only their primary programming language in which the application's core parts are written. Each program comes with a reference input data set. Each program runs between 5–10 minutes on our machine, and any startup or initialization overhead is thus negligible.

Usually, the predecessor SPEC CPU 2006 is used in academic works to evaluate all kinds of protections, even if it has been deprecated for years. For this work, we focus on the modern SPEC CPU 2017. Our results are thus not easily comparable to older results but

capture the protection's performance on modern and, in particular, much larger programs. In Chapter 5 and Chapter 6 of this thesis, we will use SPEC CPU 2006 instead to compare our solutions to related work. Most related work benchmark only parts of the SPEC suites, whereas we use all programs despite the necessary engineering effort. This complete set of test programs allows us to reason about the impact of different languages and program types on the performance overhead.

4.7 Implementation

In our experiments, we will compile and run all SPEC CPU 2017 benchmarks. For each protection, we build a protected version of each benchmark program, run it, and compare the runtime to the runtime of an unprotected version. We use the Clang/LLVM 13 compiler for C and C++ code. While an LLVM-based Fortran compiler is currently under development [179], it is not yet able to compile the Fortran programs from standard benchmarks (lacking full Fortran-95 support). To this end, we use gfortran 10 from the GNU compiler collection to compile Fortran code. We compile all benchmarks with full optimization settings, including link-time optimization: `-O3 -flto`. We omitted link-time optimization for the benchmark *wrf* only; gfortran's link-time optimization consumes more than the 32GB memory available.

While both compilers already implement stack canaries, we implement shadow stacks and return address encryption as assembly rewriters, wrapping the system's assembler `as`. Our implementations are thus compiler-agnostic and work with Clang and gfortran interchangeably but support only one target architecture (64-bit x86).

4.7.1 Stack Canaries

We use the built-in stack canaries from Clang/LLVM and GCC, as enabled with the flag `-fstack-protector-all` on both compilers. The dynamic loader initializes the 8-byte-long canary before the program starts and stores it in the thread-local storage at `fs:[0x28]`. Compilers instrument function prologues, `ret` instructions, and tail-calls.

```
; function prologue - set canary
sub     rsp, 0x18       ; allocate stack frame
mov     rax, QWORD PTR fs:0x28
mov     QWORD PTR [rsp+0x10], rax
; function epilogue - check canary
mov     rax, QWORD PTR fs:0x28
cmp     rax, [rsp+0x10]
jne     __canary_chk_fail
ret
```

In addition to the canary itself, LLVM reorders the stack layout by moving buffers close to the canary, protecting local variables. This implementation increases each function's memory usage by 8 bytes on average.

4.7.2 Shadow Stack

We implement a parallel shadow stack, as proposed in [29], which is faster than a traditional shadow stack. We decided to use a fixed memory offset for the shadow stack

(−0x70000000) instead of a random one because we want the benchmark to be stable and reproducible. A production-grade implementation could add a random offset with constant performance overhead at startup only: It could replace all shadow stack offsets in the machine code with a new random number at startup. This negligible one-time operation allows a randomized shadow stack without further performance penalties compared to our version with a constant offset. To simplify our implementation, we did not implement a growing shadow stack but reserved enough memory at the program's startup—around 400MB suffice for all programs. Initial experiments confirmed that this additional memory usage had no impact on the performance of the programs, as our test machine always has more than enough free memory. A production-grade shadow stack implementation would thus double the amount of stack memory. We instrument the same locations as stack canaries: function prologues, returns, and tail calls.

Scratch Registers. For our prologue and epilogue instrumentation, we require one scratch register because x86 instructions can have only one memory operand and we have two memory locations to access. However, a free register is not always available: LLVM can change the calling convention of functions during link-time optimization, including the registers used for parameters and the caller-saved registers. Thus, without breaking the program, we cannot modify any register in function pro- and epilogues. To this end, our implementation analyzes the function's assembly to find registers that get overwritten before being read. This analysis can also move the instrumentation point by a few instructions if necessary, as long as the instrumentation occurs before the first unsafe memory access. For example, many functions initially push callee-saved registers and restore them before returning. If a function starts with `push r11 ; mov r11, ...`, we can insert our function prologue code after the `push` instruction and use `r11` as a scratch register. If a function ends with `pop r11 ; ret`, we can insert our function epilogue code before the `pop` instruction and use `r11` as a scratch register.

This heuristic succeeds for 80% of all function prologues and 69% of all function epilogues. For all other functions, we save one register on the stack and restore it after our code has been executed, which might include some additional performance penalty. Alternatively, one could modify the compiler to have one register always caller-saved, but then our solution would depend on a single, modified compiler. Currently, there is no single compiler that handles all necessary languages.

```
; function prologue - store ra on shadow stack
mov    r11, QWORD PTR [rsp]
mov    QWORD PTR [rsp-0x70000000], r11
; function epilogue - check ra from shadow stack
mov    r11, QWORD PTR [rsp-0x70000000]
cmp    QWORD PTR [rsp], r11
jne    __shadow_stack_fail
ret
```

4.7.3 Return address encryption

Return address encryption relies on secret key storage. To simplify our implementation, we re-used the system's stack canary value stored in thread local storage because the dynamic loader cares about proper initialization. A production-grade implementation would modify

the dynamic loader to initialize a new secret value, but design and performance would not change otherwise. Our encryption is a simple *xor* cipher like [133], which is fast and secure enough for our purpose—all under the assumption that there are no information leaks.

In every function prologue, we load the secret key and *xor* the return address stored on the stack with that key. Before all function exit points, including tail calls, we restore the original saved return address by *xor*ing it with the secret key again. There is no need for an explicit check or branch here—a tampered address will be invalid, and `ret` will raise an interrupt. Return Address Encryption requires a scratch register. We use the same scratch register detection as the shadow stack implementation outlined above.

```
; function prologue - encrypt
mov     r11, QWORD PTR fs:0x28
xor     QWORD PTR [rsp], r11
; function epilogue - decrypt
mov     r11, QWORD PTR fs:0x28
xor     QWORD PTR [rsp], r11
ret
```

The drawback of this method is that the saved return address is first written to the stack by the function call, and then instantly read back and modified for encryption. However, a scheme with a single write cannot use the current processor's `call` and `ret` instructions, sacrificing all hardware optimizations made to call and returns.

One challenge with return address encryption was C++ exceptions support, as they rely on the stored return addresses for stack unwinding. We solved this issue using a patched version of LLVM's *libunwind*, which can detect and decrypt stored return addresses. Only two benchmark programs rely on C++ exceptions, which occur only once for each program run. Thus, in a complete benchmark, the changes to the unwind functions have negligible performance overhead if any.

4.7.4 Optimizations

When stack canaries were initially deployed, they introduced some non-negligible performance overhead in the protected programs. Compiler engineers developed some relaxed modes of operation which do not protect functions that are likely not vulnerable. In particular, small but frequently called functions without stack buffers can reduce the overhead a lot if not protected. Trading security for performance made stack canaries fast enough to be enabled by default. We implement a similar optimization for shadow stacks and return address encryption, omitting "safe" functions.

Partial Stack Canaries. By now, GCC [35] and Clang [96] know three modes of operation for canaries: regular mode (activated with flag `-fstack-protector`), strong mode (`-fstack-protector-strong`) and complete mode (`-fstack-protector-all`). We call the regular and strong modes "partial modes" because they protect only parts of the program.

In the regular mode, a function is protected if it contains a stack-allocated array with at least 8 bytes length. GCC uses this mode by default in many Linux distributions.

In the strong mode, the compiler adds canaries to functions that either contain any stack-allocated array independent of its length, use the address of a local variable in a function call, or assignment or use the local register variables extension.

Finally, the complete mode claims to add a canary to all functions. We show some exceptions in Section 4.7.5.

Optimized Shadow Stack / Return Address Encryption. We implement a similar optimization for our shadow stack and return address encryption tools. The optimized mode of our tools omits functions that can't overwrite the stored return address. As an assembly rewriter, we do not have information from the source code but rely on assembly analysis. To this end, a function is unsafe and needs protection if:

- the function calls another function (excluding tail calls)
- the function contains a `syscall`
- the function contains an instruction that writes memory and
  - the instruction is not `push` or `pop` (including their derivates) with a register operand and
  - the written address is not a local variable on the stack (i.e., the operand has the form `[rsp+0xc]` for some constant 0x*c*) and
  - the written address is not a global variable (i.e., the operand has the form `[rip+0xc]` or `[0xc]` for some constant 0x*c*)

On average, 15% of all functions do not need protection according to our rules. We evaluate the effectiveness of this optimization in the next section.

We implement one further optimization for shadow stack function prologues borrowed from [29]. If we cannot find a scratch register, we implement the prologue as a `pop` with memory operand, one of the very few x86 instructions that read and write memory at the same time. The final prologue will be `pop [rsp-0x70000000] ; sub rsp, 8`. The first instruction copies the return address to the shadow stack, but moves the stack pointer. The second instruction reverts this change to the stack pointer. This implementation does not need a scratch register; we save two memory-accessing instructions.

4.7.5 Evaluating Implementations

Table 4.1 summarizes all details of our implementation and the various optimizations evaluated on the SPEC 2017 benchmark programs. To this end, we compiled all programs to text assembly and analyzed the generated instructions.

Stack Canary Modes. First, we investigate how many functions get stack canaries added by the compiler in the different modes. Surprisingly, we found that compilers do not protect *all* functions even if `-fstack-protector-all` is set, but only 99% on average. In particular, C++ programs are affected—in the worst case, 3.8% of all functions remained unprotected. A deeper investigation shows that none of the unprotected function returns—functions that abort the program or end with tail calls are not considered. The second case could be a security risk. On the one hand, attackers can exploit only functions ending with `ret`. On the other, attackers can compromise the return address between prologue and tail call. The tail-called function cannot guard the return address with its canary in time, so its final `ret` can be exploited. Omitting tail-calling functions is an optimization compared to the other protections that cannot be disabled.

Table 4.1: Overview of SPEC 2017 benchmarks, their size, the protection coverage of different canary modes, and coverage of our optimization strategy ("optimized"). Additional summary of code locations where scratch registers were detected.

Benchmark	Lang.	LoC.	no.	Number of protected functions				Scratch register available in		
			funcs	canary all	canary strong	canary basic	optimized	prologues	returns	tail calls
blender_r	C++	1726K	37167	36636 (98.6%)	6933 (18.7%)	969 (2.6%)	27540 (74.1%)	31512/37167 (84.8%)	20509/40633 (50.5%)	3462/4688 (73.8%)
bwaves_r	Fortran	1304	6	6 (100.0%)	2 (33.3%)	1 (16.7%)	6 (100.0%)	5/6 (83.3%)	5/5 (100.0%)	1/1 (100.0%)
cactuBSSN_r	Fortran	170K	2630	2621 (99.7%)	442 (16.8%)	37 (1.4%)	1907 (72.5%)	2277/2630 (86.6%)	1798/2540 (70.8%)	620/693 (89.5%)
cam4_r	Fortran	243K	2383	2376 (99.7%)	1066 (44.7%)	276 (11.6%)	2187 (91.8%)	2086/2383 (87.5%)	2024/2598 (77.9%)	423/527 (80.3%)
deepsjeng_r	C++	7284	109	109 (100.0%)	21 (19.3%)	12 (11.0%)	69 (63.3%)	84/109 (77.1%)	60/123 (48.8%)	20/22 (90.9%)
exchange2_r	Fortran	1478	13	13 (100.0%)	8 (61.5%)	0 (0.0%)	12 (92.3%)	12/13 (92.3%)	15/15 (100.0%)	0/0 (-)
fotonik3d_r	Fortran	8887	40	40 (100.0%)	24 (60.0%)	4 (10.0%)	39 (97.5%)	38/40 (95.0%)	32/34 (94.1%)	7/7 (100.0%)
gcc_r	C	972K	12123	12121 (100.0%)	2543 (21.0%)	383 (3.2%)	10287 (84.9%)	10029/12123 (82.7%)	10790/13322 (81.0%)	3350/3693 (90.7%)
imagick_r	C	174K	2077	2073 (99.8%)	553 (26.6%)	288 (13.9%)	1617 (77.9%)	1602/2077 (77.1%)	1688/2233 (75.6%)	257/477 (53.9%)
lbm_r	C	1037	18	18 (100.0%)	5 (27.8%)	0 (0.0%)	18 (100.0%)	14/18 (77.8%)	8/14 (57.1%)	3/4 (75.0%)
leela_r	C++	31K	382	372 (97.4%)	85 (22.3%)	3 (0.8%)	287 (75.1%)	297/382 (77.7%)	205/365 (56.2%)	73/80 (91.2%)
mcf_r	C	2669	38	38 (100.0%)	4 (10.5%)	2 (5.3%)	31 (81.6%)	33/38 (86.8%)	24/44 (54.5%)	0/0 (-)
nab_r	C	16K	222	220 (99.1%)	54 (24.3%)	29 (13.1%)	198 (89.2%)	160/222 (72.1%)	178/225 (79.1%)	13/21 (61.9%)
namd_r	C++	6396	124	120 (96.8%)	19 (15.3%)	7 (5.6%)	121 (97.6%)	50/124 (40.3%)	40/117 (34.2%)	17/72 (23.6%)
omnetpp_r	C++	86K	6146	5915 (96.2%)	838 (13.6%)	99 (1.6%)	4802 (78.1%)	4914/6146 (80.0%)	3852/5272 (73.1%)	2435/2789 (87.3%)
perlbench_r	C	291K	2395	2322 (97.0%)	623 (26.0%)	136 (5.7%)	2130 (88.9%)	1966/2395 (82.1%)	2270/2598 (87.4%)	483/577 (83.7%)
povray_r	C++	81K	1541	1530 (99.3%)	354 (23.0%)	48 (3.1%)	1340 (87.0%)	949/1541 (61.6%)	997/1540 (64.7%)	306/359 (85.2%)
roms_r	Fortran	220K	81	81 (100.0%)	48 (59.3%)	8 (9.9%)	78 (96.3%)	78/81 (96.3%)	68/82 (82.9%)	29/29 (100.0%)
wrf_r	Fortran	494K	7921	7885 (99.5%)	1615 (20.4%)	673 (8.5%)	6817 (86.1%)	6710/7921 (84.7%)	3171/7899 (40.1%)	1440/1570 (91.7%)
x264_r	C	73K	1221	1216 (99.6%)	185 (15.2%)	37 (3.0%)	1075 (88.0%)	921/1221 (75.4%)	744/1224 (60.8%)	183/216 (84.7%)
xalancbmk_r	C++	295K	13761	13251 (96.3%)	2356 (17.1%)	35 (0.3%)	9842 (71.5%)	10538/13761 (76.6%)	9300/12423 (74.9%)	2308/3306 (69.8%)
xz_r	C	20K	369	362 (98.1%)	64 (17.3%)	21 (5.7%)	269 (72.9%)	276/369 (74.8%)	247/433 (57.0%)	62/103 (60.2%)
(average)	C	194K	2307.9	99.2%	21.1%	6.2%	85.4%	78.6%	69.1%	72.9%
	C++	319K	8461.4	97.8%	18.5%	3.6%	78.1%	71.1%	57.5%	74.6%
	Fortran	163K	1867.7	99.8%	42.3%	8.3%	90.9%	89.4%	80.8%	93.6%
(average)	(all)	224K	4125.8	99.0%	27.0%	6.0%	84.8%	79.7%	69.1%	79.7%

The canaries in the strong mode protect 27% of all functions on average. The percentage in C/C++ programs is a bit lower (19.9% average). In the basic (default) mode, on average 6% of all functions get protected (5% in C/C++ programs). The majority of functions remain unprotected, reducing the performance penalty of the protection. These numbers align with related work: Molnar et al. [35] reported that 20.5% of all Linux kernel functions are protected in strong mode, and 2.8% of all functions are protected in basic mode.

Our Optimized Mode. In contrast, our optimized mode protects 85% of all functions (82% in C/C++ programs). There are two main differences between the relaxed canary modes and the optimized shadow stack and return address encryption modes: language knowledge and security. First, the compiler has much more source information to determine if a piece of code is safe or not. It can use detailed information about stack variables, layout, and variable usage. In contrast, our optimization is assembly-based and can't prove a function safe if pointer arithmetic is used for write access. Second, stack canaries have a weaker level of protection which is threatened by fewer functions. Canaries protect only against consecutive overwrites; therefore, a canary in an unsafe function protects all functions higher in the call stack. Furthermore, only specific primitives like arrays are prone to overflow attacks. In contrast, shadow stacks protect against all kinds of overwrites, no matter if the attack was started on local variables or in the local function. Therefore, all functions that might be memory-unsafe must be protected, and any function that calls memory-unsafe functions must be protected, too. Our optimization maintains these higher security properties without trading security for performance.

Scratch Registers. Finally, we analyze the availability of scratch registers in function prologues (where the function starts), before a function returns, and before the function exists by jumping into another function (tail calls). 80% of all prologues have a scratch register available, while only 69% of all returns have one available. In the remaining cases, we need two additional instructions. Before our inserted code, we save the scratch register's value to the stack with `mov [rsp-8], r11`. After our code, we restore this value from the stack with `mov r11, [rsp-8]`. 80% of all tail calls have a scratch register available. On average, we find fewer scratch registers in C++ programs and more scratch registers for Fortran code. We also note that scratch register discovery works especially bad for one particular SPEC2017 target: *namd_r*—on 40% of all prologues and 34% of all epilogues have scratch registers available.

4.7.6 Evaluating Security

Comparison. We already summarized the security level of our three protections in their initial description; see Section 4.5: Stack canaries are strictly weaker than shadow stacks and return address encryption because they only protect against consecutive overwrites like a buffer overflow. In contrast, shadow stacks and return address encryption protect against any modification of the return address. Defeating canaries is easiest: the attacker must leak the canary from the stack. Defeating return address encryption is more challenging: the attacker must not only leak an encrypted return address from the stack but also leak enough information to compute (parts of) the unencrypted return address. If the attacker knows both encrypted and unencrypted return addresses, he can recompute the secret key k and break the protection in subsequent attacks. Defeating shadow stacks is the hardest:

the attacker must be able to write multiple values to arbitrary addresses during a single function execution, i.e., overwrite the return address on both local and shadow stacks. The attacker must also leak the random offset between the default stack and shadow stack if it is randomized. In contrast to the required leaks from all other schemes, this information is likely not stored on the stack.

Security of Modes. All protections can be applied to a subset of functions for performance reasons, see Section 4.7.4. In particular, stack canaries have very relaxed modes that protect only 6% (basic) or 27% (strong) of all functions. In the basic mode, stack canaries lose parts of their security. Many vulnerabilities are not covered, particularly vulnerabilities involving more than one function or small buffers. In the strong mode, stack canaries retain most of their security. Only a few rare and very specific vulnerabilities can target the now unprotected functions. For example, when handling a buffer at a higher stack address, an integer overflow might allow attackers to corrupt the return address of unrelated functions—functions that might not have a canary even in strong mode.

In principle, the shadow stacks and return address encryption could ignore the same functions and obtain similar guarantees. However, this partial mode would remove all the additional protection compared to stack canaries. Shadow stacks and return address encryption include protection against arbitrary writes, in contrast to stack canaries, which protect against consecutive overwrites only. This higher level of protection is only enforceable if all return addresses on the stack are covered when memory corruption occurs—in contrast to stack canaries, where only the return address nearest to the corruption must be covered.

For example, let us assume a function that calls other functions but does not use stack variables. The canary in a called function will stop any consecutive overwrite in the called function—the called function does not threaten the calling function's stack frame. Thus, the calling function does not need a canary if it does not write to the stack itself. This is different for protections that are not limited to consecutive overwrites. If the called function can selectively overwrite the return address of the calling function, then the calling function must also protect its return address. Thus, protecting only the subset of functions like the strong canary mode would weaken the security of shadow stacks and return address encryption; it is not applicable for stronger protections.

We proposed a custom optimization mode for shadow stacks and return address encryption that ignores functions only if no unsafe memory write can occur, including called functions. In contrast to optimized canaries, we protect any function that call other functions. Thus, only the most recent stackframe can contain an unprotected return address, and this return address can only be unprotected while memory-safe instructions are executed. To this end, the optimized shadow stack and optimized return address protections do not lose security compared to their unprotected versions.

TOCTTOU. There is one potential threat remaining in our implementations: *time-of-check-to-time-of-use* (TOCTTOU) vulnerabilities. In principle, an attacker can try to change the return address on the stack after the protection's checks ran but before the actual return executes. If the timing fits, the protection's check succeeds, but the return still jumps to an attacker-controlled address. A second TOCTTOU vulnerability opens in the function prologue: An attacker can try to modify the stored return address between `call` instruction and the protection's prologue when the protection is not yet active. If successful, the protection will consequently check for the attacker-modified address later.

Stack canaries do not have to consider TOCTTOU attacks. They only protect against stack-based buffer overflows. Having a buffer overflow on one stack corrupt a return address on another thread's stack is impossible; therefore, concurrent modifications are no additional issue for stack canaries.

We decided not to defend against TOCTTOU attacks for shadow stacks and return address encryption because such a defense would come with performance and compatibility problems. The risk of a successful TOCTTOU attack is low; we would consider it negligible. Most stack-based attacks start from variables and arguments on one stack and only affect this stack or potentially known global addresses (e.g., global data or heap). It is unlikely that an application shares a stack address with structures outside of this stack. While it is theoretically possible that a vulnerability on one stack can be re-purposed to target leaked addresses of other stacks, this is a very unusual scenario. Next, TOCTTOU attacks require *very* precise timing. The checks read the return address from the stack to check it only a few instructions before the actual return. The actual return will execute only a few cycles after the check and will be fast itself—it has the return address already in L1-cache. A concurrent modification would not only require perfect timing down to cycle level but also a way to invalidate or flush the attacked thread's CPU caches. Finally, there is only one try for a TOCTTOU attack—if the attack starts a few cycles too early, the whole program will crash. Even for local attackers, TOCTTOU attacks are often impossible to execute. On the contrary, defending against TOCTTOU is possible but has a huge performance overhead that hinders the adoption of a protection scheme. Dang. et al. [29] suggest replacing the final `ret` with a `jmp` that reads the checked return address from a register, which could prevent TOCTTOU attacks in the function epilogue. This change costs an additional 4.7% of performance (on SPEC CPU 2006, excluding Fortran) and is incompatible with certain protections (such as Intel CET's [136] indirect branch tracking or the "retpoline" defense against microarchitectural attacks). Defending against the TOCTTOU possibility in the prologue is even harder—the only possible defense is a change in the calling convention, where the original return address is passed to the callee by register or where the protection's prologue is moved to the caller. These changes make protected code inherently incompatible with unprotected code, further hindering its adoption.

We conclude that timing-based circumvention of the protections are possible in theory. Still, their probability is negligibly low and does not justify strong performance and compatibility penalties that can prevent adoption. We will thus measure the protection's performance without any TOCTTOU-specific changes.

4.8 Performance Evaluation

4.8.1 Methodology

We run all benchmarks on an Intel i5 4690 CPU (4 × 3.5GHz) with 32GB of memory. We store benchmarks on a SSD drive; initial measurements with am in-memory storage showed that disk speed does not affect performance measurements. We use Debian 11 with Linux kernel 5.10 as the operating system.

Our benchmark environment should generate sound, reproducible results: First, we disabled all unnecessary services during benchmarks to avoid any influences from external

software. Furthermore, we use `cpuset` [87] to shield one CPU core—only the benchmark itself and necessary kernel threads can use this core. We also increased the process priority of benchmarks to their maximal value, avoiding any slowdown from potential other activity. The operating system is configured to give each benchmark run the same initial state: before each run, all caches are flushed, and ASLR has been disabled so that memory addresses are reproducible. We set the CPU scheduler to "performance" and disabled CPU turbo boost—the benchmarking CPU core runs at a constant frequency. We further fixed the clock frequency to 3.0GHz (out of the possible 3.5GHz) and monitored it constantly to avoid any thermal or power-based throttling.

The `runcpu` command from the SPEC CPU benchmark suite runs every program three times and reports the median runtime of these three runs. The median should avoid outliers in the measured numbers—although this is not enough in some cases, as we show in Section 4.8.2. The benchmark also checks the output of the programs, ensuring that the protections do not cause any error or behavior change. We tested all compiler-based protection mechanisms with at least 10 complete SPEC CPU 2017 benchmark runs; thus, we execute each program 30 times per protection. In total, we spend more than 38 days executing SPEC benchmarks. We compared the average runtime of each benchmark program with the wall clock time an unmodified benchmark run required. The overhead of a protection mechanism is then calculated per program by dividing the *additional* time by the unmodified benchmark run time:

$$\text{overhead} := \frac{t_{protected} - t_{base}}{t_{base}}$$

This factor expresses how much more time a computation will take if the protection mechanism is applied; it is commonly used in system security research.

We are confident that our setup, benchmarks, and evaluation methodology conform to the best practices in performance measurement, such as outlined in related work [57, 77]. Our setup also follows all benchmarking suggestions from the LLVM project [93].

4.8.2 Measurement Soundness

To check our measured result for validity, we review the standard deviation of all results. If results vary too much between runs, this variation could indicate a problem with our benchmarks. It quickly turns out that most benchmarks have a pretty low standard deviation of 0.02% up to 0.27% of their respective runtime. However, two benchmarks stood out: omnetpp (up to 3.55% standard deviation) and xalancbmk (up to 2.67% standard deviation). In absolute numbers, runtime changed about 30–40 seconds in a 380–480 seconds execution.

First, we triple-checked our setup for potential problems or overseen sources of randomness and asked fellow researchers for advice. We found no problems. Then we used `perf` to profile the benchmarks in question. We did not find any meaningful insight. In particular, the correlation between cache performance and runtime was weak. We further looked for problems or overseen features in the hardware. We re-ran the two problematic benchmarks on two more different machines to see if we could reproduce the randomness in the results. We saw the same variance in the results on a modern Intel i7 9700K (4×3.6GHz) from 2018 with 64GB of memory. We did not see any variance on an old Intel Q6600 (4×2.4GHz)

from 2007 with 8GB of memory, even if booted with a one-to-one copy of our primary benchmarking system. We thus have the justified assumption that any hardware feature in modern CPUs can either improve or slow down these two benchmarks considerably. Without more profound knowledge about the CPU internals and without a possibility to debug them, we can neither prove this assumption nor fix the underlying issue.

To get reasonable, comparable, and valid results, we use mathematical methods to clean up the measured results of the two affected benchmarks. In our analysis, we will consider no absolute runtime but overheads, i.e., the absolute runtime is not too important, as long as the relative runtime between different runs is not affected. To this end, we must ensure that any applied method is applied equally to the results of each protection; no protection must be preferred or disadvantaged.

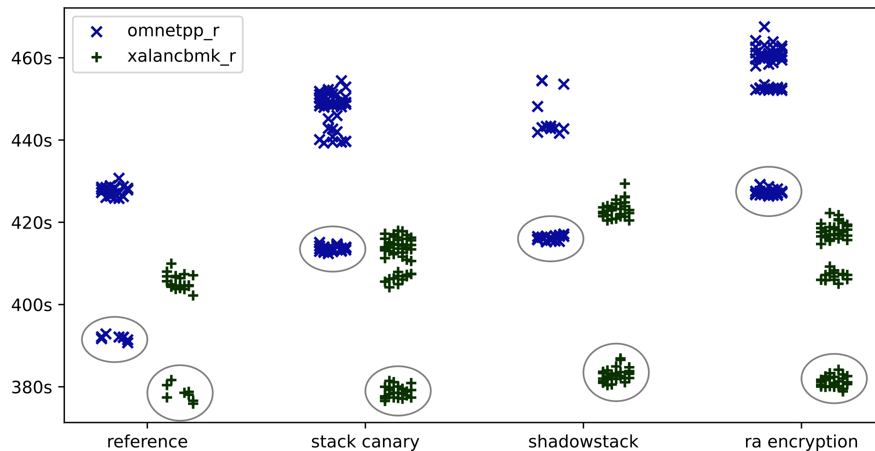


Figure 4.5: All benchmark runtime results of *omnetpp* and *xalancbmk*. Circles show the cluster that remains after data cleaning.

When inspecting the raw results for the two benchmarks in Figure 4.5, we notice a pattern in all results: In both benchmarks, the results are grouped into two clusters. For example, the reference benchmark for omnetpp seems to take either around 395 or around 430 seconds, with no results in between. From this observation, we build the justified assumption that there is an unknown source of overhead in the benchmarking system, which triggers not in every benchmark run. We can, however, see for each result if the overhead's condition has been triggered or not.

With this knowledge, we decided to pick each protection's lower cluster for these two benchmarks and drop the remaining results. This result selection is analogous to the minimum runtime selection that is often used in smaller benchmarks [22]. We repeated these two benchmarks until the lower cluster had at least 30 results—comparable to other benchmarks. To this end, we did 30–85 additional executions per benchmark and protection. Finally, we have a standard deviation of 0.19% for omnetpp and 0.35% for xalancbmk, which is in line with the other results.

Is this fair? From our initial inspection, the clustering and the difference between the clusters were similar for all protections; thus, no protection was preferred or disadvantaged. The two clustered benchmarks have the same number of considered runs as all other benchmarks. The standard deviations between all benchmarks are in line, so no benchmark's

variation can influence the final overhead results disproportionately. We conclude that cluster-based filtering of the two benchmarks is fair concerning the overall result and yields proper overhead measurements.

For transparency, we list the standard deviation (σ) of all benchmarks in Table 4.2. The majority has a standard deviation of 0.07% or lower, so we can see one digit after the point as significant.

4.8.3 Performance Overhead on SPEC

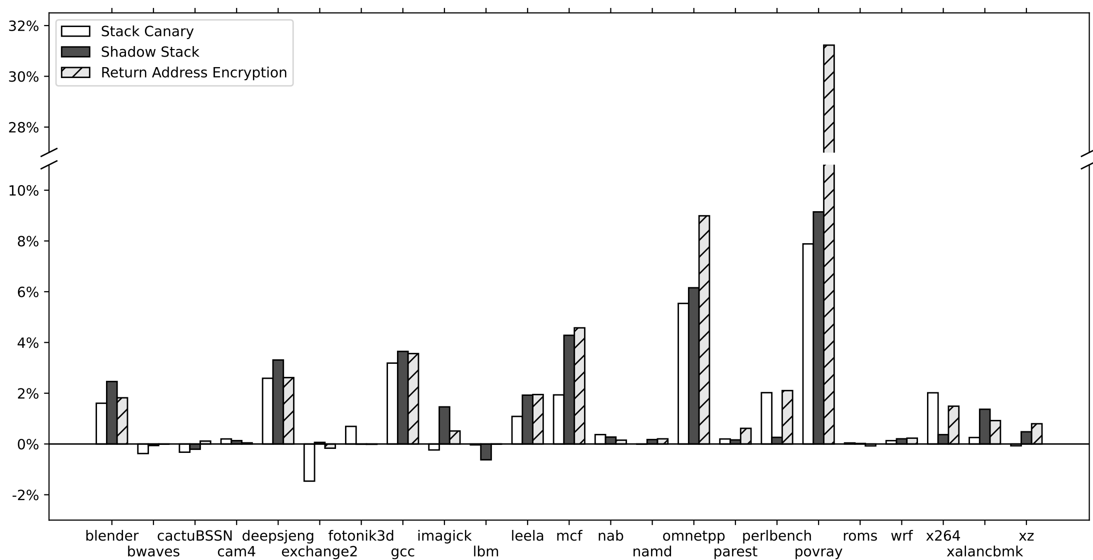


Figure 4.6: Overhead of all protections on all benchmarks.

Figure 4.6 shows the overhead of all protections on each benchmarked program; and Table 4.2 contains the raw results. First, we notice that all benchmarks apart from omnetpp and povray experience less than 5% overhead, which is considered tolerable, for example, by Microsoft [100]. The mean overheads range from 1.2% (stack canaries) over 1.5% (shadow stacks) to 2.5% for return address encryption. The median overhead of any protection is at most 0.6%, which we'll detail later.

The worst case for all protections is *povray*: stack canaries have 7.9% overhead, shadow stacks 9.1%, and return address encryption has an unbearable overhead of 31%. We noticed that povray's source code contains many small or empty functions where function inlining optimizations are not applied. These functions are called very often, and their instrumentation is executed very often, which explains the high overall performance impact. However, we cannot explain the high overhead of return address encryption compared to the overhead of a shadow stack.

When we compare the protections with each other, we quickly see that stack canaries are the fastest protection, and return address encryption is often the slowest. The difference between stack canary and shadow stack is often small, except for *imagick* (+1.7%) and *mcf* (+2.4%). When we compare shadow stack performance to return address encryption, we notice that shadow stacks are generally faster. In particular, *perlbench* shows this: return

Table 4.2: Overhead of all protections on all benchmarks.

Benchmark	Language	Stack Canary			Shadow Stack		RA Encryption		σ
		(all)	(strong)	(basic)	(all)	(opt)	(all)	(opt)	
blender_r	C++	1.6%	0.7%	1.4%	2.5%	1.7%	1.8%	1.7%	0.09%
bwaves_r	Fortran	-0.4%	-0.2%	-0.2%	-0.1%	-0.1%	-0.0%	-0.2%	0.13%
cactuBSSN_r	Fortran	-0.3%	0.9%	-0.5%	-0.2%	-0.0%	0.1%	0.1%	0.20%
cam4_r	Fortran	0.2%	0.1%	-0.1%	0.1%	0.2%	0.0%	0.0%	0.17%
deepsjeng_r	C++	2.6%	0.1%	0.2%	3.3%	0.6%	2.6%	1.7%	0.04%
exchange2_r	Fortran	-1.5%	-0.3%	-0.1%	0.1%	0.0%	-0.2%	-0.2%	0.04%
fotonik3d_r	Fortran	0.7%	0.0%	-0.0%	-0.0%	-0.1%	-0.0%	-0.0%	0.05%
gcc_r	C	3.2%	0.7%	-0.2%	3.6%	3.6%	3.6%	4.1%	0.07%
imagick_r	C	-0.2%	0.3%	1.1%	1.5%	1.4%	0.5%	1.2%	0.02%
lbm_r	C	-0.0%	-0.1%	-0.1%	-0.6%	-0.9%	-0.0%	-0.8%	0.04%
leela_r	C++	1.1%	2.4%	0.7%	1.9%	1.8%	1.9%	2.1%	0.01%
mcf_r	C	1.9%	-0.0%	-0.0%	4.3%	-0.1%	4.6%	-0.1%	0.08%
nab_r	C	0.4%	-0.0%	-0.0%	0.3%	0.5%	0.2%	0.3%	0.01%
namd_r	C++	-0.0%	-0.2%	-0.1%	0.2%	0.3%	0.2%	0.2%	0.03%
omnetpp_r	C++	5.5%	2.9%	0.9%	6.2%	4.2%	9.0%	5.6%	0.19%
parest_r	C++	0.2%	-0.2%	-0.2%	0.2%	-0.4%	0.6%	-0.1%	0.06%
perlbench_r	C	2.0%	-0.5%	-0.2%	0.3%	2.2%	2.1%	2.3%	0.19%
povray_r	C++	7.9%	3.7%	-0.1%	9.1%	7.5%	31.2%	29.5%	0.12%
roms_r	Fortran	0.0%	0.1%	-0.3%	0.0%	0.1%	-0.1%	-0.2%	0.27%
wrf_r	Fortran	0.1%	-0.0%	-0.0%	0.2%	0.1%	0.2%	0.3%	0.11%
x264_r	C	2.0%	-1.5%	-1.4%	0.4%	0.8%	1.5%	0.7%	0.04%
xalancbmk_r	C++	0.3%	-1.2%	-0.6%	1.4%	-2.0%	0.9%	-1.8%	0.35%
xz_r	C	-0.1%	-0.1%	0.4%	0.5%	0.5%	0.8%	0.9%	0.07%
(mean)		1.2%	0.3%	0.0%	1.5%	0.9%	2.5%	1.9%	0.07%
(median)		0.3%	-0.0%	-0.1%	0.3%	0.3%	0.6%	0.3%	0.07%
(max)		7.9%	3.7%	1.4%	9.1%	7.5%	31.2%	29.5%	0.35%

address encryption has 2.1% overhead compared to 0.3% for shadow stacks—a 7× increase. On *x264*, the overhead of return address encryption is 3× the overhead of a shadow stack. Return address encryption outperforms shadow stacks on three of our 23 benchmarks: *blender*, *deepsjeng*, *imagick*.

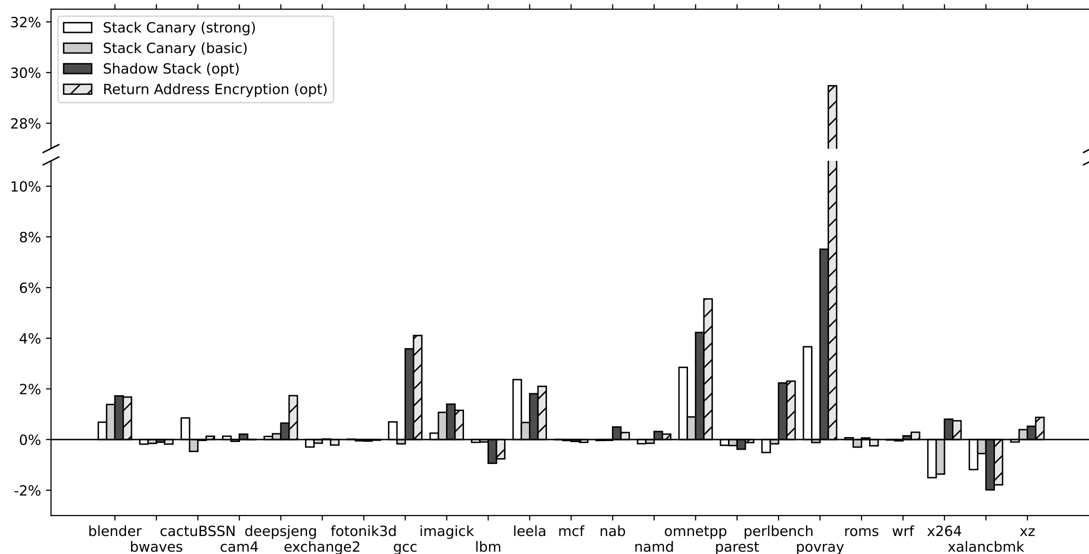


Figure 4.7: Overhead of all relaxed and optimized protections on all benchmarks.

Optimized Protections. Figure 4.7 shows the overhead of the partial stack canary modes (`-fstack-protector-strong` and the basic `-fstack-protector`) and our optimized versions of shadow stack and return address encryption. Again, Table 4.2 contains the raw numbers.

We directly see that the partial stack canary modes do a great job in reducing performance penalties. The basic mode has no overhead on average, and even the strong mode has only 0.3% overhead. Even programs with higher stack canary overhead such as *gcc*, *omnetpp*, or *povray* lose their overhead almost entirely in basic mode. The only cases where basic mode produces a considerable overhead are *blender* (1.4%) and *imagick* (1.1%). In both cases, the overhead likely comes from the stack and code layout changes only—because the strong mode, which protects more functions, has less overhead (0.7% and 0.3%). The strong mode reduces overheads considerably but not always completely.

Our optimized shadow stack and return address encryption modes reduce the overhead of their base schemes by 0.6% mean, but not as well as the partial canary modes. This is as expected because partial canary modes include 6%–27% of all functions, while the optimized mode includes 85% of all functions (in turn, without sacrificing any security). The optimized mode cannot fix the performance of return address encryption on *povray*, but removes the 4% overhead on *mcf* almost entirely. The optimization affects both shadow stack and return address encryption equally on most programs. The only significant difference is *deepsjeng*, where the optimization removes most of the shadow stack overhead but only partially increases the performance of return address encryption. There are two exceptional cases where the optimization leads to a significantly worse performance: optimized shadow stack on *perlbench* gives an additional 1.9% overhead, and optimized return address encryption

on *gcc* adds 0.5% overhead. We assume that the non-optimized protection hit a favorable code layout by chance, which is realigned when the optimization drops some instructions. We also notice that the optimized shadow stack has less mean overhead than full stack canaries while offering a higher level of security. Finally, we notice that *imagick* receives a significant performance penalty for all optimizations—including the partial canary modes.

Negative Overhead. Some protections seem to have a negative overhead on some benchmarks. This is counter-intuitive because each protection only adds more instructions to functions but never removes work. In particular, stack canaries in their strong or basic modes improve many programs by a tiny percentage. Changes in the stack layout introduced by the additional canary can have such an effect: the alignment of function frames on the stack changes, so data that was split between two cache lines before can be on the same cache line after layout changes (and vice versa). Related literature [121] shows that changing the stack offset is already enough to provoke this effect (1–4% derivation). Changing the offset of each function can have the even bigger effects we see here. But this effect also applies in the real world; any attempt to exclude it from our measurements would render them less practical. Some benchmarks also show a minimal negative overhead (up to −0.2%) with shadow stack or return address encryption. Still, this slight negative overhead is within the standard deviation and can be seen as an artifact of measuring real systems. An outlier is *lbm*: It has a non-negative overhead for all protections while the benchmark itself is exceptionally stable; the worst standard deviation we saw was 0.05%. In particular, the optimized shadow stack and return address encryption get considerably faster (up to −0.9%, 2 seconds). We suspect that the additional instructions change the code layout and trigger a similar effect to the stack layout changes: frequently called functions might be placed in better cache positions. In line with related work [121], we used the average of all 23 benchmarks so that the impact of code or memory layout changes can make up for each other. *xalancbmk* shows a special case: apparently, the unprotected reference binary has an unfortunate code layout. Almost all protections that change the code layout as a side effect hit a more fortunate layout and therefore experience a speedup. This speedup is more significant than the slowdown of the protection itself for the partial canary modes and the optimized protections.

4.8.4 Overhead per programming language

We found interesting insights when we grouped benchmarks by their primary programming language. From Figure 4.8 and Table 4.3, we can see the influence of the programming language of a benchmark.

Fortran-based benchmarks experience little to zero overhead on all protections. In particular, the protections without stack layout changes (shadow stack and return address encryption) change the performance of a Fortran program by at most ±0.2%. For stack canaries, the result is similar with two outliers due to stack layout changes: *exchange2* receives a 1.5% speedup, and *fotonik3d* receives a 0.7% slowdown. The reason is that Fortran programs typically execute much fewer function calls. Even benchmarks that combine Fortran with C or C++ code show this behavior, as long as the program's main computation is written in Fortran.

When comparing C and C++ programs, we see that all protections have more mean

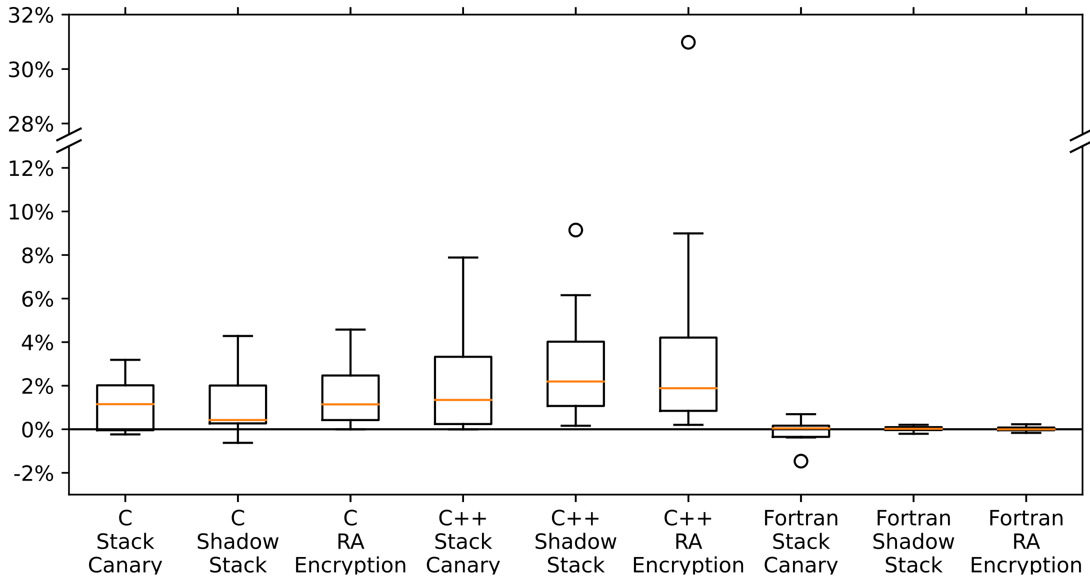


Figure 4.8: Overhead per programming language.

overhead on C++ programs. C++ idioms like inheritance and virtual functions (see Section 5.4.1) produce many smaller functions that cannot be inlined by compiler optimizations because they are not directly called but referenced in a table. Many small functions mean many function prologues and epilogues that need protection. Therefore, all protections have at least double mean overhead on C++ benchmarks than on C benchmarks. The difference is big enough to make a difference in adoption. All protections have an overhead lower than the stated 5% [100] on C benchmarks; down to 3.6% if optimization is enabled. But each protection has at least two C++ benchmarks where its overhead is above 5%, which is too much for general real-world adoption.

We conclude that the programming language of a benchmark is highly relevant for a protection's overhead. For the evaluation of any protection, the results will differ depending on the programming languages used in the evaluation. Related work often omits Fortran programs from evaluations, thus losing generality.

4.8.5 Overhead Comparison With Literature

Finally, we compare our results to the performance overhead of the same mechanisms reported in the literature. Some factors can make our results different than older work: First, we use 64bit x86 machines for our benchmarks, while related work might use 32bit x86 machines. For example, in the most recent evaluation, Dang et al. [29] tested both but reported 32bit only, claiming that the overhead between 32bit and 64bit is comparable, but 32bit provides "a lower bound." Second, many related work use older benchmarks like SPEC CPU 2006 and exclude Fortran programs. We have shown earlier that Fortran programs have a different overhead than C/C++ by design. In Table 4.3, we also report our measured overheads on C/C++ only to be more comparable. Third, related work often uses much older compilers and hardware, which can deal better or worse with the added protection code. We intended to capture these differences with this work.

Table 4.3: Mean overhead of all protections per language and summary of overheads.

Protection	Overhead per language			Overhead	Overhead
	C	C++	Fortran	(C/C++)	(total)
Stack Canaries	1.1%	2.4%	-0.2%	1.7%	1.2%
(strong)	-0.2%	1.0%	0.1%	0.4%	0.3%
(basic)	-0.1%	0.3%	-0.2%	0.1%	0.0%
Shadow Stack	1.3%	3.0%	0.0%	2.1%	1.5%
(optimized)	1.0%	1.7%	0.0%	1.3%	0.9%
RA Encryption	1.6%	5.6%	0.0%	3.6%	2.5%
(optimized)	1.1%	4.5%	0.0%	2.8%	1.9%

Our measured stack canary has a mean overhead of 1.1% on C and 2.4% on C++. Combined C and C++ overhead is 1.7%. Thus stack canaries are faster than the results from Dang et al. [29], who reported a 2.54% average overhead on C/C++. In particular, Dang et al. reported that their fastest shadow stack was slower than stack canaries. This is no longer true; our optimized shadow stack has less mean overhead than full stack canaries on C and C++. In contrast, Szekeres et al. [164] report that the stack canaries' overhead "is negligible (less than 1%)" without further justification, showing a discrepancy. We can confirm these statements for the partial modes strong/basic only, which are more likely used in practice.

Our parallel shadow stack implementation has a mean overhead of 1.3% on C to 3% on C++, combined 2.1%. These results are lower than the original parallel shadow stack evaluation [29], which measures 3.5% average overhead on C/C++ programs, but did not further split the overhead by programming languages. In turn, [29] has summarized more related work using shadow stacks. If evaluated on any SPEC benchmark, all these implementations had overheads higher than 5%. Szekeres et al. [164] reported 5% average overhead on their shadow stack implementation. We cannot be sure if our implementation compares to their design without a description of how they implemented the shadow stack in their paper.

The third protection mechanism, return address encryption, was never evaluated in isolation by related work. Thus we can hardly compare our results with the literature. It was proposed as part of a bigger system, G-Free [133], which has 3.1% overhead in total on some non-standardized real-world C programs. Our results of 1.6% overhead on C benchmarks do not contradict these findings, at least.

Finally, we can say that the return address protections have less overhead than stated earlier on modern software using modern compilers and modern hardware. This discovery could lead programmers to re-evaluate and potentially deploy stronger defenses with their applications.

Table 4.4: Mean binary size overhead of all protections per language and summary of additional binary size.

Protection	Binary size per language			Size Overhead	Size Overhead
	C	C++	Fortran	(C/C++)	(total)
Stack Canaries	3.6%	6.5%	1.9%	5.0%	4.1%
(strong)	0.9%	1.3%	1.2%	1.1%	1.1%
(basic)	0.6%	0.1%	0.0%	0.4%	0.3%
Shadow Stack	3.4%	6.5%	1.4%	4.9%	3.8%
(optimized)	2.5%	4.3%	1.3%	3.4%	2.7%
RA Encryption	2.9%	7.4%	1.4%	5.2%	4.0%
(optimized)	2.4%	5.8%	1.3%	4.1%	3.2%

4.8.6 Binary Size Overhead

In Table 4.4, we show a summary of the protection's size overhead. All protections add instructions to the programs, which make them larger. We state the relative number of additional sizes each protection introduces. This additional size includes a small runtime library for shadow stacks and the patched libunwind for return address protection.

When applied to all functions, all protections have a similar overhead of 3.8% up to 4.1%. The number of still protected functions mainly determines the size overhead of partial or optimized modes. The strong canary mode protects only 6%–20% of all functions, which is proportional to the saved size. The optimized modes protect 85% of all functions; they do not save much binary size.

When analyzed per programming language, the size overhead shows the same characteristics as the performance overhead. C++ programs increase most, up to 7.4% for return address encryption. C programs increase less, but still by some reasonable amount (2.9%–3.6%). Fortran programs seem less affected by additional binary size than other languages. In contrast to performance, the binary size of Fortran programs still sees an impact up to 1.9%.

Today, storage on x86 systems is rather cheap, so is sufficient transmission capacity. A few percent increase in binary size will likely not be noticed by most users. In particular, more and more software comes with statically linked dependencies, increasing application size a lot more. We conclude that the increased size of protected applications does not prevent adoption.

4.9 Compatibility Evaluation

Szekerez et al. [164] state that performance and cost overhead is not the only criterion relevant for adoption; one must also consider a protection's compatibility with existing programs and systems. They divide this requirement further into three categories:
Source Compatibility. The protection must not require changes to the source code of an

application. This requirement is satisfied by all protections; they are applied entirely in the build pipeline. Developers only have to configure the compiler properly.

Binary Compatibility. Protections must be able to use unmodified, legacy libraries or system libraries. All protections satisfy this requirement. In particular, a protected application can mix protected and unprotected libraries. Developers and maintainers can incrementally deploy each protection.

Modularity Support. Individual modules at compile- or link-time should be independent of each other's protection status. For all protections, this is implementation-dependent. In general, all protections are applied on a per-function level. Protected and unprotected functions can call each other without changes to the calling convention and, in particular, without knowing at compile time if the called function is protected or not. We can freely combine compilation units, static libraries, and dynamically loaded modules—as long as the protection is properly initialized.

Initialization. The initialization is the only potential compatibility problem for all protections. Stack canaries require a secret canary value in the thread-local storage of each thread. The current Linux implementation initializes this value in the system's dynamic loader. This loader loads every application; therefore, canaries are properly initialized in any process.

Return Address Encryption also needs an initialized secret value in the thread-local storage. It can be initialized by the program or the dynamic loader but not by a dynamic library. The TLS value must be initialized before the program starts any additional threads. However, the program can load dynamic libraries anytime, so a protected library loaded by an unprotected application might find itself in an uninitialized thread. There are multiple possible solutions for this compatibility problem: First, the secret value could be per library instead of per thread. Then each protected library can initialize a custom, local secret when the library is loaded. Second, the system's dynamic loader could generate the secret—all applications would start with a properly initialized secret. Third, return address encryption might use the stack canary value as a secret, whose initialization is already widely deployed. However, the canary is not entirely random; 24 out of 64 bits are always fixed. The attacker's probability of guessing the correct key is higher than necessary. Depending on the concrete use case, any of these solutions can make return address encryption fully compatible with unprotected modules.

Shadow Stacks require the program to initialize a second stack for each thread. The limits are thus similar to return address encryption: either the dynamic loader or the application can create shadow stacks, but not dynamic libraries. The only clean solution is a modification to the system's dynamic loader or the C standard library. Other solutions fail if a protected dynamic library is loaded after the application has started, additional threads have already been created at this point, and the library's protected functions are used in different threads. A less clean compatibility fix could rewrite the program to add shadow stack initialization to the main program or use `LD_PRELOAD` to add load time initialization.

To summarize, stack canaries are highly compatible, mainly because they have seen widespread support in existing systems. Return address encryption achieves a similar level of compatibility with a proper implementation. Shadow stacks still have minor compatibility problems whose possible fixes depend on the concrete situation.

4.10 Excursion: Return Address Protections on ARM

Besides 64-bit x86, the 64-bit ARM architecture also plays an important role today. ARM is mainly used in mobile devices such as smartphones and in recent Apple PCs. On ARM, the situation for return address protection is different because return addresses are not stored on the stack by default. A call on ARM with instructions `bl` or `blr` stores the return address in the register `x30`, not on the stack. In turn, the return instruction `ret` uses the address in register `x30` (or any other register, if necessary). So a call or return does not necessarily need protection.

Return address protection can only be applicable when the return address in register `x30` is temporarily spilled on the stack, for example, because the function calls other functions or register pressure becomes too high. Register spilling often happens in the function prologue and epilogue, but this is not required. A compiler may decide to spill the register only on the stack for specific control flow paths. Furthermore, the compiler may use a different register than `x30` for the return address. All this information is required to build a performant return address protection. They are only available in the compiler; therefore, protections for ARM should be implemented in the various compilers, not as an assembly rewriter.

On the other hand, return address protections on ARM likely have lower performance overhead and higher security. All functions that do not call other functions and do not need too many registers do not need protection and do not have performance overhead. When implementing a protection like a shadow stack or return address encryption, we do not have to read the return address from the stack initially and have it already in a register after the final check. For a shadow stack, we could even omit the spilled register on the real stack and use the shadow stack only. Finally, a properly implemented return address protection for ARM would not have any TOCTTOU issues—there is no trade-off between performance and TOCTTOU protection.

Clang already contains a production-ready shadow stack implementation for 64-bit ARM, called ShadowCallStack [176]. The necessary runtime is included in Android's C standard library; other platforms must provide a custom shadow stack runtime. From version 8.3 on, a hardware-assisted return address protection based on Pointer Authentication [142] can be implemented. Return addresses are signed in the function prologue and the signature is checked before returning. The signature key is managed by the CPU and thus unaccessible to attackers.

4.11 Conclusion

In this chapter, we have analyzed three different return address protections. We have used modern compilers, benchmarks, and hardware to get results applicable in this decade. We showed that all protections have acceptable performance and size overheads on most programs, and we revealed an exceptional case: C++ code with many small methods like *povray*. In particular, we showed that optimized shadow stacks beat full stack canaries not only in terms of security, but also performance. The overhead of our shadow stack implementation on modern software is lower than expected by related work. Thus, we suggest the integration of fast, optimized software-only shadow stacks in compilers and the development of a commonly accepted shadow stack runtime library. We further suggest that

maintainers of high-risk applications evaluate software-only shadow stacks on their systems as an alternative to stack canaries. With the emerging availability of hardware-based shadow stacks as part of Intel CET, software-only shadow stacks can be a valuable fallback to provide equal protection on both new and legacy hardware.

We advise against the real-world adoption of return address encryption. It is slower than shadow stacks but has no security advantage. Return address protection might be viable only in rare cases where shadow stacks face compatibility problems or memory shortage.

We have further provided some insights into the impact of the chosen programming language on the performance overhead: Fortran programs face almost no overhead, and C++ programs see more overhead than plain C programs. In particular, we must consider this result when looking at other benchmarks. Often, people evaluate C or C++ only for language-agnostic designs. From these results, we cannot conclude the overhead of a system on other languages because we showed that language is an important factor in performance overhead.

In future work, we suggest an additional analysis that compares the overhead of software-only shadow stacks with hardware-assisted shadow stacks like Intel CET. Further evaluation of the different performance characteristics of ARM-based return address protections, including hardware-assisted protections, can also be of interest. Finally, we are certain that more clever optimizations could be invented that reduce the performance penalty of shadow stacks or return address encryption even further.

Availability

We released the implementations as Open-Source Software, they are avaliable on Github: `https://github.com/MarkusBauer/return-address-protections`

5

NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking

5.1 Motivation

Nowadays, the vast majority of remote code execution attacks target virtual function tables (*vtables*). Attackers hijack vtable pointers to change the control flow of a vulnerable program to their will, resulting in full control over the underlying system. We describe NoVT, a compiler-based defense against vtable hijacking. Instead of *protecting* vtables for virtual dispatch, our solution *replaces* them with `switch-case` constructs that are inherently control-flow safe, thus preserving control flow integrity of C++ virtual dispatch. NoVT extends Clang to perform a class hierarchy analysis on C++ source code. Instead of a vtable, each class gets unique identifier numbers which are used to dispatch the correct method implementation. Thereby, NoVT inherently protects all usages of a vtable, not just virtual dispatch. We evaluate NoVT on common benchmark applications and real-world programs, including Chromium. Despite its strong security guarantees, NoVT *improves* the runtime performance of most programs (mean overhead −0.5%, −3.7% min, 2% max). In addition, protected binaries are slightly smaller than unprotected ones. NoVT works on different CPU architectures and protects complex C++ programs against strong attacks like COOP and ShrinkWrap.

5.2 Problem Description

For decades, attackers have exploited heap errors in C++ programs to mount *code-reuse* attacks. Browsers are a particularly rewarding and valuable target for heap-based code-reuse attacks because browsers contain a massive amount of C++ code and browsers include a scripting engine that allows adaptive attacks. For example, 70% of all vulnerabilities in Google Chrome are memory safety problems [167]. The problem is not specific to browsers, though. In fact, code-reuse attacks are fairly common also in other popular working environments. For instance, use-after-free bugs are the main source of vulnerabilities in Windows [103].

A fundamentally important attack step in exploiting these vulnerabilities is *vtable hijacking*. Attackers commonly use heap corruption vulnerabilities (like *use-after-free* bugs) to corrupt a C++ object. C++ objects contain the address of a *vtable*, which is a read-only list of function pointers pointing to all methods of the given object. Using the memory corruption, an attacker can overwrite the address to an object's vtable with the address of an attacker-controlled memory structure. This memory structure is filled with a faked vtable containing arbitrary function pointers. As soon as the program uses the vtable to retrieve a method pointer, the attacker has full control over the instruction pointer.

Unfortunately, C++ programs contain numerous vtables as compilers use them to implement core language features of C++: class inheritance and virtual methods. The attack surface is significant. Each call of a virtual method relies on a vtable pointer, and some programs use millions of virtual method calls per second. Even worse, researchers showed that vtable hijacking attacks are Turing-complete, even if only existing vtables can be reused (the COOP attack [151]).

Consequently, researchers aimed to defeat vtable hijacking attacks, either with compiler-based program modifications or binary rewriting tools. Previous work [13, 69, 106, 184, 209, 39, 44, 64, 208, 36] protects vtable-based virtual dispatch with security checks added

to the program: before a virtual method is called, either the vtable or the method pointer in the referenced vtable are checked for validity. The set of valid vtables or methods is determined by the C++ class hierarchy and type system. However, these additional checks come with a performance penalty that is potentially non-negligible [69, 172, 44, 64, 208, 36]. Furthermore, recent work [151, 55] introduced new ways to bypass many existing defenses, including [106, 209, 69, 44, 64, 36, 172]. Most proposed solutions only protect vtable-based virtual dispatch, but not other usages of the vtable (e.g., virtual offsets and runtime type information) [13, 39, 175, 106, 184]. Finally, and quite surprisingly, no prior work has attempted to solve the root cause of vtable hijacking: the mere *existence* of vtables.

5.3 Contributions

In this chapter, we radically change the way how C++ member functions are dispatched. Like prior work, we also observe that all possible class types can be determined at compile time [31, 13, 69, 172, 184, 139] if no dynamic linking is required. However, instead of following the well-explored idea of *protecting* vtables accordingly, we show that vtables can be *eliminated*, which tackles the root cause of vtable hijacking. Given the source code of a C++ program, we leverage the class hierarchy to eradicate vtables and to restrict virtual dispatches to the minimal valid set of methods per call site. Technically, we replace vtables with class-unique identifiers (IDs). Instead of vtable pointers, class instances contain an ID that determines their class (dynamic type). Whenever a virtual method is dispatched on an object instance, we load the ID from the instance and build a `switch(ID) case` construct that calls the respective method for a given ID. For each virtual call, we only handle IDs of classes that are possible by the call's static type, effectively preventing function-wise code reuse attacks [151]. While this sounds inefficient at first, we actually *improve* the runtime performance of most tested programs. As a side effect, unlike most related works, we protect *all* constructs that relied on vtables before: virtual offsets, runtime type information, and dynamic casts. Our approach is generic, agnostic to the operating system and system architecture, and is applicable to other compilers and ABIs.

We present a prototype of our protection—NOVT (*No VTables*). NOVT is implemented in Clang/LLVM 10, based on the Itanium ABI. While still a prototype, NOVT can handle complex programs up to million lines of code. We evaluate NOVT on the SPEC CPU 2006 benchmark, Chromium, MariaDB, Clang, CMake, and Tesseract OCR, with a mean runtime overhead of −0.5%, 2% worst case (i.e., speeds up programs on average). Binaries protected by NOVT are slightly smaller than unprotected ones. NOVT's protection level is optimal (as defined in ShrinkWrap [55]), protects against code-reuse attacks like COOP [151], and is applicable to any valid C++ program without code changes (given all linked C++ source code). Our prototype has been released as open-source software[1].

To summarize, our contributions are:

- We show that all C++ semantics can be implemented without relying on vtables.
- We introduce NOVT, an LLVM-based prototype that protects C++ programs by removing vtables. NOVT safeguards even complex programs against vtable hijacking,

[1] `https://github.com/novt-vtable-less-compiler/novt-llvm`

including Chromium (29.7 MLoC), LLVM / Clang (3 MLoC), and the C++ standard library (437 KLoC).

- The level of protection offered by NoVT is optimal and complete. NoVT also protects every use of vtables beyond dynamic calls, including virtual inheritance offsets that are vital for field access and object casts.
- NoVT shows *negative* performance overhead for most tested programs and is thus the first vtable "protection" scheme that does not slow down the majority of programs. At the same time, NoVT also reduces binary size.

5.4 Background

Inheritance in C++ can become quite complex, as C++ supports features like multiple inheritance and virtual bases. We, therefore, start by providing relevant background information on these C++ details. The code in Figure 5.1 will serve as a running example to illustrate how C++ handles classes with inheritance.

5.4.1 C++ Inheritance and Vtables

C++ classes can have *virtual methods* which child classes can overwrite. Calling a virtual method on a class instance invokes the method defined by the actual dynamic type at runtime, regardless of the type in source code (the static type). That means when we call `g()` on a pointer of type `B*`, either `B::g()` or `D::g()` can be executed, depending on the dynamic type of the instance. To dispatch virtual functions, all major C++ compilers use virtual function tables (*vtables*). A vtable is an array of function pointers of all methods of a class and possibly additional information. Each class instance (with virtual methods) contains a pointer to the vtable of its class. When a virtual method is dispatched, the compiler emits code that loads the vtable pointer, fetches the function pointer from the table, and finally, calls this pointer.

5.4.2 C++ Multiple Inheritance

In C++, a class can have multiple base classes (*multiple inheritance*). Figure 5.2 shows the inheritance in our example, where classes B and D inherit from multiple classes. A and NV are base classes, and C inherits only from A. A is always inherited *virtual*, so that D includes one copy of A, B, and C each. Consequently, the four classes A–D form an "inheritance diamond," a common problem in languages with multiple inheritance.

The compiler computes the memory layout of a class according to the Itanium ABI [24]. Derived classes always include their base (parent) classes in memory. That is, the memory representation of a class instance always starts with its primary vtable pointer, followed by all direct base classes in order of inheritance, and last, all fields defined by this class are appended. Figure 5.3 shows the memory layout of our example classes. If an instance of type `D*` is cast to `B*` and method `g()` is dispatched, the generated code would look up that method in `D`'s vtable and call `D::g` with the correct *this* pointer (pointing to the `D` object). However, if an instance of type `D*` is cast to `C*` and method `h2()` is dispatched, we would

```
1 class NV {
2     // 8 bytes, no virtual methods
3     uint64_t nv1;
4 };
5 class A {  // 24 bytes
6     uint64_t a1, a2;
7     virtual void f();
8 };
9 class B : public NV, public virtual A {
10     // 72 bytes (40 + 8(NV) + 24(A))
11     uint64_t b1, b2, b3, b4;
12     virtual void g();
13     virtual void g2();
14 };
15 class C : public virtual A {
16     // 56 bytes (32 + 24(A))
17     uint64_t c1, c2, c3;
18     virtual void h();
19     virtual void h2();
20 };
21 class D : public B, public C {
22     // 120 bytes (24 +48(B) +32(C) +24(A))
23     uint64_t d1, d2;
24     void f() override;
25     void g() override;
26     void h() override;
27 };
```

Figure 5.1: C++ code as running example with multiple (classes B and D) and virtual inheritance (classes B and C)

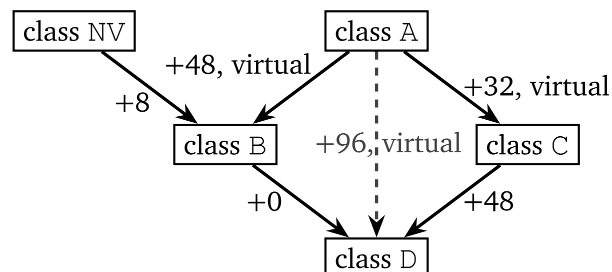


Figure 5.2: Class hierarchy graph for our running example (Figure 5.1). The given offsets relate to the memory layout in Figure 5.3 (e.g., NV starts at offset +8 in B). The dashed line represents *indirect* virtual inheritance.

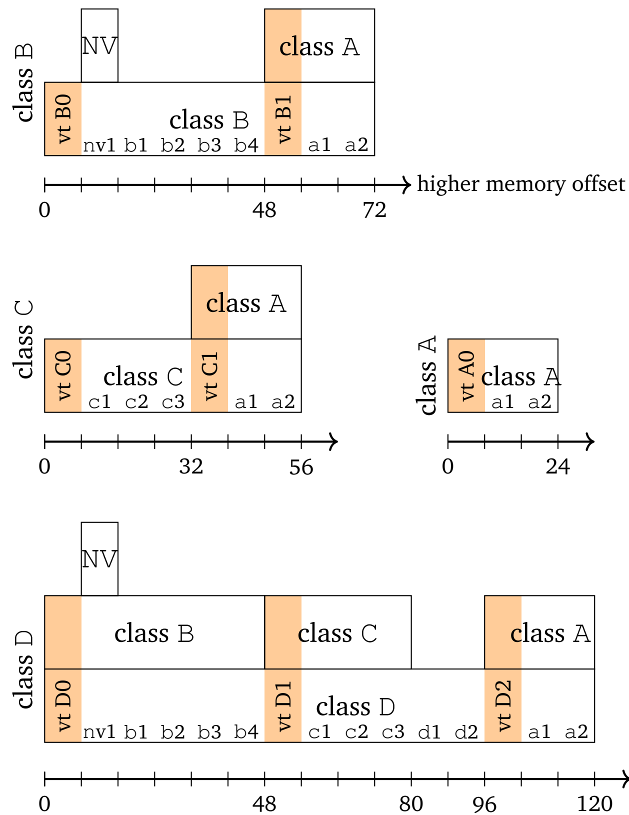


Figure 5.3: Memory layout of our example classes according to the Itanium ABI. Vtables B0 and D0 have a `B`-compatible layout. Vtables A0, B1, C1, and D2 have an `A`-compatible layout. Vtables C0 and D1 have a `C`-compatible layout.

face two problems: First, the vtable layout of `B` and `C` is incompatible because `g2` is stored at index 1 in `B`'s vtable, while index 1 in `C` is `h2()`. That means we cannot dispatch both `g2` and `h2` with the same vtable. Second, *this* would not point to an instance compatible with type `C`—it points to an instance of type `D`, which starts with fields `b1`-`b4`, not with `c1`-`c3` as expected by `C::g2`. To mitigate this problem, Itanium requires *secondary vtables*. The instance of `D` contains a second vtable pointer at offset 48 (the beginning of the `C` structure in `D`). When we cast `D` to `C`, we correct the pointer by this offset so that it now points to the secondary vtable, which gets a `C`-compatible layout, and methods inherited from `C` can be dispatched without additional effort. Methods overwritten by `D` get a special vtable entry that moves *this* back to the beginning of `D` before calling `D`'s implementation.

5.4.3 C++ Virtual Inheritance

With the sketched inheritance model, multiple copies of a C++ base class can be included in a derived one. This is counter-intuitive and not always desired. To solve this, C++ provides *virtual* inheritance. A virtual base class is always included only once, no matter how many base classes inherit from it. In our example, `A` is a base of both `B` and `C`, which in turn are

both bases from `D`, but `D` includes `A` only once. To this end, the memory layout of a class with virtual bases is different: Instances of virtual bases are included at the end of the final class and deduplicated.

This change has some implications. The computed memory layout (Figure 5.3) shows that the memory offsets between a class and its base are no longer constant. When method `B::g2` accesses the field `A::a1` on an instance of `B`, it knows that this field is at offset 56 (*this*+56) because `A` is at offset 48 and `a1` is at offset 8 in `A`. But if the same method is inherited to `D`, `A::a1` is at offset 104 (`A` starts at offset 96 in `D`). The same problem occurs when casting between these types. Again, C++ solves this issue with vtables. In addition to function pointers, a vtable contains the memory offset to all its virtual bases and virtual children (including indirect ones). When a field in a virtual base is accessed or a cast occurs, the compiler first loads the pointer to the object's vtable, then loads the *virtual offset* from the (known) index of the vtable, and finally computes the address of the base or children as *this+virtual offset*. To this end, vtables can also contain a pointer to meta-information about the class, so-called *runtime type information* (*RTTI*), which is used for exceptions and dynamic casts.

To build (and destroy) class instances, compilers may need to use so-called *construction vtables*. These transient tables contain the virtual offsets of a child class but the inherited methods of a parent class. When constructing the child class, they ensure that no method on the child is called before the parent classes are fully initialized. Furthermore, construction vtables guide the objects to use the correct virtual offsets.

5.4.4 Vtable Hijacking

As vtables contain function pointers, they are a valuable target for memory corruption. To hijack control flow, an attacker can modify the vtable pointer stored in the first bytes of an instance, e.g., by exploiting a heap overflow or use-after-free vulnerability. Usually, the attacker overwrites the vtable pointer with an address of an attacker-controlled memory region. Then, the attacker fills this memory region with a pointer to the code they want to execute, creating a fake vtable. When a virtual method is dispatched on the tampered class instance, the program loads a function pointer from attacker-controlled memory and calls it. By tampering with a single vtable pointer, the attacker can thus leverage a potentially small memory corruption to execute attacker-controlled code.

Researchers showed that attacks do not necessarily need fake vtables: "Counterfeit Object-Oriented Programming" (COOP) [151] attacks lead to turing-complete code execution by chaining pointers to existing, valid vtables from a sufficiently large program. Similar to ROP, several gadgets in the form of faked objects with pointers to actual vtables are stored in an attacker-controlled buffer. Later a loop with a virtual dispatch inside is used to dispatch these gadgets one by one. Again, if this loop is in a virtual function, regular vtable hacking (with any vtable containing this virtual function) is used to start the attack. COOP breaks naïve protections that just check if vtable pointers actually point to a vtable because all vtable pointers point to original vtables from other classes.

However, virtual dispatch is not the only security-critical operation on vtables—a fact that several related works dismiss. An attacker can overwrite the vtable pointer with a pointer to attacker-controlled memory and change the virtual offset there. Whenever a

method or field inherited from the base class is used from the manipulated object, the attacker has full control over the *this* pointer of that method or the address of that field, even if the invocation is non-virtual. With this power, common methods like attribute getters and setters can be turned into arbitrary memory read and arbitrary memory write primitives while the control flow of the program stays unaltered. Therefore, we argue that a strong vtable protection must also protect virtual offsets in addition to virtual dispatch.

5.5 Attacker Model

We want to mitigate memory corruption attacks against C++ class instances in which an attacker aims to divert control to an arbitrary function outside of the instance's scope. Hereby, we explicitly include COOP-style attacks [151], i.e., we also aim to prevent the reuse of existing yet arbitrary virtual methods. Furthermore, we aim to protect virtual offsets from arbitrary modifications, as outlined in Section 5.4.4. This attacker model is based on the general model from Section 2.3: we assume that the attacker has arbitrary read and write access to the heap and other memory locations that contain object instances, and they know about the program's memory layout.

We assume that all executable pages are non-writable (`W⊕X`) and that return addresses are protected by other means (e.g., the schemes from Chapter 4). In line with all related works in this context, we ignore C-style function pointers that might even occur in C++ programs; we will consider them separately in Chapter 6.

Summarizing, our threat model reflects common remote code execution attacks such as against Javascript engines in off-the-shelf browsers.

5.6 Design and Implementation

Our protection scheme NoVT removes vtables from a C++ program and replaces them with constant identifiers that are used for dispatching virtual calls. Whenever a vtable was used before, we generate a `switch-case` struct that dispatches the minimal set of possible identifiers, as determined by static types, and aborts execution otherwise. This section outlines our overall methodology. In Sections 5.6.1 and 5.6.2, we explain how we use *class hierarchy analysis* and a *class identifier graph* to determine the set of valid methods for each virtual call. In Sections 5.6.3 and 5.6.4, we show how we create dispatch functions and class identifiers to replace vtables. In Section 5.6.5, we show how we optimize the resulting structure to improve performance. In Sections 5.6.6, 5.6.7 and 5.6.8 we show how we build our prototype NoVT as a fork of Clang 10.

5.6.1 Class Hierarchy Analysis

To replace vtable-based virtual dispatch, we first need to learn the class hierarchies [31] of the program we want to protect. To this end, we add metadata to each class that has at least one virtual method, a virtual base, or inherits a class with virtual methods or virtual bases. Classes without any virtual methods or inheritance cannot be used in virtual dispatch or virtual inheritance and would not contain a vtable pointer anyways (class `NV` in our

example). We ignore them in the remainder of this chapter, as they thus never undermine security according to our threat model. For each other class, we record its virtual and non-virtual base, including the memory offset between the derived class and its base and the defined virtual or overridden methods. We also store a reference to their vtables (which will be removed in a later step) and to all generated construction vtables (including layout class and memory offset from base class to layout class). To avoid name clashes and to support C++ templates, we mangle all class names according to the Itanium ABI [24].

We can construct a class hierarchy graph from the stored inheritance information at link time (see Figure 5.2). Each node in this graph represents a class, each edge represents an inheritance path. Each edge is marked with the memory offset between both classes, i.e., the memory location of the base class inside the derived memory layout. When casting between these classes or dispatching inherited methods, we must correct the pointer (e.g., *this*) for this offset.

5.6.2 Class Identifiers

From the class hierarchy graph, we then determine the necessary class identifiers we need to create. Each class identifier will later replace a primary or secondary vtable, i.e., a class can have multiple class identifiers. Each class identifier is a pair $ID = (cls, o)$ where cls is a class and o is a memory offset. The memory offset denotes that identifier ID will later be written at offset o bytes from the start of the instance. If we read an identifier from an instance pointer and know its offset, we can compute the beginning of the instance (to adjust *this*). Class identifiers with offset 0 signal the beginning of an instance (like a primary vtable), offsets $\neq 0$ have the same purpose as secondary vtables.

For objects in construction, we use combinations of two class identifiers. The first identifier denotes the class under construction and is later used to dispatch virtual methods. The second identifier denotes the class whose memory layout is applied and is later used to resolve virtual offsets. In our example, we need a construction identifier $((\texttt{C}, 0), (\texttt{D}, 48))$ ("`C`-in-`D`") while constructing the `C` instance in `D`. It denotes that the virtual methods from `C` should be used, but if fields from virtual base `A` are accessed, we have to respect `D`'s memory layout (e.g., `A` starts at +48 bytes, not at +32 as in `C`). Construction identifiers avoid that overridden methods of an object are called while the object is not fully constructed; they are similar to *construction vtables* in Itanium.

To this end, we construct a *class identifier graph* (see Figure 5.4 for an example). Each node in this graph is a class identifier, and edges between identifiers show the inheritance between their classes: Two nodes (c_1, o_1) and (c_2, o_2) are connected with an edge $(c_1, o_1) \longrightarrow (c_2, o_2)$ iff i) c_1 is a base class of c_2, ii) casting from a c_2-pointer to a c_1-pointer modifies the pointer by $(o_2 - o_1)$ bytes. In practice, that means: If offset o_2 in a class of type c_2 refers to a field inherited from c_1 at offset o_1, then these identifiers are connected. In our example, $(\texttt{A}, 0)$ and $(\texttt{B}, +48)$ both refer to the beginning of an `A` instance, therefore they are connected. $(\texttt{A}, 0)$ and $(\texttt{D}, +48)$ are not connected because `A` starts in `D` at offset +96.

We generate these class identifiers by traversing the class hierarchy graph top-down, e.g., bases (parents) before derived classes. Created identifiers are marked *virtual* if they have been generated using virtual inheritance, *non-virtual* otherwise. This marking is vanished after all identifiers have been created. For each class c, we create identifiers in these steps:

1. We create one default identifier representing the class at offset 0: $(c, 0)$. We mark this identifier as non-virtual.

2. We traverse all identifiers $id' = (c', o')$ of all base classes c' (given the memory offset o between c and c'):

   - If the identifier is marked non-virtual (i.e., it has not been created using any virtual bases), we create an identifier $(c, o + o')$ and connect it with an edge from id'. This new identifier is marked virtual iff c' is a virtual base of c.

   - If the identifier is marked virtual, we determine its root $(c'', 0)$ in the graph (a unique node generated by rule 1). Next, we determine the virtual offset o'' of c'' in c. Finally, we create a new identifier (c, o'') and connect it with an edge from id'. The new identifier is always marked virtual.

3. For each construction vtable c-in-c' of the class c, we traverse all identifiers $id' = (c, o)$ of this class. For each identifier, we compute the memory offsets o_1 (between $c + o$ and $c + 0$) and o_2 (between $c + o$ and $c' + 0$). We create a construction identifier $((c, o_1), (c', o_2))$ and connect it with an edge from id'.

There is a strong relation between vtables and class identifiers. Every class identifier corresponds to exactly one vtable; connections between identifiers imply that these vtables can be expected in the same location for a given (static) type. We can clearly see this connection when we compare the class identifier graph from Figure 5.4 with the vtables in Figure 5.3: Class `B` has two vtables: one primary vtable and one that has the layout of class `A`'s vtable (e.g., `B` cast to `A`). These two tables correspond to the class identifiers `B.0` and `B.1`. Given a pointer with a static type of `A`, it points to an instance starting with a pointer to the primary vtable of `A` or the secondary vtable of `B`. Therefore, class identifiers `A.0` and `B.1` are connected. Similar for `D`: class `D` has three vtables that correspond to the class identifiers `D.0`, `D.1` (`D` casted to `C`), and `D.2` (`D` casted to `A`). Similarly, all construction identifiers correspond to a construction vtable. We will use this connection later and replace vtable pointers with a unique number per class identifier. The edges between class identifiers help us to enumerate all possible class identifiers that can occur for a given static type.

5.6.3 Dispatch Function Generation

While a vtable-based dynamic dispatch can be compiled independently of other compile units, this is no longer possible for our protection (and, in general, for any protection that relies on class hierarchy analysis). To dispatch a function, any approach based on a class hierarchy analysis needs to know all possible functions that can be called, which might not even be declared in the current unit. To counter this and allow incremental compilation, we compile dynamic dispatch in two steps. When compilation requires a dynamic dispatch, we declare a *dispatch function* instead and replace the dynamic dispatch with a (static) call to this dispatch function. The generated dispatch function has exactly the same type as the virtual function we want to call. It is annotated with the static type of the object we dispatch on and the method name we need to dispatch. To account for templates and overloaded functions, we mangle this method name according to the Itanium ABI [24].

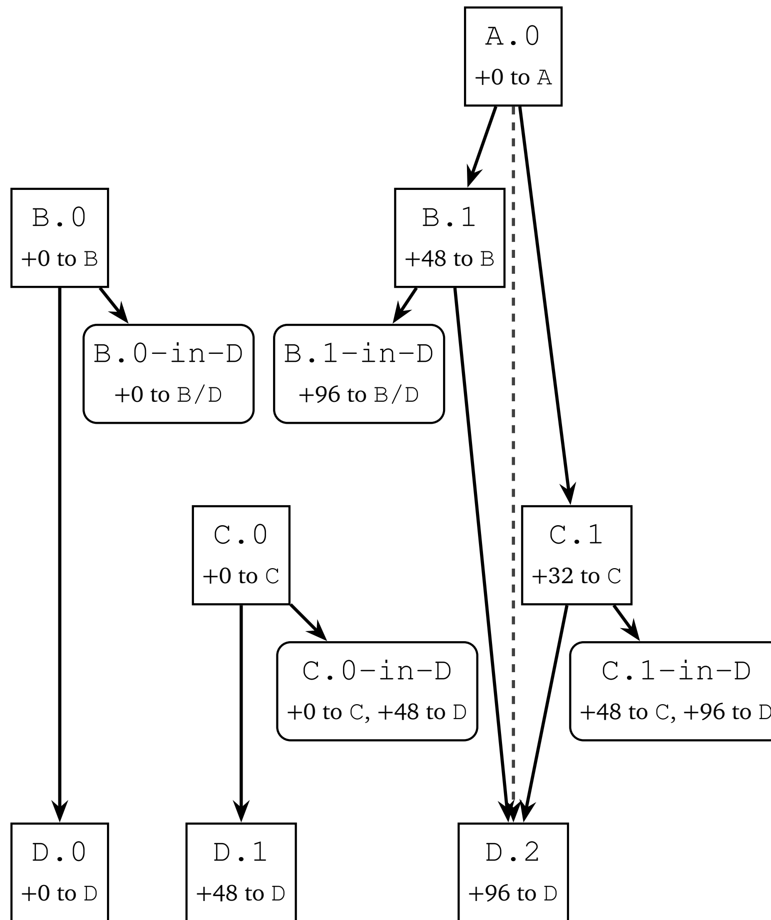


Figure 5.4: The class identifier graph of our example, including all possible construction identifiers (rounded rectangles).

We define these methods later in the linking phase. For each method, we identify the `this` argument and start with loading the class identifier using that pointer. We then create a `switch-case` structure (with LLVM's `SwitchInst`). A case in that switch instruction handles each possible class identifier. We can get all possible class identifiers by traversing the class identifier graph, starting with the static type's primary class identifier. For example, when dispatching on class `C` we would start with identifier `C.0` and traverse `D.1` and `C.0-in-D`. For each case, we can determine the method to be called by the class identifier's (dynamic) type using standard inheritance rules. We emit a (static) call to this method and return its result. To catch potential memory corruptions, we emit an LLVM Trap intrinsic (x86's `ud2`) in the default case of the switch. The program crashes to avoid control flow hijacks if an invalid class identifier occurs at runtime. The order of the traversal here is not important. LLVM will reorder all emitted cases during assembly generation depending on the assigned identifier number. Also, in the rare cases where LLVM does not reorder checks, the order has a negligible effect on performance. We use the same method to implement other constructs that used to rely on vtables: Whenever the compiler needs to know the virtual

offset of a virtual base, we emit a call to an *offset function* (typed pointer → word-sized int). This function is later defined by a `SwitchInst` that simply returns the correct offset for all possible class identifiers. Dynamic casts that cannot be resolved at compile-time are replaced with a *cast function* (typed pointer → pointer). For each possible class identifier, this function returns either the pointer (potentially with an offset) or `nullptr` if the class cannot be cast. Last, we replace every access to runtime type information with a call to a *rtti function* that returns a pointer to the correct RTTI structure when called.

5.6.4 Storing Class IDs and Removing Vtables

The mechanisms described until now already replace all C++ concepts that normally rely on vtables. We finally can delete the vtables themselves and replace the instance's pointers to them with the class identifier that corresponds to a given vtable. To this end, we assign a unique number to every class identifier in the class identifier tree. Next, we find direct usages of the vtable, which happens only in class constructors (when the vtable's pointer is written to the instance). We replace that vtable pointer with the corresponding class identifier's number and write it into the memory slot that used to store the vtable pointer. This modification does not change the memory layout of classes, i.e., we do not introduce memory overhead with this step. All dispatcher functions (and friends) load this identifier number later from the vtable pointer slot and use it as described. Finally, having removed all references to the vtables, they will be removed by a following *Dead Globals Elimination* pass.

5.6.5 Optimizations

We extend this basic methodology with optimizations that boost the runtime efficiency of our protection.

5.6.5.1 Dead Class Identifiers

Class identifiers that are never used in a generated function or class identifiers whose identifier number is never written in a constructor can be safely removed. Removing them saves some branches in the generated functions as well as it allows for a more dense numbering. We evaluate optimizations on the C++ programs from SPEC CPU 2006 and Chromium. Applied after all other optimizations from this section, we can, on average, remove 29% of all class identifiers because they are never used in a constructor and additional 22% because they are never used in a generated function.

5.6.5.2 Merge Cases

In the generated functions, many cases can be merged, for example, when multiple classes inherit the same method without overriding it. Whenever cases execute the same action (the same method dispatched, return same virtual offset, etc.), we merge them into one case. This reduces code size and improves later optimization results. In our experiments, we could remove 24% of all case handlers on average.

5.6.5.3 Devirtualization

Whenever a generated function has only one possible case (possibly after combining cases, excluding the error case), we can statically determine the only legal path and remove the switch-case around it. Only a single static call or single constant offset remains; the virtual operation has been statically resolved (devirtualized). We mark the generated function for inlining to eliminate any performance impact. On average, 57% of all generated functions in our experiments can be devirtualized.

5.6.5.4 Merge Identifiers

We look for pairs of class identifiers that trigger the same behavior in all generated functions and merge them into a single identifier, thus reducing the total number of identifiers. These "equal" identifiers occur, for example, when a subclass does not overwrite any methods or when a construction identifier is rarely used. We search these equal identifiers by iterating all generated functions: whenever a pair of two identifiers are handled in the same function (i.e., occurs in the same `SwitchInst`) and the two identifiers trigger the same behavior, we mark them as "potentially equal." We mark them as "not equal" if they trigger different behavior and must not be merged. In a second step, we check all pairs: If two identifiers are marked "potentially equal" but no "not equal" marking has been set, they can be safely merged without changing program semantics. This holds because the only (legal) usage of class identifiers are the generated functions, and there is no case where switching between the two would change the behavior of a generated function. The "potentially equal" and "not equal" relations are propagated to the merged identifiers; hence more than two identifiers will be merged if possible. With this optimization, we can merge 10% of all identifiers on average in our experiments.

5.6.5.5 Optimizing Identifier Numbers

The performance of our protection is dominated by the efficiency of the compiled switch instruction. The efficiency of the generated code depends mainly on the number of cases, the density of the numbers, and the maximal size of the numbers. Dense packs of numbers can be handled by more efficient constructions (e.g., jumptables) than sparse sets. We thus set out to assign consecutive numbers to identifiers that are used in the same dispatch function(s).

To this end, we first group identifiers that are used in the same generated dispatch function and trigger different behaviors there. We then assign numbers to each group independently. All identifiers in a group need a different identifier number to be distinguishable. This is not the case for identifiers in different groups. Such "colliding" identifiers are never used together and therefore do not have to be distinguishable. Consequently, each disjoint group will have its own independent numbering starting with 0. Hence, unrelated identifiers from different groups can have the same number. Assigning a number multiple times does neither harm correctness nor security. Whenever such a duplicate number occurs in the program, it is absolutely clear from the context to which of the identifiers it belongs.

We use a non-optimal recursive algorithm (shown in Algorithm 1) to assign numbers to a group of identifiers. For each set of identifiers larger than some threshold, the algorithm

splits it into two subsets that will receive consecutive numbers recursively. One of the subsets is the maximal subset used in a generation function. We assume a hierarchical structure of used identifier sets in our generated functions, and this algorithm tries to follow that hierarchy. Our experiments show that this non-optimal algorithm significantly (50% on average) compresses the identifier space.

```
function CreateIdentifierNumbers:
    input  : identifiers // identifier subgraph set
    input  : next_number // initial 0
    output : next_number // next free number

    if ||identifiers|| > 5 then // small sets need no advanced ordering
        biggest_subset := ∅;
        /* find the biggest subset from all generated functions, ignoring small functions */
        foreach func in generated_functions do
            if func.used_ids ⊊ identifiers and 4 ≤ ||func.used_ids|| and
              ||func.used_ids|| > ||biggest_subset|| then
                biggest_subset := func.used_ids;
            end
        end
        if biggest_subset ≠ ∅ then
            /* identifiers used together should get connected numbers */
            next_number := CreateIdentifierNumbers (biggest_subset,
              next_number);
            next_number := CreateIdentifierNumbers (identifiers −
              biggest_subset, next_number);
            return next_number
        end
    end
    /* Order small sets and sets that cannot be split. The order of identifiers is close to prefix
       traversal of the identifier tree. */
    foreach id in identifiers do
        id.number := next_number++;
    end
    return next_number
end
```

Algorithm 1: The algorithm used to assign numbers to class identifiers.

5.6.6 Implementation

We have implemented the NoVT prototype on top of Clang 10 and LLD 10. NoVT modifies C++ programs during compilation and linking. In the following, we will detail our implementation choices for both phases.

In the compilation phase, our modified compiler adds information about classes, inheritance, and virtual methods as metadata to its output. We replace virtual dispatches

with a call to a *dispatch function*. Similarly, we replace virtual offset loads with a call to an *offset function*. The same holds for `dynamic_cast` (*cast functions*) and RTTI loads (*rtti functions*). We enforce link-time optimization (LTO) in our compiler, so the output is always LLVM bitcode ready for further analysis. Vtables are still emitted; they will be removed later.

In the linking phase, we generate the newly introduced functions in a compiler pass introduced to `lld` or LLVM's `gold` plugin. First, we combine the metadata from all compilation units and reconstruct the full class hierarchy of the program. From this class hierarchy, we build an *identifier graph*, a structure that is similar to vtables and their inheritance relations. Using this graph, we assign IDs to all classes and change their constructors to write that ID to each class instance, replacing the vtable pointer. For each dispatch function, we determine the set of possible call targets from the identifier graph and insert a `switch` statement that can call exactly these candidates. Similarly, we generate all necessary offset, cast, and rtti functions. We optimize our identifier graph and the generated functions to counter possible performance overhead caused by the protection. Finally, the compiler's code generation applies off-the-shelf compiler optimizations.

5.6.7 Compiler-Assisted Optimizations

LLVM runs its optimizations that interact with our generated code and further improve the performance. First of all, LLVM's *Dead Global Elimination* pass eliminates all (now unused) vtables. Additionally, it identifies now unused methods and RTTI structures and eliminates them, which was not possible before our transformation. We evaluate the positive effect of this program size reduction in Section 5.7.4.

Next, LLVM can decide to inline some or all of our generated functions, especially if they are short or only called from a few locations. On the other side, short methods might be inlined into our generated function. Both inlining operations save us a `call` instruction. In the best case, our virtual dispatch is compiled without a single assembly `call`. Inlining gets especially performant if LLVM can infer the result of the identifier number `Load` instruction using constant propagation (for example, if a constructor has been inlined before). LLVM has a pass that allows even interprocedural constant propagation, on our transformed program this optimization is able to devirtualize some further callsides. Finally, LLVM uses *tail calls*: when we call a regular method from a dispatcher function, LLVM emits a `jmp` instead of a `call`. As the dispatcher function and method have the same signature, the method can return instead of our dispatcher function later. Tail call optimization saves us a second `call`, a second `ret` and the space of a return address on the stack.

Last but not least, LLVM has several tweaks to efficiently implement a `SwitchInst` in assembly. The most well-known trick is a *jumptable*: Given numbers close to each other, LLVM uses that number to load the address of the next instruction and jump to it. Jumptables are similarly efficient than vtables (a `cmp` instruction more), but they are memory-safe because the jumptable index is bounds-checked before usage. Jumptables are scalable, and their performance does not depend on the number of possible cases. Alternatively, LLVM can translate functions with only a few cases to a chain-like or tree-like structure of compares and jumps. These structures do not have the overhead of an additional memory access. Given a small number of possible cases, they are usually faster than jumptables (or regular vtable-based dispatch). Another trick is to use bitstring tests to check for many cases at once.

Given many numbers that fall to a single case (e.g., many classes inheriting a non-overridden method), LLVM uses the bittest instruction (`bt`) to select a bit from a pre-computed 64-bit word. If the selected bit is 1 the case is correct.

5.6.8 Usability

For convenience, we modified Clang to use LTO by default, to use our modified `lld` linker by default, and to include an additional library search path containing a pre-compiled protected `libstdc++`. With these small changes, our modified Clang is a drop-in replacement for typical compilers (real Clang, mainly compatible with g++). For most build systems, we only need to set an environment variable to get protected binaries (`CXX=novt-clang++`).

5.7 Evaluation

We now evaluate NoVT to see its impact on the programs to protect. Most importantly, we aim to assess the provided security level and the performance overhead. Next, we also want to know the impact on the binary size and the limitations of this protection.

For our evaluation, we use all C++ programs from the standardized SPEC CPU 2006 benchmark [58, 161], namely *astar*, *dealII*, *namd*, *omnetpp*, *povray*, *soplex*, and *xalancbmk*. Using that benchmark allows us to compare our results against other solutions (using the same benchmark). *namd* is unique because it does not use virtual inheritance nor virtual dispatch. We still include it in our evaluation to show the absence of any impact of our approach if no protection is necessary. To demonstrate scalability and practicability, we also evaluate NoVT on Chromium 83 [166] (Google Chrome), which consists of 29.7 million lines of C/C++ code, many dependencies, and is highly relevant in practice. Additionally, Chromium does not tolerate any slight error in the C++ language semantics and therefore is a good test case to show that our approach does not break programs. To show a broad compatibility with a variety of popular C++ projects, we evaluate NoVT on MariaDB (a SQL database server with 2.4 MLoC), CMake (340 kLoC), Clang/LLVM 9 (2.9 MLoC) and Tesseract OCR (300 kLoC).

All programs have been compiled with full protection to get comparable results, including a protected C++ standard library and full link-time optimization enabled. Reference is always the same program compiled with unmodified Clang 10, full LTO, and a statically linked LTO-ready version of the C++ standard library. For SPEC CPU 2006 we use all available optimizations (`-O3`); for Chromium we enable all provided optimization options in its build system. Other programs use their respective defaults for a release build.

5.7.1 Security Evaluation

NoVT protects any vtable-related operation against a memory-modifying attacker. This includes virtual memory offsets, dynamic casts, and rtti access. We also protect member function pointers. Manual inspection of the generated dispatchers shows that even concurrent memory modifications cannot give an attacker more control about the instruction pointer, no intermediate values are stored on the stack. Our protection restricts the callable methods at each virtual dispatch location to the minimal set that would still be allowed by

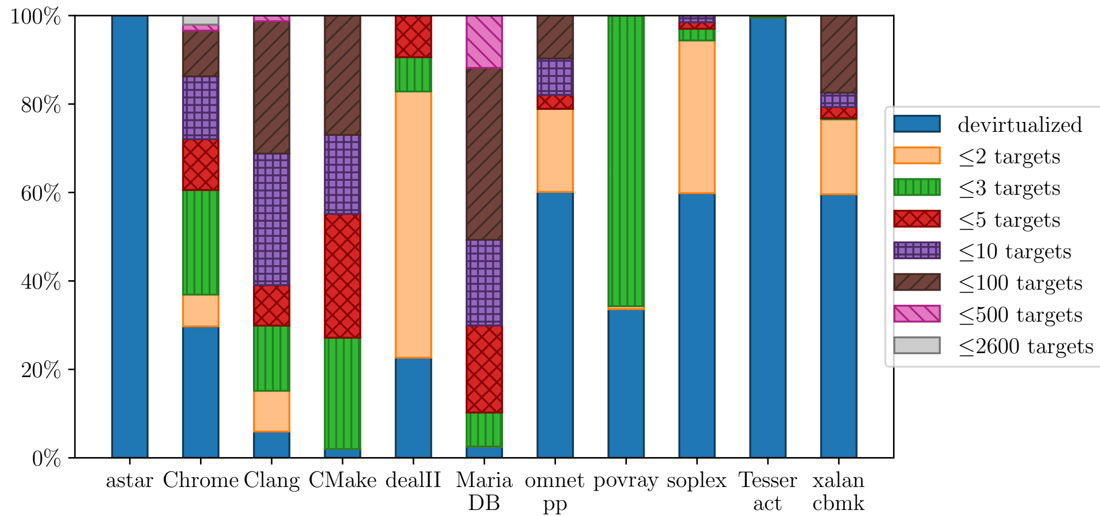


Figure 5.5: Virtual dispatchers broken down by their number of possible call targets.

the type system, depending on optimizations even less. Haller et al. [55] showed that this restriction is *optimal* without (potentially expensive) context analysis.

This implies of course that an attacker can trigger other, possibly unintended methods by overwriting the type ID in memory, as long as these methods are still allowed by the type system. However, modifying the stored identifier number does not give full control over the instruction pointer, and the number of callable functions is considered low enough to prevent code-reuse attacks (including COOP attacks) [151]. In the worst case, this very limited set of methods might contain security-critical functionality: For example, assume classes `AdminUser` and `RegularUser` which both inherit from `User`. For a regular user object typed `User*` in source code, an attacker could overwrite the stored ID with the ID of `AdminUser`. This attack gives access to methods overridden by `AdminUser` that were unreachable before, but does not give access to any additional methods added by `AdminUser`, nor can the attacker trigger different methods with potentially incompatible signature or methods unrelated to `User`, as he could with regular vtables. The impact is quite program-specific, as an attack requires specific call patterns in code. This restriction applies for all solutions that restrict virtual dispatch based on a class hierarchy analysis, including [13, 69, 106, 184, 28, 172, 209]. Our level of protection is equal to or better than these solutions.

In order to assess our intuition that NoVT prevents COOP-style attacks, we have to inspect how many valid targets each dispatcher allows to use. Successful COOP attacks [151] require a large set of "vfgadgets"—virtual functions that can be called by manipulating the object. Schuster et al. identified ten different types of vfgadgets that must be available for turing completeness (e.g., virtual functions reading memory, performing arithmetic operations, etc.), some of them more than once (e.g., reading memory to different registers). In unprotected programs, any virtual function (referenced in a vtable) is a possible target of any virtual dispatch in a COOP attack. With NoVT applied, all vfgadgets must be different implementations of the same method inherited from a single base class, with the exception

of the “main loop” vfgadget, which in turn must dispatch that method on that base class. From Figure 5.5, we see that in a protected program, most virtual dispatches have less than 10 possible targets, particularly for SPEC programs. The majority of these target functions likely do not qualify as vfgadgets because of their size and complexity. Even if, they are likely vfgadgets of the same type, because all target functions share the (high-level) semantics of the parent method.

Chromium represents a good worst-case example for vtable protection: its large codebase contain 39,100 virtual class definitions, with more than 54,700 vtables containing over 210,000 distinct virtual functions. NoVT restricts virtual dispatches to 2.9 possible targets on average. Excluding devirtualized calls, 5.7 possible targets per virtual dispatcher remain. This low number of valid targets does not allow to spawn successful COOP attack, i.e., the vast majority of virtual dispatches is COOP-safe. When looking at the maximum number of targets, we see that a small fraction of virtual dispatches have a higher number of targets (up to 2600). These dispatchers are required by some very generic interfaces in Chromium’s source code that are implemented by many classes: `mojo::MessageReceiver`, `blink::ScriptWrappable`, `blink::GarbageCollectedMixin` and `google::protobuf::MessageLite`. However, we believe that these generic interfaces do not pose an additional risk for COOP-style attacks, because they all share similar function types and purposes. It is unlikely that all required types of vfgadgets are present in a set of almost-identical functions. For example, for any protobuf-inheriting object, the function `Clear` will always write memory but never be usable as a memory-reading vfgadget. Restricting these dispatches further is not possible without breaking the legitimate use of these interfaces. This assessment is further supported by the COOP authors who state that a C++ class-hierarchy-aware restriction of virtual call targets reliably prevent COOP attacks even for large C++ target applications (“COOP’s control flow can be reliably prevented when precise C++ semantics are considered from source code” [151], Section VII B).

NoVT applies the same level of protection to virtual offsets. Whenever a virtual offset is retrieved from an object, we restrict the possible results to the minimal set possible in the type system. Without this protection, an attacker could hijack vtables to arbitrarily change the *this* pointer of some inherited methods. Attackers can leverage the *this* pointer to get arbitrary memory read/write primitives from a single corrupted vtable pointer. Only few related work protects virtual offsets [69, 209], and none of them restricts virtual offset to the minimal possible set.

NoVT actually found an invalid virtual dispatch in one of the SPEC CPU benchmarks (*xalancbmk*, in `SchemaValidator::preContentValidation`). This benchmark does an invalid sidecast of an object instance, then calls a virtual method on it and finally checks if the cast was valid at all. This error would remain undetected with traditional vtables, because both intended and actual vtables have the same layout, but NoVT detects that this dispatch violates the type system and halts the program. We are not the first to report this issue [13, 172] and provide a patch file in our source release.

The protection strength could be improved further by randomization: In contrast to vtables, type IDs are not addresses and could be randomized independently of ASLR. A simple randomization could use a constant, random offset added to all type IDs, preserving the structure of the dispatchers including their speed. To break this randomization, an attacker has to leak one type ID from memory. A stronger randomization could assign all

type IDs completely random. This approach will be less efficient, because dispatchers with jumptables are not possible anymore, but an attacker would have to leak each type ID from an existing class instance before overwriting any stored ID.

5.7.2 Runtime Evaluation

The performance overhead of any protection may hinder its real-world adoption. In this section, we therefore provide thorough experiments that confirm that NoVT has a negligible (actually, *negative*) run time slowdown.

5.7.2.1 Benchmark Selection

We evaluate the performance overhead of NoVT using the SPEC CPU 2006 benchmark suite, parts of Chromium's benchmark suite, six well-known browser benchmarks, and standard benchmarks of MariaDB, Clang, and Tesseract OCR.

For SPEC, we run the full benchmark suite, excluding programs without C++. SPEC runs every program three times and takes the median runtime as a result. We run each experiment 20 times to get a meaningful result without outliers.

For Chromium, we use all standardized benchmarks contained in Chromium's benchmark set [169], which are *octane 2* [132], *kraken 1.1* [115], *jetstream2* [197], *Dromaeo DOM* [114] (not to be confused with the Dromaeo Javascript benchmark) and *speedometer2* [198], which is a real-world benchmark comparing the performance of the most popular web frameworks. We add *sunspider 1.0.2* [199] because it has been used to evaluate many related work [209, 208, 44, 69, 13, 184, 210, 39]. Other relevant literature used additional benchmarks to assess performance impact on HTML/rendering performance, but these historical samples are not included anymore in modern Chromium (which switched from Webkit to blink). Instead, we benchmark the blink HTML engine's performance with the quite extensive blink benchmark set (326 samples in total). To avoid any bias, we selected all benchmark sets that i) contained more than just a few samples, ii) worked flawlessly on an unmodified reference Chromium, and iii) whose results had a reasonably low standard deviation (below 1%). These were the blink benchmark suites *css*, *display_locking*, *dom*, *events*, *layout*, *paint* and *parser*. We take the geometric mean of all these results as Chromium's performance overhead. Again we repeated all experiments 20 times. For MariaDB, we use the DBT3 benchmark provided by the developers with the InnoDB database engine. For Clang, we measure the time it takes to compile and optimize SQLite. For Tesseract OCR, we use the benchmark from the Phoronix test suite [80]. We do not conduct runtime analysis on CMake, because its runtime is mainly determined by the speed of the programs it invokes (e.g. compilers), and there exists no benchmark targeting CMake.

5.7.2.2 Benchmark Environment

We ran all SPEC benchmarks on an Intel Core i5-4690 CPU (4×3.5 GHz, no hyperthreading) with 32 GB of RAM. The operating system was Debian 10 "Buster" with kernel 4.19. We used the "performance" CPU governor, disabled CPU boost, and applied `cpuset` to minimize the impact of the environment and operating system on the measurements. The standard deviation of all benchmarks was at most 0.7%, and 0.32% on average. We ran all Chromium

benchmarks on an Intel Core i7-6700k CPU (4×4.0 GHz with hyperthreading) with 16 GB of RAM and an AMD Radeon RX480 graphics card. Operating system was Ubuntu 18.04 with kernel 5.3. The standard deviation of all Chromium benchmarks was always below 0.92%, and 0.37% on average.

5.7.2.3 Performance Overhead

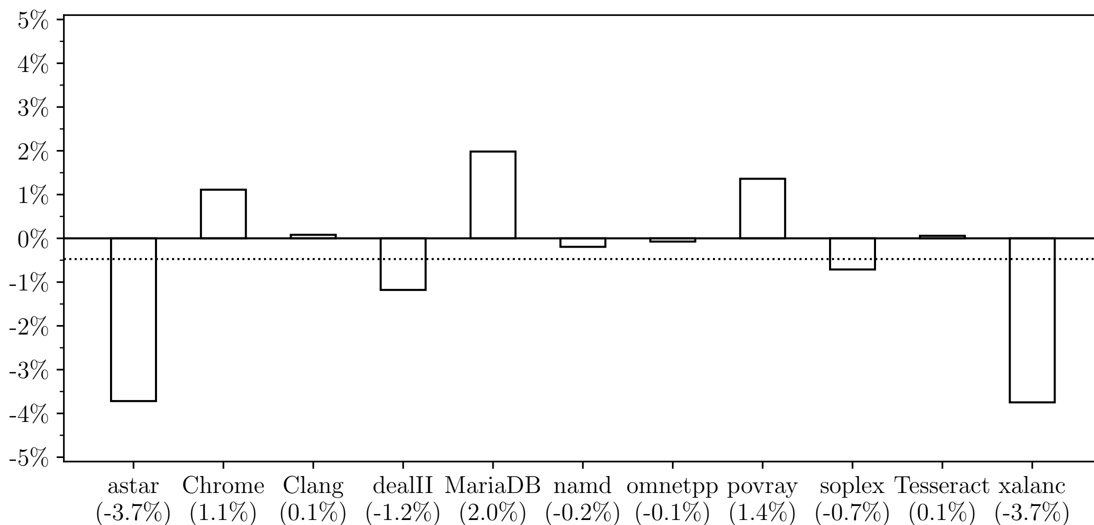


Figure 5.6: Performance overhead of NoVT on all programs.

Figure 5.6 and Figure 5.7 show the performance overhead on the different programs and benchmarks. We can see that many programs actually get faster after protection (astar, deadII, soplex, xalancbmk), while few get slightly slower (Chromium, povray, MariaDB). The average overhead is −0.5% and thus negative. The average overhead on the set of programs commonly used in related literature (SPEC CPU and Chromium) is −0.9%. That is, our benchmarks get faster on average, with a worst-case overhead of 1.98% on MariaDB. The best result is *astar*. This program has been completely devirtualized by NoVT and improves its performance by 3.7%. The highest overhead in a SPEC benchmark occurs on *povray*. Manual investigation shows that povray has only 28 classes, excluding the standard library, and 1500 virtual calls per second—xalancbmk has 63 million virtual calls per second. Disabling the protection on parts of the class hierarchy reveals that its overhead only loosely correlates to the number and structure of the generated dispatch functions. We believe that this overhead instead comes from subtle changes in the program's code layout. Evaluating hardware performance counters on synthetic microbenchmarks did unfortunately not give conclusive insights: NoVT's overhead does not correlate with the cache miss rate. However, protected programs seem to have a lower branch misprediction rate.

We summarize the size and call frequency of the generated functions in Figure 5.8 and the number of virtual calls, virtual offset accesses and dynamic casts in Table 5.1. We can see that NoVT can handle billions of dispatches while still having a negative overhead in some cases. The same holds for virtual offsets. NoVT does not necessarily impose an overhead

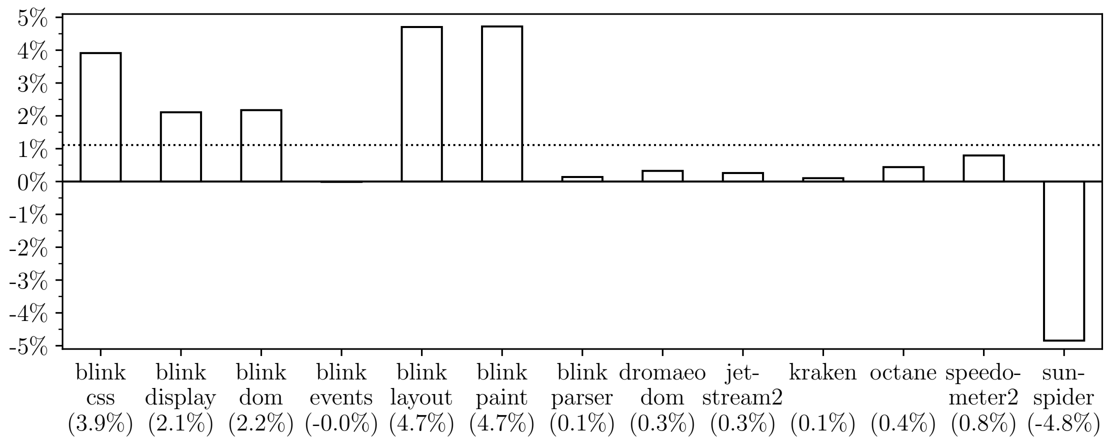


Figure 5.7: Performance overhead of NoVT on Chromium.

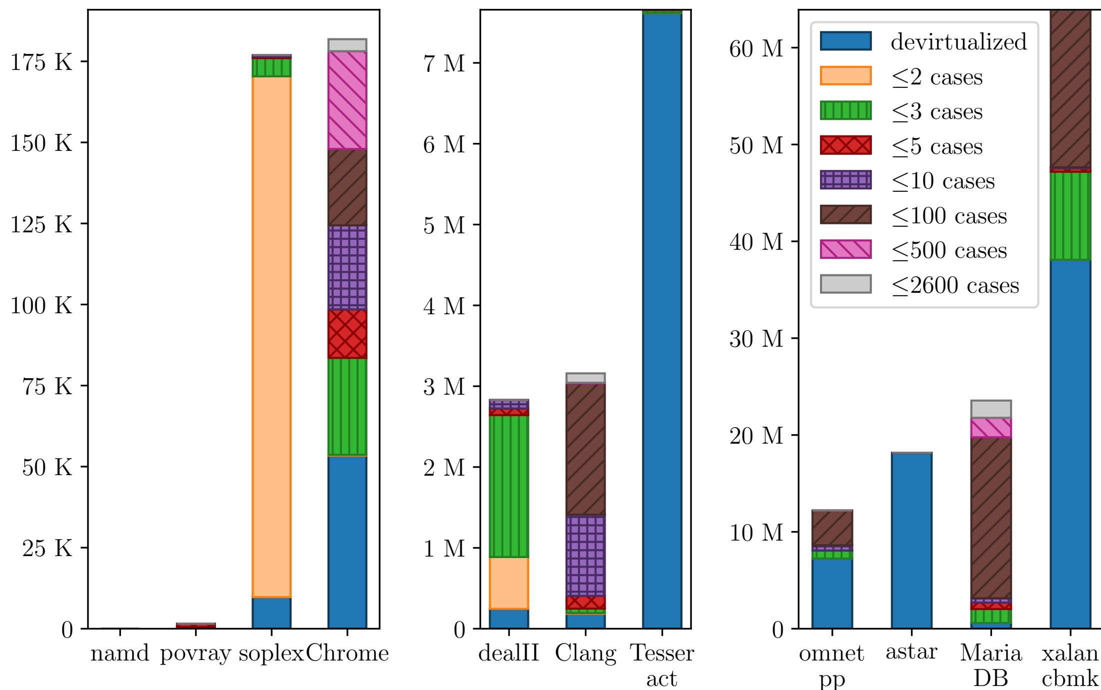


Figure 5.8: Virtual actions (calls, vbase offset, etc) per second, broken down by the number of `switch` cases in the NoVT dispatcher functions.

when virtual inheritance is used extensively. However, the actual performance overhead is not only subject to the class structure. Again, we rather speculate that performance is due to the code layout, which is out of our control.

For Chromium (Figure 5.7), we can easily see that real-world benchmarks tend to have a much lower performance overhead than synthetic HTML engine benchmarks. The worst case overhead is 4.7%, while the mean overhead is only 1.1%. We expect the real-world

Table 5.1: Number of protected operations per benchmark.

Benchmark	# virtual call	# virtual offset	# dynamic_cast	# rtti access	runtime
Chromium: blink css	1,123,380,274	364,138	0	0	251 sec
Chromium: blink display_locking	24,693,016	168,377	0	0	49 sec
Chromium: blink dom	1,542,188,247	630,578	0	0	80 sec
Chromium: blink events	8,315,599	70,093	0	0	94 sec
Chromium: blink layout	35,869,225,028	1,755,609	0	0	1150 sec
Chromium: blink paint	5,607,395,213	859,399	0	0	385 sec
Chromium: blink parser	4,331,060,409	523,289	0	0	453 sec
Chromium: dromaeo	1,947,841,146	30,748	0	1	149 sec
Chromium: jetstream2	804,807,604	316,224	2	5	180 sec
Chromium: kraken	59,387,742	59,417	0	1	34 sec
Chromium: octane	118,479,849	24,373	0	1	50 sec
Chromium: speedometer2	541,254,241	194,405	0	0	74 sec
Chromium: sunspider	5,766,321	18,361	0	1	18 sec
Clang	97,010,210	961,927	493,848	0	31 sec
MariaDB	15,547,629,306	59,940	43,740	0	263 sec
Tesseract	292,155,762	192,129	16,793	0	38 sec
astar	4,996,986,681	0	0	0	286 sec
dealII	201,873,776	98,486,795	225,926,036	0	188 sec
namd	0	0	0	0	292 sec
omnetpp	3,361,136,271	14	47,429,169	330	279 sec
povray	153,212	0	0	0	98 sec
soplex	3,259,515	32,123,619	161	0	202 sec
xalancbmk	9,867,616,106	612,327	48	0	160 sec

overhead to be close to the latter—in particular, *speedometer2* (+0.8%) is a good candidate to measure this because it tests the performance of widely used web frameworks. *sunspider* (−4.8%) shows that even some parts of Chromium actually got faster. While this result looked suspicious at first glance, we repeated this experiment twice to exclude any error on our side, but we can reproduce this behavior with a reasonably low standard deviation (0.5% max). Further investigation revealed that sunspider is not the only case where the protected Chromium is faster than the reference (like the "blink image_decoder" benchmark, roughly −2%). However, all other cases did not comply with our rules as described in Section 5.7.2.1, and we hence did not include them because they were not sufficiently representative.

Chromium seems to use less virtual dispatch than some of the SPEC benchmarks. We observed the most intensive use in the benchmark "blink display_locking" with 510,000 virtual calls per second. On the other hand, Dromaeo DOM used only 20,000 virtual calls per second. The size of the generated functions was roughly proportional in all benchmarks; it is summarized in Figure 5.8. A direct connection between size and number of the called virtual functions could not be observed.

Summarizing, we can say that NOVT protects most programs without any performance penalties, and only some programs experience a slight yet negligible slowdown from the protection. Even complex programs like Chromium with large generated functions do not necessarily suffer from performance drain. Finally, with a focus on Chromium, we can say

Table 5.2: Summary of assembly constructs used to build virtual function dispatchers.

Binary	Compare chains	Trees	Jumptable	Partial Jumptable	Other/ukn	Total
Chromium	39077 (82.71%)	3289 (6.96%)	4331 (9.17%)	114 (0.24%)	435 (0.92%)	47246
Clang	2307 (49.63%)	1094 (23.54%)	489 (10.52%)	528 (11.36%)	230 (4.95%)	4648
MariaDB	1107 (45.86%)	537 (22.25%)	382 (15.82%)	281 (11.64%)	107 (4.43%)	2414
CMake	259 (53.96%)	99 (20.62%)	55 (11.46%)	39 (8.12%)	28 (5.83%)	480
Tesseract	215 (78.75%)	10 (3.66%)	14 (5.13%)	1 (0.37%)	33 (12.09%)	273
astar	1 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1
dealII	72 (79.12%)	2 (2.20%)	11 (12.09%)	0 (0.00%)	6 (6.59%)	91
omnetpp	95 (59.38%)	12 (7.50%)	43 (26.88%)	0 (0.00%)	10 (6.25%)	160
povray	33 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	33
soplex	98 (73.68%)	0 (0.00%)	33 (24.81%)	0 (0.00%)	2 (1.50%)	133
xalancbmk	1320 (81.89%)	136 (8.44%)	103 (6.39%)	37 (2.30%)	16 (0.99%)	1612

that the performance bias we introduced with our protection was more than compensated by the performance gain of enabling full link time optimization.

5.7.2.4 Compilation Time

The compilation time of our solution itself is unobtrusive. Our passes take a few milliseconds up to a few seconds to run on a SPEC program, and less than 40 seconds on Chromium. This compilation time is negligible in a build process of multiple minutes (SPEC) or multiple hours (Chromium), so we did not optimize the compile time in our prototype. Admittedly, building with Clang's link time optimization is slower than a regular build.

5.7.3 Generated Code Evaluation

NoVT replaces vtables with switch-case constructs in dispatcher functions, which the compiler then further optimizes during assembly generation. Table 5.2 shows that dispatchers generally follow one of four low-level code structures. Most dispatchers consist of a linear *chain* of `cmp` or `bt` checks on the type ID, followed by a conditional jump to the target function. These dispatchers usually check for only a small number (at most 4) of type IDs. They are considered the fastest because they do not access any memory. Dispatchers handling a large number of type IDs often utilize a *jumptable*, guarded by an initial range check. Jumptables require a memory read access but scale well in the presence of many type IDs. If a jumptable is unsuitable, LLVM generates a *tree*-like comparison structure that includes `cmp`-based range checks and bitset tests. In particular, trees are used if the number of type IDs is low, if the type IDs are sparsely distributed, or if many type IDs default to the same inherited method. While trees are slower than short compare chains, they also operate without memory access and are better suited if the type IDs are not distributed dense enough for a jumptable. The maximum tree depth we saw was five. Finally, multiple jumptables can be combined by a tree (*partial jumptable*). Our distribution algorithm tries to distribute type IDs as densely as possible, to avoid these nested structures.

Table 5.3: Size of binaries before and after protection.

	Unprotected binary size	Protected binary size	Binary size overhead
Chromium	211,025 KB	197,959 KB	−6.19%
Clang	125,914 KB	125,779 KB	−0.11%
CMake	11,149 KB	10,629 KB	−4.66%
MariaDB	21,643 KB	24,779 KB	14.49%
Tesseract	3,647 KB	3,162 KB	−13.30%
astar	149 KB	143 KB	−3.96%
dealII	1,322 KB	817 KB	−38.13%
namd	379 KB	371 KB	−2.08%
omnetpp	1,447 KB	847 KB	−41.45%
povray	1,532 KB	1,520 KB	−0.79%
soplex	1,034 KB	551 KB	−46.69%
xalan	4,624 KB	3,983 KB	−13.87%

From a security perspective, all generated code structures are equal—memory is only read, and all memory indices are bounds-checked. The type ID is read only once and all intermediate computation happens in registers, no TOCTOU attack (time-of-check-time-of-use) is possible.

5.7.4 Binary Size and Memory Overhead

We evaluate the size impact of NoVT by compiling each program with and without protection, stripping the resulting binaries, and compare their size. Both protected and unprotected versions include a statically linked C++ standard library, and both versions run link-time optimizations over that standard library. Table 5.3 shows the size of the resulting binaries. It turns out that binaries usually get smaller after protection, with the exception of MariaDB. This might be counter-intuitive because each generated function should be larger than the vtable entries they replace, and many vtable entries are covered by more than one generated function. We identified two reasons for this observation: First, the virtual dispatch itself is smaller: in the protected version, a single `call` instruction to a generated function suffices to do a virtual dispatch; in vtable-based programs we would need a memory load and an offset calculation first. If the same dispatcher function is called from many locations we can save some bytes there. Second, our structure allows for more efficient dead code elimination: Our approach already identifies dead classes in its internal optimization steps and can remove them. From the final structure, a simple dead global elimination pass can identify virtual methods that are never called and rtti entries that will never be accessed. In particular, the C++ standard library contains much code that is not used in every program.

The memory overhead of our solution is negligible, memory usage reduced slightly for all tested programs. As we only change code and do not alter the memory layout of

objects, the used heap memory (including data segment) does not change. Given that LLVM compiles all our calls inside generated functions as tail calls, our solution does not use any additional memory on the stack. So the only difference in memory consumption comes from the different size of the binary (which is usually smaller).

5.8 Compatibility and Limitations

NOVT is compatible to both small and extremely large (e.g., Chromium) programs. We tested the correctness of all benchmarked programs with different inputs (usually from their benchmark suites). To this end, we compiled LLVM 9 and Clang 9 with NOVT and ran their quite extensive unit test suite (around 3800 tests) to confirm that no error was introduced by NOVT—excluding a single test that required dynamic linking, which NOVT does not support. CMake and MariaDB also have extensive test suites; we confirmed that NOVT does not alter the outcome of any test.

While all benchmarks were evaluated on the 64-bit x86 architecture, NOVT works for any architecture supported by LLVM. NOVT can, thus, for example, also protect mobile or embedded applications. To demonstrate this, we compiled a set of 40 small test programs for different architectures, including 32-bit x86, ARM, 64-bit ARM, MIPS, 64-bit MIPS, and 64-bit PowerPC. We verified that these protected programs work as expected by executing and using them in a QEMU emulator.

Our current prototype assumes the program to be compiled with Clang. We chose Clang due to its wide popularity and acceptance. Clang can build even the Linux kernel [180] and has interfaces compatible to GCC or Microsoft's C++ compiler. Having said this, porting our approach to other compilers is just a matter of engineering. Finally, we do not lose compatibility by using full LTO. While it increases compilation time, using full LTO improves runtime performance. Alternatively, NOVT could be implemented based on the faster "thin LTO".

Our approach has one main limitation: It is not compatible with dynamic linking or dynamic loading of C++-based libraries. This is an inherent drawback of all protection schemes that require full knowledge of the class hierarchy during build time. After the program is compiled, it is not *per se* possible to add more allowed classes to a specific virtual dispatch function. If a class from a runtime-loaded C++ library is now inheriting from a class already known, the new loaded class cannot be respected in the protection, calls to this class are not permitted. To the best of our knowledge, most vtable protections have this disadvantage [13, 69, 28, 172, 39], and those that support dynamic linking face either performance overheads [184] or weaker security guaranties [209, 106].

In fact, the lack of support for dynamic linking is not as crucial as it may look at first sight. First, programs compiled with NOVT can still dynamically link and load classical C libraries or C++ libraries that expose a C-style interface. Other C++ libraries can be compiled with NOVT and then be linked statically into the final binary—like we did for `libstdc++` in our experiments. Second, modern application deployment systems like Flatpak, Docker or Snap already bundle and ship an application together with all its dependencies. Applications packaged by such a system do not have any advantage of dynamic linking a library. Even without such a deployment system, some applications statically link the majority of their dependencies. For example, release builds of Google Chrome contain statically linked

libraries, including the C++ standard library. Third, many programming languages apart from C/C++ already use static linking as their default (or only) way of linking, including Go, Rust, Haskell and OCaml.

As a major endeavor, in principle one could extend NoVT to support dynamic libraries and dynamic loading of C++ code with an arbitrary interface. The method could be similar to TyPro's approach (see Section 6.7): a runtime library could collect class layout information from all libraries, analyze it, and generate new dispatchers using just-in-time compilation. While this significant improvement would increase the compatibility of our protection, we expect a negative impact on performance.

5.9 Related Work

5.9.1 Attacks on Vtables

Code-reuse attacks are still the most prevalent attack method on C(++) programs nowadays. C++ programs are particularly prone to call-oriented programming (COP) [19], as they usually contain many indirect calls (necessary for virtual dispatch). Schuster et al. [151] present *Counterfeit Object-oriented Programming (COOP)*—a new attack targeting C++ programs by using only valid vtables. COOP attacks chain virtual functions together in a way that resembles the original calling structure of a C++ program, breaking most vtable protection schemes available at that time. Haller et al. [55] revealed common errors in vtable protections and improved upon the precision of GCC's *VTV* protection scheme. They prove that their correction to VTV is optimal, given context insensitivity.

5.9.2 Vtable Protections

Seeing the popularity of vtable hijacking attacks, it is not surprising to see the wealth of literature on C++ vtable protection schemes. Table 5.4 summarizes related works, all of which enforce a C++-specific CFI policy. We group these into approaches that rely on compilers or binary rewriting.

Protecting binaries is naturally harder as vtables and virtual calls must be extracted from a (possibly stripped) binary. A first such approach was *T-VIP* [44], which enforced that vtable pointers point to read-only memory, preventing arbitrary control over the instruction pointer. Later *vfGuard* [64] and *VTint* [208] enforced a stronger policy—only detected vtables are allowed as targets of vtable pointers. The recent *VCI* [36] scheme can extract a class hierarchy from binaries and restrict the set of possible vtables further while still not achieving the precision of a compiler-based approach.

In contrast, compiler-based protections can get all necessary information from source code and hence have higher precision. *SafeDispatch* [69] was one of the first vtable-specific approach that protects virtual calls with little overhead. However, this little overhead could only be achieved by dynamic profiling of the application. SafeDispatch did protect not only virtual calls, but also included virtual offsets stored in vtables (with a weaker protection level, as outlined by ShrinkWrap [55]). It uses a class hierarchy analysis [31] to infer valid call targets. *Redactor++* [28] is another solution based on randomization and information hiding. While not enforcing CFI, it hides necessary vtable information such that

Table 5.4: Related work in comparison to NoVT, grouped by binary- and source-based solutions. While state-of-the-art binary defenses are still struggling with COOP attacks, source-based defenses have solved this issue for some time. However, surprisingly many solutions are not optimal as outlined by ShrinkWrap. Similarly, only few solutions protect other vtable usages than calls.

Type	Related Work	Protection: calls / offsets / rtti & casts	Handles method pointers	Defeats COOP	Optimal (1)	Runtime Overhead	Remarks
Binary	T-VIP [44]	✓××	×	×	×	∼25% (SPEC)	requires profiling
	vfGuard [64]	✓××	×	×	×	18.3% (Internet Explorer)	
	VTint [208]	✓××	×	×	×	0.4% (SPEC), 1.4% (Chromium*)	instruments less calls than other solutions
	VCI [36]	✓××	×	partial	×	7.79% (SPEC + Chromium*)	
Source-code based	VT-Guard [106]	✓××	✓	×	×	unclear	patent does not detail its performance penalties
	SafeDispatch [69]	✓✓×(2)	✓	✓	✓	2.1% (Chromium)	requires profiling for performance
	VTV [184]	✓××	✓	✓	×(3)	1% - 8.7% (SPEC)	requires profiling for performance
	Redactor++ [28]	✓××	✓	✓	✓	8.4% (SPEC), 7.9% (Chromium)	probabilistic defense, requires execute-only memory
	LLVM-VCFI [172]	✓××	×	✓	×	1.97% (SPEC), 2.9% (Chromium) [13]	
	VTrust [209]	✓✓✓(4)	×	✓	×	2.2% (partial SPEC)	
	OVT/IVT [13]	✓××	×	✓	✓(5)	1.17% (SPEC), 1.7% (Chromium)	ShrinkWrap-safe configuration doubles overhead on SPEC
	VIP [39]	✓××	×	✓	✓	0.7% (SPEC)	compilation can take hours
	CFIXX [17]	✓××	✓	✓	n.a.	4.96% (SPEC)	Object Type Integrity, not CFI. Requires MPX CPU instructions
	NoVT	✓✓✓	✓	✓	✓	−0.9% (SPEC), 1.1% (Chromium)	

(1) The solution restricts possible calls to the minimal possible set, as shown in ShrinkWrap [55]
(2) Virtual offset checks are not optimal and have additional runtime overhead
(3) ShrinkWrap [55] proposed a fix (without additional runtime overhead)
(4) For virtual offsets, RTTI access and dynamic casts, their check is weaker (no check if the valid vtable matches the static type)
(5) optional, with higher runtime overhead
(*) Only a few benchmarks have been used (*octane* etc.), but no HTML or rendering workloads (which typically have more overhead)

COOP attacks require hardly feasible guesswork. Its defense is probabilistic and could be circumvented in some settings [97], and it requires a system offering execute-only memory. *VTrust* [209] improves over these solutions in terms of compatibility, and protects vtables without knowledge of a full class hierarchy. VTrust protects not only virtual dispatch but also virtual offsets and type information, but at a much weaker level: when resolving virtual offsets or loading rtti, any vtable can be used, the check is type-agnostic.

Despite these academic progresses, major C++ compilers use their own vtable protections: Microsoft included a canary-based solution named *VT-Guard* [106] in their Visual C++ compiler. This solution is not strong enough to prevent COOP attacks. Recently Microsoft announced *xFG* [158] with improved, but not optimal security. GCC included a method called *VTV* [184] which has been improved by ShrinkWrap [55]. VTV focuses on compatibility, but has a non-negligible runtime overhead (and again requires dynamic profiling to achieve its performance). LLVM has its own forward-CFI approach [172] that includes a vtable-specific protection. While this solution is more performant than VTV, it is neither optimal nor complete. Again, all of these solutions focus on virtual dispatch exclusively.

Recent work has brought different improvements. Bounov et al. introduced *OVT* and *IVT* [13], two protections striving to improve performance by ordering and interleaving vtables. They achieve an overhead as low as 1.17% with a non-optimal protection (and the possibility to turn it into an optimal one). VIP [39] improves the security guarantees of vtable protections, introducing a pointer analysis technique that is used to reduce the set of possible vtables in a way that is not possible without context. However, their analysis takes up to an hour on SPEC and 6 hours on Chromium. As an alternative to vtable-based CFI schemes, *Object Type Integrity* protects vtables pointers (instead of protecting virtual calls) [17]. Their prototype *CFIXX* has a reasonable overhead, but requires Intel's deprecated MPX CPU extensions.

NoVT improved over all previous solutions in terms of performance and protection. We protect all usages of a vtable, including virtual offsets, type information and dynamic casts, while most previous solutions only protected virtual calls. Our protection is optimal in a context-free setting (as shown in [55]), for all protected usages. To the best of our knowledge, we are the first vtable protection that actually speeds up most of the programs it is applied to. At the same time, our solution lives with the same limitation (no dynamic linking) than most previous source-based solutions [69, 28, 172, 13, 39]. Solutions that (partially) support dynamic linking have a higher overhead [184, 17, 209] or lower security level [106, 209].

5.9.3 Alternatives to Vtables

Decades ago, similar to our general idea, the compiler community explored alternatives to vtables. Although these approaches did not have a security focus and hence also did not discuss security-critical considerations in this respect, we will briefly describe them.

Porat et al. [139] used static type checks and direct calls to speed up hot paths in virtual dispatch—virtual method implementations for frequently used classes are called directly, while less frequent classes fall back to classical vtables. Vtables are not changed, and no type IDs are introduced. However, this approach only offers small speed improvements but no security gain.

SmallEiffel [206] was an experimental compiler for the Eiffel programming language that does not use vtables. Instead, it uses type IDs for classes and a binary search tree for virtual dispatch. While the basic idea of type IDs is similar to NOVT, SmallEiffel's design can't be easily ported to C++, because Eiffel lacks many features like mulitple or virtual inheritance that C++ has. Furthermore, SmallEiffel has an unclear security contribution because unexpected type IDs trigger undefined behavior. In contrast, NOVT's type identification system is more advanced, can deal with all object-oriented features of a modern language like C++, has a very strong focus on security and the generated dispatchers can deliver better performance than binary search trees.

5.9.4 Replacing Pointers with Identifiers

HyperSafe [195] shares some design ideas with NOVT: To protect C-style indirect function calls and returns in hypervisors, HyperSafe replaces function pointers and return addresses with function IDs. At each indirect control transfer, the ID is checked and resolved through a per-callside lookup table. In contrast, NOVT targets C++, is usable in general, and uses flexible switch instructions instead of lookup tables.

µRAI [6] protects return targets on microcontrollers. Partial paths in the control-flow graph get a "function ID" assigned, one register is reserved to maintain the current ID at runtime. Instead of common return instructions and return addresses stored on the stack, µRAI can determine the unique correct return address from the current ID. A jumptable over all possible IDs replaces the return, removing return addresses on the stack altogether.

TYPRO [P2], which we present in Chapter 6, assigns IDs to functions and builds similar `switch` statements to replace C-style indirect calls.

5.10 Conclusion

Vtable hijacking attacks are a real threat to many applications written in C++, including high-value targets like browsers. NOVT uses a modified compiler to protect programs given complete source code (including libraries). We thereby radically change the way of *protecting* vtables and, instead, *eliminate* them. NOVT replaces traditional, vtable-based virtual dispatch with direct calls—based on a class identifier in each C++ class instance, it calls an object's method non-virtual. Using a class hierarchy analysis at link-time, NOVT determines which method implementations are possible for each virtual call; at runtime, only these methods are callable. After protection with NOVT, no traditional vtables or vtable pointers remain in the program.

NOVT is compatible with all C++ programs, except dynamic linking and loading—as with most previous solutions, NOVT relies on knowledge of the complete class hierarchy. Legacy software can easily be protected without any source code modifications or additional dependencies. According to [55], NOVT's offered protection level is optimal for a type-based solution. NOVT can defend against strong vtable-based attacks like COOP, even for large programs. NOVT has been evaluated on SPEC CPU 2006, Chromium, MariaDB, Clang, CMake, and Tesseract OCR. The introduced performance overhead is often negative, −0.5% on average and 2% in the worst case. The generated binaries get usually smaller; no memory overhead is introduced, and the impact on compilation time is minimal.

Availability

Our prototype has been released as Open-Source Software; it is available on Github:
`https://github.com/novt-vtable-less-compiler/novt-llvm`

6
TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation

6.1 Motivation

Maliciously-overwritten function pointers in C programs often lead to arbitrary code execution. In principle, forward CFI schemes mitigate this problem by restricting indirect function calls to valid call targets only. However, existing forward CFI schemes either depend on specific hardware capabilities, or are too permissive (weakening security guarantees) or too strict (breaking compatibility).

We describe TyPro, a Clang-based forward CFI scheme based on type propagation. TyPro uses static analysis to follow function pointer types through C programs, and can determine the possible target functions for indirect calls at compile time with high precision. TyPro does not underestimate possible targets and does not break real-world programs, including those relying on dynamically-loaded code. TyPro has no runtime overhead on average and does not depend on architecture or special hardware features.

6.2 Problem Description

Code-reuse attacks exploit memory corruption vulnerabilities by overwriting code addresses with references to malicious code, ultimately gaining arbitrary code execution. Control Flow Integrity (CFI) aims to mitigate code-reuse attacks by enforcing that return and indirect call targets are valid [1]. We have already covered methods that protect return addresses in Chapter 4 and C++ virtual calls in Chapter 5. The remaining part are indirect calls to function pointers in C, which still lack protection so far. In this chapter, we thus focus on forward CFI in C programs.

In the C language, an indirect call invokes a *function pointer*. Forward CFI schemes check this pointer before the call, ensuring it points to a "valid" target. To this end, industry-grade and widely deployed forward CFI schemes (Microsoft's Control Flow Guard [101], or Intel CET's indirect branch tracking [155, 136]) merely test if *any* valid function is called. This crude over-approximation enables attackers to call functions that are not reachable from the given call site. Therefore, more precise forward CFI schemes compute a tailored *target set* for each indirect call, containing all function pointers that are allowed for this call. On the one hand, the target set must be large enough to allow every intended function pointer. Otherwise, the protected program may crash. On the other hand, the target sets must be minimal, as every unnecessary target represents a gadget that attackers might use in code-reuse attacks. The most promising example of such a more precise scheme is Clang's CFI [170]. It checks the C type of the called function and compares it to the expected indirect call type. Such compiler-integrated analyses allow for easy integration in off-the-shelf software—significantly easing wide CFI deployment.

Unfortunately, as we will show in Section 6.9.1, Clang CFI and its strict type checks regularly miss valid call targets. Ultimately, this impreciseness may lead to unforeseen program crashes during runtime, which *cannot* be detected beforehand. That is, whether or not a CFI-protected program crashes is only known (at some point) during runtime. Due to these deficiencies, even hardened OSes cannot deploy Clang CFI to all applications. To quote the HardenedBSD maintainers: "*We may need to disable cfi-icall [Clang's forward CFI for C-style function pointers] for more applications, and we'll need to rely on our user base to identify edge cases.*" [56]

Clearly, this is an unsatisfying state. As a non-solution, one could revert to more permissive CFI schemes. For example, IFCC [184] just compares the *number* of arguments, but not their *types*. But this policy allows many unnecessary targets, bloating the surface of code-reuse gadgets (cf. Section 6.9.2). Seeing that such weaker CFI designs unnecessarily undermine security, we seek to understand the root causes of Clang CFI's failures. We identify three conceptual reasons why Clang CFI is incompatible with popular software projects such as lighttpd, nginx, or redis. First, Clang CFI lacks type propagation. Therefore, it does not allow casts to/from undefined types (`void *`) that programmers often use to build inheritance-like constructs. Second, Clang CFI does not support variadic functions. Third, Clang CFI does not support dynamic linking.

As a workaround, developers could try to rewrite programs that cause CFI incompatibilities. However, this is a non-trivial task, requires significant code revisions, and comes at the price of losing flexibility. For example, plugin interfaces (e.g., in nginx and lighttpd) heavily rely on variadic functions or generic APIs that operate on `void *` pointers. Furthermore, the lack of support for dynamic linking impedes CFI adoption and cannot be "fixed" by refactoring.

Alternatively, and this being the idea of this work, we can design a CFI system for Clang that supports the above compatibility features. However, the C standard foresees concepts that impose challenges, such as (i) function pointer casts, (ii) function pointers that are part of compound data types such as `struct` and `union`, or (iii) function pointers that are propagated through other indirect calls. Supporting these features is vital to not break programs. Moreover, any function pointer analysis must be context-sensitive to refine the set of valid call targets. That is, not only do we have to match function types, but we *also* have to verify that a given pointer can ever be used as an indirect call target in a benign execution path. Finally, the target sets may *change* when new program parts are loaded during runtime. However, existing CFI schemes often assume a static setting and cannot support shared libraries.

To tackle some of these challenges, existing forward CFI schemes (i) use dynamic or runtime analysis [74, 63, 34], (ii) require kernel-level modifications [34, 63, 186, 45] or orthogonal defenses such as shadow stacks to be in place [73, 74], or (iii) rely on architecture-specific features to recognize valid call targets in real-time [73, 34, 63, 186, 45, 118, 98, 74]. Therefore, these solutions sacrifice generality and are not agnostic to the underlying OS and hardware. Thus, we still lack a generic and software-only forward CFI solution that is neither too permissive nor too restrictive.

6.3 Contributions

In this chapter, we propose TyPro, a drop-in replacement for Clang's forward CFI scheme. TyPro uses static analysis to propagate function types (1) to gain a compatible CFI scheme for Clang, and (2) to tackle the open challenges in existing forward CFI systems. To this end, we extract types and casts from a program's Abstract Syntax Tree (AST), and follow how they propagate to other functions (i.e., contexts). We derive rules from the C standard that capture all permitted type propagations. We then leverage a solver that uses the type information and propagation rules to extract accurate *target sets*, i.e., functions that are valid for a given call target. We enforce these target sets by rewriting indirect function calls

with switch/case constructs that can no longer be abused for function pointers other than those in the target set.

We developed TyPro as an LLVM-based open-source prototype[1]. TyPro is a software-only solution, fully compatible with dynamic linking and loading of shared libraries. Our systematic conformity to the C standard pays off, especially for large real-world programs: Our protection does not break legacy code and computes target sets more precisely than industry standards (CFGuard, CET) or IFCC. Furthermore, TyPro is efficient and does not cause measurable runtime performance overhead in protected applications.

6.4 Overview

6.4.1 Attacker Model

TyPro aims to defend against function pointer corruptions in C programs, where an attacker wants to divert control flow to execute arbitrary code. Our attacker model bases on Section 2.3: We consider attackers that know the program's memory layout and can read from and write to all memory locations within the boundaries of page permissions. We assume `W⊕X`, i.e., that no pages exist that are both writable *and* executable. TyPro protects forward edges (function pointers), so we assume that return addresses are covered by any other orthogonal scheme from Chapter 4. Furthermore, while we consider dynamically-*loaded* code, we exclude dynamically-*generated* code such as just-in-time compilation, and suggest additional protections [130, 207, 159] if necessary.

6.4.2 Challenges

To build a secure forward CFI, one has to solve the challenge of finding a precise set of allowed target functions for each indirect call in C. Not every function that is ever referenced by a function pointer—we refer to those as *address-taken functions*—is a valid call target for any indirect call site. There are two validity conditions to be checked, whereas existing CFI schemes only consider the first. (1) The type of the address-taken function "matches" the type of the function pointer used in the indirect call. For example, an indirect call that passes just a single argument is clearly incompatible with functions that expect multiple arguments. Types do not necessarily have to be identical but should be "compatible". (2) There is a program execution in which the function's pointer will be passed to the respective indirect call site. That is, assume the function signatures of two functions `A` and `B` are identical, but `B`'s pointer is kept local in a function unrelated to the call. Then, `B` is never a valid call target.

Related software-only CFI schemes only perform function type checking and ignore the function's context. For example, Clang CFI [170] and MCFI [129] perform a rigid function type checking. While this strict type checking is intuitive and straightforward, it is too restrictive and regularly corrupts programs. Indeed, called functions may have different types than the function pointers. In the code example shown in Figure 6.1, we see a trivial example where this happens in lines 8–12: The indirect call in line 12 targets the function `f1` (expecting an argument of type `long`), which has a different signature than the function

[1] `https://github.com/typro-type-propagation/TyPro-CFI`

```
 1 typedef void (*fptr_long)(long);
 2 typedef void (*fptr_int)(int);
 3 typedef void (*fptr_ptr)(fptr_long);
 4 void f1(long a) {}
 5 void f2(long a) {}
 6 void f3(long a) {}
 7
 8 void scene1_a() {
 9     fptr_int f = (fptr_int) &f1;
10     scene1_b(f);
11 }
12 void scene1_b(fptr_int f) { f(0); }        // call1
13
14 struct S { fptr_long one; fptr_int two; };
15 void scene2_a() {
16     struct S s = { &f2, 0 };
17     scene2_b(&s);
18 }
19 void scene2_b(struct S *s) { s->one(0); } // call2
20
21 fptr_long callback;
22 void set_callback(fptr_long f) { callback = f; }
23 void scene3_a() {
24     fptr_ptr some_cb_target = &set_callback;
25     some_cb_target(&f3);                   // call3
26 }
27 void scene3_b() { callback(0); }          // call4
```

Figure 6.1: Code example showing different ways to transfer function pointers.

pointer in argument `f` (expecting an argument of type `int`). Consequently, a strict type check will not allow this function call and will mistakenly terminate the program.

As we will also experimentally show, this strict function type checking is impractical for many programs. On the other extreme, we may thus consider forward CFI schemes with less restrictive type checks. For instance, one could simply count the number of arguments instead of validating their type, like IFCC [184]. However, such generous "type matching" allows significantly more valid call targets than required for correctness, increasing the attack surface. Regarding our code example, for the indirect call in line 12, IFCC considers not only `f1` as a target, but also any other function with only one argument present in the codebase. However, in principle, the indirect call in `scene1_b`, with callee of type `fptr_int` can only target `f1`, but not any other function. Such over-permissive CFI systems unnecessarily bloat the attack surface for code-reuse attacks.

6.4.3 Methodology at a Glance

We aim for a sweet spot between the "too permissive" and "too restrictive" forward CFI schemes. In particular, our goal is to correctly track function types even in (typical) situations, that are not covered by existing strict CFI schemes, such as the following three: (i) *A function pointer is cast to another type.* For example, regarding Figure 6.1, the invocation of `f` (line

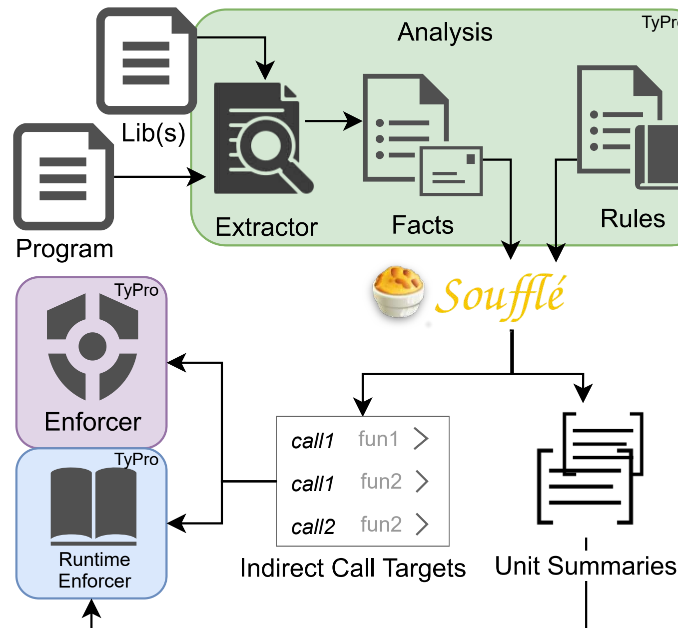


Figure 6.2: TyPro's workflow.

12) requires such tracking, as discussed before. (ii) *A function type is hidden in compound data structures such as struct, union, pointer, or array.* In Figure 6.1, lines 14–19 provide such an example, where the function pointer `f2` is part of struct `S` that is passed as pointer to the caller function `scene2_b`. And (iii), *a function pointer is propagated through other indirect calls*, such as the example in lines 21–27, where the call target for *call4* depends on the arguments and target of the indirect *call3*. None of the current strict CFI schemes correctly cover all these three situations, which causes them to regularly fail many real-world programs (as demonstrated in Section 6.9.1).

While our methodology seems similar to common data-flow analysis, it has a major difference: we track types only, not actual data. There are much less types in a program than values, and tracking them is much simpler, therefore our analysis is much more lightweight than data-based approaches.

We propose a static analysis method that propagates function types to tackle the challenge of collecting restricted yet correct sets of call targets. Our workflow is presented in Figure 6.2. TyPro operates on source code in the form of an abstract syntax tree (AST). We track types per function, including casts between them, and we track which types are exchanged between functions, even in nested data types. If there is a propagation path from an address-taken function's type to an indirect call's type, we know that this function can be a valid target.

Our type propagation is roughly split into three phases. First, we extract "initial" type information from the AST. Consider the indirect call in lines 8–12 in Section 6.4.2. Here, our analysis collects `f1`'s and `fptr_int` type declarations, a cast from `f1`'s type `fptr_long` to `fptr_int`, and also that `fptr_int` is used as `scene1_b` argument type. We store this information as *facts*—logical formulas that are *assumed* to be true before the analysis starts. Facts represent one of the Horn clause types [60]. This encoding has been employed

by a plethora of analyses for various properties [12, 41] as it allows for leveraging advanced solvers [113, 71] to compute the actual result.

Second, we use the facts to find a set of valid targets for every indirect call in the code at link-time. To this end, we first specify a set of *rules*—another Horn clause type. They are logical implications that describe how to *derive* new information (rule's body) starting from the initial facts or the information derived being true by the previous rules' applications (rule's head). If the preconditions encoded in the body are satisfied, we call the rule *applicable*. In particular, our rules describe how to compute all possible type propagations permitted by the C standard. They thus form the basis for computing the final set of allowed functions (i.e., the *target set*) for each indirect call.

Third, we supply the facts and the rules to a solver that derives the minimal target sets while applying the rules mentioned above. This process continues to the point when rule applications do not derive any new target for any indirect call. In our example, the solver concludes that `f1`'s type is the only type that propagates to *call1*, and consequently, `f1` is the only valid function in *call1*'s target set.

Finally, after the type propagation analysis stage, when generating the program binary, we *enforce* the resulting target sets. We assign an ID to each address-taken function and replace the function's address with that ID. We transform every indirect call into a switch over different direct calls that only target functions in the valid target set. This method is similar to the C++ vtable protection NoVT from Chapter 5. To support dynamic linking and loading, a *runtime library* updates the target sets whenever new modules appear at runtime via just-in-time compilation. The final result is a protected program binary, which runs everywhere the original binary can run. This program has no additional dependencies except the runtime library shipped together with the program, which can optionally be statically linked.

6.4.4 Type Propagation vs. Data Flow

In contrast to the existing techniques based on data flow [73, 63, 186, 34, 74, 202], our approach does not compute the type evolution during program execution. Data flow tracking is precise but an expensive computation, as it requires several non-trivial components (e.g., control flow). Instead, our type propagation performs light-weight processing of the information extracted only from the points in the program's source code where types are created, manipulated (e.g., through casts), and used. Although employing data flow-based approaches may help to obtain even more precise approximations while computing indirect call targets, our experimental evaluation (presented in Section 6.9) demonstrates the effectiveness of our system in reducing the target sets for indirect calls. In other words, we do not see any drastic effects from the approximations we use, while the absence of flow information helps to make the compilation time tractable, as discussed in Section 6.10.

6.5 Target Set Computation

Our analysis technique starts by extracting *facts* about the program's code. Then we use the facts in the program-independent *rules* computing the type propagation. We use the sample code shown in Figure 6.1 to highlight the propagation rules necessary for our analysis's

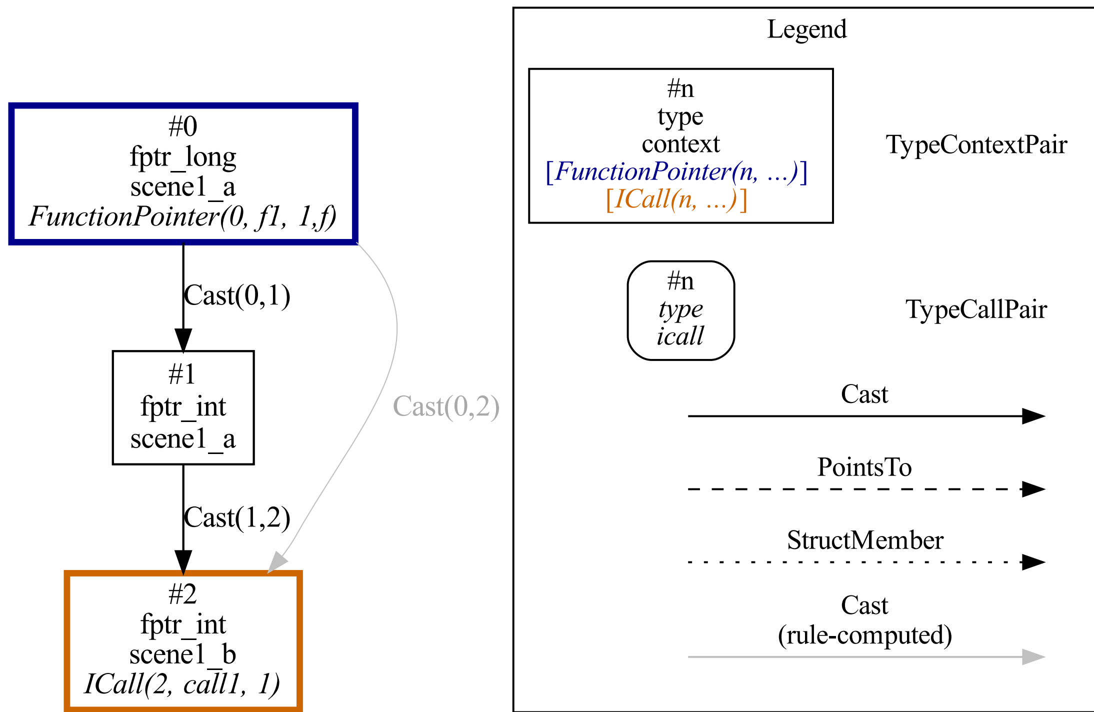


Figure 6.3: Graphical representation of the collected and derived facts for "`scene1_a`" and "`scene1_b`".

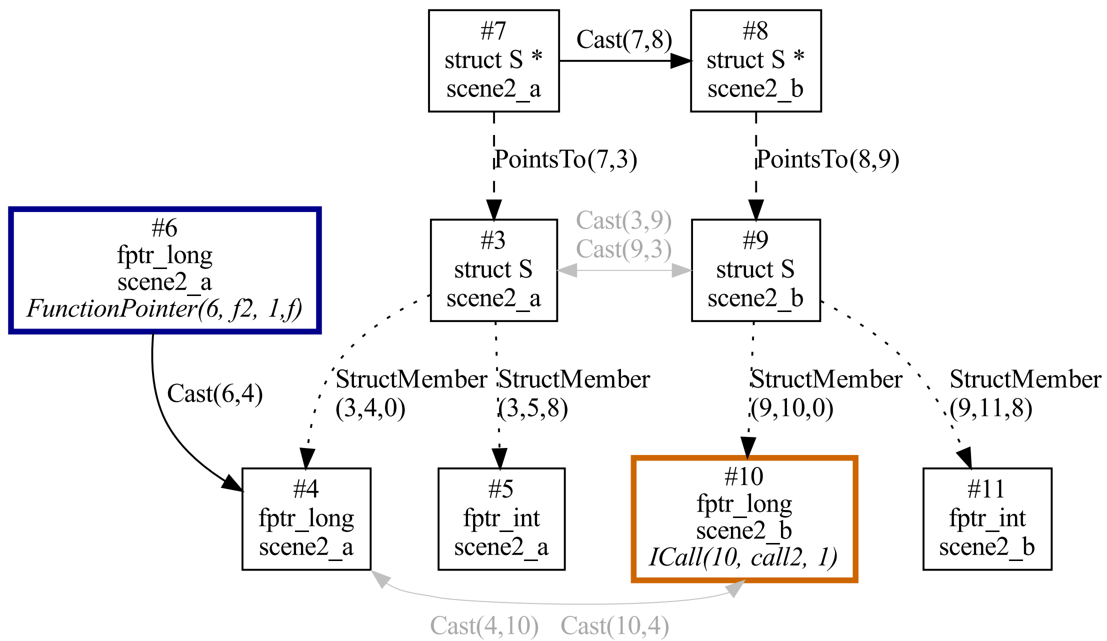


Figure 6.4: Graphical representation of the collected and derived facts for "`scene2_a`" and "`scene2_b`".

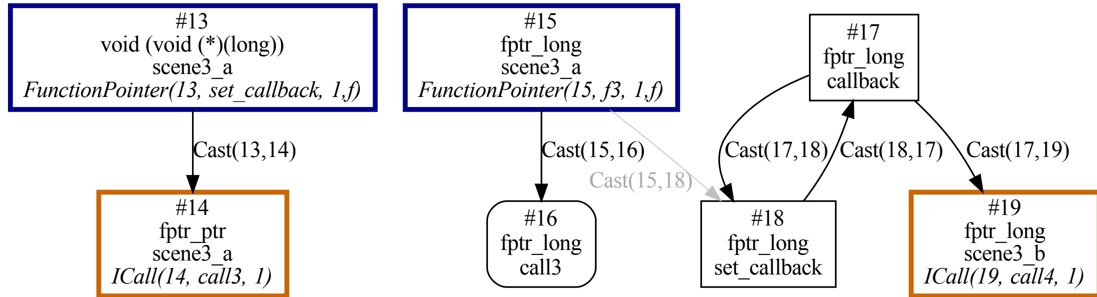


Figure 6.5: Graphical representation of the collected and derived facts for "`scene3_a`" and "`scene3_b`".

TypeContextPair : $\mathbb{N} \times \mathbb{S} \times \mathbb{S}$	(available types and their contexts)
TypeCall : $\mathbb{N} \times \mathbb{S} \times \mathbb{S}$	(types of indirect call parameters)
PointsTo : $\mathbb{N} \times \mathbb{N}$	(pointer to pointee mapping)
StructMember : $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$	(struct fields with memory offset)
UnionMember : $\mathbb{N} \times \mathbb{N} \times \mathbb{S}$	(union fields with field type)
Cast : $\mathbb{N} \times \mathbb{N}$	(type propagations)
ICall : $\mathbb{N} \times \mathbb{S} \times \mathbb{N}$	(indirect calls for callee type)
FunctionPointer : $\mathbb{N} \times \mathbb{S} \times \mathbb{N} \times \mathbb{B}$	(address-taken functions)
TargetSet : $\mathbb{S} \times \mathbb{S}$	(possible indirect call targets)

Figure 6.6: Predicate signatures for the facts used by analysis.

correctness. As the result of our analysis, we obtain the possible target functions set for every indirect call in a single module of the program. Later, in Section 6.7, we will extend this technique from a single module to multiple modules linked at runtime.

6.5.1 Analysis Input Generation

Our pipeline starts after Clang's parser produces an AST representation of the C program, which intuitively constitutes a collection of expressions [173]. From this tree, we extract only the information relevant to types, their propagation and usage. We then create the corresponding Horn clause facts, as introduced in Section 6.4.3. Our facts use several boolean relations, as shown in Figure 6.6. Each relation has a corresponding predicate signature which determines its domain. For instance, predicate signature $\mathbb{N} \times \mathbb{S} \times \mathbb{S}$ for relation TypeContextPair expresses that facts in this relation take three arguments, the first one is a natural number $\mathbb{N}$ (a unique identifier), the second and the third are strings $\mathbb{S}$ (in this case: a type name and a function's name). Facts are specific to each program. We collect the facts for each compilation unit (source file) and merge on linking. Overall, we split facts into four groups depending on the information they contain: initial types and context, type constructs, functions & calls, and casts & transfers:

Initial Types and Context Collection. Before we can follow a type's propagation, we first need to know all types in the program's code. In addition, we require a *context* annotation

for each type. The context is a function or a global variable where C expressions occur. Using pairs of types and context instead of pure types improves the precision of the analysis—type propagations can be kept local to a group of functions. With contexts considered, type propagations from unrelated parts of the source code will not influence the computation. We leverage the AST to extract unique pairs of C types and their context to get the required information. In particular,

- For every expression e with type t in function f, we collect the pairs (t, f). For example: (`fptr_int`, `scene1_a`).
- For every function declaration f with return or parameter types $t_0, \ldots, t_m$, we collect the pairs $(t_i, f), i \in \{0, \ldots, m\}$. For example: (`fptr_int`, `scene1_b`).
- For every global variable g of type t, we collect the pair (t, g). For example: the pair (`fptr_long`, `callback`). If g is initialized, we also collect the pairs (t', g) for every expression e with type t' inside the initialization code.

We assign each unique pair an identifier $n \in \mathbb{N}$. This allows us to refer to each pair by number, simplifying the rules, saving storage, and improving computation time as outlined in Section 6.8. To this end, we define a function $N : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{N}$ that returns the identifier n for each collected type-context pair. For convenience, we define its second version $N : \mathbb{S} \rightarrow \mathbb{N}$ that returns the identifier n for each AST expression, based on its C type and the corresponding context object. We store each type/context pair (t, c) together with their corresponding identifier $n = N(t, c)$ as fact TypeContextPair(n, t, c).

In our example (Figure 6.1), line 9 corresponds to several AST expressions encoded as fact TypeContextPair(0, `fptr_long`, `scene1_a`) as we use function pointer `&f1` of type `fptr_long` in the context of function `scene1_a`, and TypeContextPair(1, `fptr_int`, `scene1_a`) as we cast `&f1` to type `fptr_int`. Similarly, line 19 produces the fact TypeContextPair(8, `struct S *`, `scene2_b`), as we declare parameter of type `struct S *` for function `scene2_b`. Recall that also global variables can be used as a context. For example, we define a global variable `callback` in line 21, which adds the TypeContextPair (17, `fptr_long`, `callback`) fact.

Type Constructs. To correctly track type propagation in complex data structures, we must collect the construction information of all derived types in a program. In particular, for every fact TypeContextPair(n, t, c), we do the following depending on t:

- If t is a pointer or array type ($t := t'*$ or $t := t'[\ldots]$), we compute the identifier $n' = N(t', c)$, add another fact TypeContextPair(n', t', c), and also store the pointer or array type structure as PointsTo(n, n') fact.
- If t is a structure type ($t :=$ `struct{`$t_0, \ldots, t_m$`}`), we compute the identifiers $n_i = T(t_i, c)$ with $i \in \{0, \ldots, m\}$ where each type t_i corresponds to a struct field. Furthermore, we add fact TypeContextPair(n_i, t_i, c) for each field, and record the type structure as StructMember(n, n_i, x_i) facts, where x_i is the byte offset of field i in the struct's layout.
- If t is a union type ($t :=$ `union{`$t_0, \ldots, t_m$`}`), similar to the structure type, we add facts TypeContextPair(n_i, t_i, c) for each member, but record the type structure as UnionMember(n, n_i, t_i) facts.

Figure 6.4 visualizes the facts that we derive for our example code, they represent a type's construction in a tree-like form. The type constructs analysis starts with fact TypeContextPair(8, `struct S *`, `scene2_b`)—as obtained during the initial types and context collection—that implies that `struct S*` is a pointer type. Hence, we introduce another fact for the pointee TypeContextPair(9, `struct S`, `scene2_b`) and store the points-to information as the fact PointsTo(8, 9). Consequently, we check the new fact TypeContextPair(9, `struct S`, `scene2_b`) and as it is a structure type fact (as shown on the line 14) we also add facts TypeContextPair(10, `fptr_long`, `scene2_b`) and TypeContextPair(11, `fptr_int`, `scene2_b`), plus the facts capturing type structure: StructMember(9, 10, 0) and StructMember(9, 11, 8).

Functions & Calls. To determine the target sets for indirect calls, we need to identify all function pointer types that propagate to each indirect call. To this end, we need to know all functions taken as function pointers (address-taken functions) and all types of the functions that get indirectly called—the start and destination of our propagation path, respectively. To this end, we collect fact TypeContextPair(n, t, c) plus the additional information from expression e if one of the cases applies:

- **Address-taken functions**: If e represents the address of a function f outside of a direct call, we store the function with its type, its number of arguments m, and whether it accepts a variable number of arguments $vararg$ in FunctionPointer($n, f, m, vararg$) fact. In our example, we produce facts FunctionPointer(0, `f1`, 1, false) for `&f1` in line 9, and FunctionPointer(13, `set_callback`, 1, false) for the AST expression "`&set_callback`" in line 24.

- **Indirect calls**: For indirect calls, i.e., $e := e'(a_1, \ldots, a_m)$, we record the callee's expression identifier, a reference to the call expression, and the number of arguments in fact ICall($N(e'), e, m$). In our example, line 12 produces fact ICall(10, *call2*, 1), and line 25 creates fact ICall(14, *call3*, 1).

Casts & Transfers. After having collected types, start and destination of type propagation paths, we need to know the actual propagation steps: We look for the specific AST expressions that transfer the type or context of another expression. This information populates Cast facts. In other words, we do not differentiate between type casts or context transfers in our analysis because it operates on type/context pairs. In particular, for an AST expression e, for which we also collect the fact TypeContextPair(n, t, c), we check for these cases:

- **Type casts**: For a cast $e := (t)\ e'$ and sub-expression e' with TypeContextPair(n', t', c), we produce a fact representing a cast from t' to t: Cast(n', n). This kind of fact covers all casts that C supports, including implicit casts and qualifier casts. In our example, we have a cast in line 9, casting an expression of type `fptr_long` to `fptr_int` in the context of function `scene1_a`. Using the corresponding type/context identifiers (obtained during the initial type and context collection) that the analysis stores in the TypeContextPair facts, we record this as Cast(0, 1), see Figure 6.3.

- **Global variable uses**: If expression $e := g$ accesses a global variable g, having a corresponding TypeContextPair(n', t, g) fact, we record this as an implicit cast: Cast(n', n). Intuitively, this cast allows us to derive type t propagation from the

global's context to the context of expression e. If this access could be a write, we need to account not only for context transfer from the global context but to the global context itself. In other words, type t should become accessible at the global context. We capture this bidirectional context transfer by adding fact Cast(n, n'). Note that the type of the global variable t equals the type of the expression accessing it, only the context changes.

In our example, there is a global variable `callback` which is written in function `set_callback` and accessed in function `scene3_b`, see Figure 6.5. For the write access, we record facts Cast$(17, 18)$ and Cast$(18, 17)$. For the read access, we record Cast$(17, 19)$.

- **Direct calls**: Similar to global variables, direct calls also manipulate the context. In a call $e := f'(a_1, \ldots, a_m)$ to a function with declaration $f'(p_1, \ldots, p_m)$, we produce for every argument a_i a separate cast fact: Cast$\left(N(a_i), N\left(p_i, f'\right)\right)$. For the return value of type rt, we also produce a cast fact: Cast$(N(rt, f'), n)$. In our example, `scene1_b` is called in line 10, with an argument represented as TypeContextPair(1, `fptr_int`, `scene1_a`). We capture this by recording Cast$(1, 2)$, see Figure 6.3. Similarly, for the call from `scene2_a` to `scene2_b` in line 17, we record Cast$(7, 8)$ for its first argument.

- **Indirect calls**: For indirect calls $e := e'(a_1, \ldots, a_m)$, the type of e' is a function pointer $t_{e'} = rt(*)(p_1, \ldots, p_m)$ with parameter types p_i and return type rt. We want to store the casts as we do for a direct call. To this end, we use a new relation TypeCall, which uses indexing similar to TypeContextPair, but records a reference to indirect call e (also discussed in *Functions and Calls*) instead of a context. We add the facts TypeCall(n_i, p_i, e) and TypeCall(n_{rt}, rt, e), also we define $n_i = N(p_i, e)$ and $n_{rt} = N(rt, e)$ for $i \in \{1, \ldots, m\}$ accordingly. With these new facts, we can then add the casts similar to a direct call: Cast$(N(a_i), n_i)$ and Cast(n_{rt}, n).

  Note that a TypeCall fact does not contain the context information in which the type of the argument (in the indirect call) is defined. Ultimately, this approximation allows our analysis to extract all possible contexts for each of the types used in an indirect call, which is crucial for the correctness of the target set computation that we discuss in Section 6.5.2. In our example, we have indirect call *call3* in line 25. In addition to indirect call fact ICall discussed earlier, in this case, we produce a cast fact for its first argument Cast$(15, 16)$ and a TypeCall(16, `fptr_long`, *call3*) fact (see Figure 6.5). In Section 6.5.2 we show how these facts are used to relate this indirect call parameter type to the type its argument FunctionPointer(15, `f3`, 1, false).

6.5.2 Type Analysis

Having obtained *facts* from the AST, we now describe how we use these facts for type propagation reasoning. Our type analysis is based on a collection of *rules*. We define these generic rules as constrained Horn clauses [60] *once*, i.e., they are program-independent. The rules thus specify the logic behind our analysis type reasoning. They follow type propagations across functions, through nested compound types and indirect calls. Finally, they specify how to obtain from these type propagations a set of possible targets for each

indirect call as a result of TargetSet relation computation: For every function f, which is a valid target for indirect call c, the rules show how to derive TargetSet(c, f).

Figure 6.7 depicts the rules we use to compute TargetSet relation containing the final result, i.e., targets for each indirect call. We obtain this result when the analysis cannot extend TargetSet relation via rule application; in other words, no more information can enter the relation. Note that all variables used in the rules have the types corresponding to the predicate signature presented in Figure 6.6. Moreover, these variables are universally quantified—each rule can be used multiple times for each variable assignment that makes it applicable. Ultimately, TargetSet contains a mapping from indirect calls to their possible targets.

The first group of rules, which consists of the rules (T), (P), (S), and (U), allows our analysis to extend the Cast relation. Computation of this relation allows our analysis to account for transitivity of multiple propagations (rule (T)), pointer aliasing (rule (P)), and structures & unions (rules (S) and (U)). Intuitively, the Cast relation captures all possible propagation paths of the types and contexts in the code.

The second group of rules—(F1) for fixed and (F2) for variable number of arguments—use the Cast relation to compute our final result, the TargetSet relation. In particular, we access the computed propagation paths between function pointers and indirect calls.

However, with the first two groups of rules, we can compute only a partial result for the TargetSet relation. The last group of rules (IC1, IC2) uses this partial result to enrich TargetSet, accounting for context casts from the arguments of an indirect call to its actual target, as it happens in *call3*.

In the rest of this subsection, we detail the reasoning behind the rules in each group.

Type/Context Manipulations. Rule (T) computes the relation Cast, e.g., type and context propagation paths over multiple casts. In our example code, this happens in lines 8–12 (see Figure 6.3) for which we already extracted the facts Cast$(0, 1)$ and Cast$(1, 2)$. Using these facts, rule (T) derives the fact Cast$(0, 2)$ (grey arrow in the figure), revealing the relation between *call1* and `f1`. Later, we will use this new fact to derive one of the target set results, namely TargetSet(*call1*, `f1`).

Rule (P) handles pointer casts and pointer aliasing. Whenever a pointer is cast to another pointer type, the two pointers *alias*, they point to the same value in memory. For us, this implicates that something can be referenced by two potentially distinct pointee types, so an implicit type or context propagation can happen. This propagation is possible in both directions, depending on which pointer is accessed afterward. In our example, this is in lines 14–19 (Figure 6.4). From this code, we extracted two PointsTo facts, one for the type `struct S *` in `scene2_a`, and one in `scene2_b`, plus a fact for context propagation Cast$(7, 8)$ between these two. Then we use rule (P) to derive Cast$(3, 9)$ and Cast$(9, 3)$.

Rules (S) and (U) handle struct and union types. When one struct is transferred into another, this implicitly transfers all its fields. By referencing the fields by their memory offset, we map fields to each other and collect newly-introduced Cast facts. We handle unions similarly. However, values from a union can only be read as the same type as they were written. We use the field type instead of the field byte offset for unions. In our example, this happens in lines 14–19 (Figure 6.4). Having derived Cast$(3, 9)$ with the rule (P) before, now we derive Cast$(4, 10)$ (and Cast$(5, 11)$) using rule (S). These new casts connect the function pointer "`&f2`" from line 16 with *call2*, as the result of computation of Cast relation.

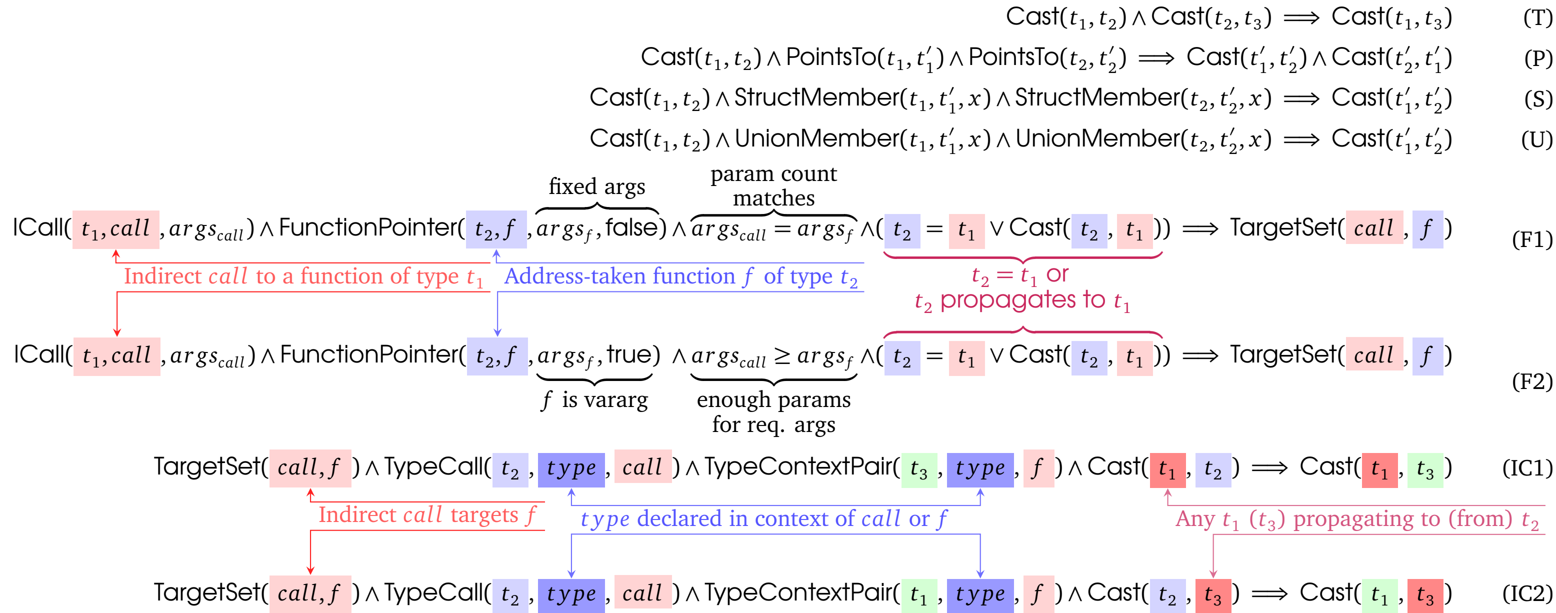


Figure 6.7: Rules for computing the final result with all possible function types for each call.

$$\text{ICall}(t_1, call, args_{call}) \wedge \text{FunctionPointer}(t_2, f, args_f, _) \wedge args_{call} = args_f \wedge t_2 = t_1 \implies \text{TargetSet}(call, f) \qquad \text{(ClangCFI)}$$

$$\text{ICall}(_, call, args_{call}) \wedge \text{FunctionPointer}(_, f, args_f, _) \wedge args_{call} = args_f \implies \text{TargetSet}(call, f) \qquad \text{(IFCC)}$$

Figure 6.8: Rules demonstrating the core of Clang CFI and IFCC computation of the final function types for each call.

Target Set Computation. Using the possible type propagation paths obtained from the Cast relation, the rules (F1) and (F2) compute the actual target set TargetSet. If a function pointer propagates to the callee argument of an indirect call, rules (F1) and (F2) add this function as a possible target. In addition, these rules check that the number of parameters is valid. Rule (F1) handles functions with a fixed number of parameters, which must match the number of arguments in the indirect call. Instead, while otherwise similar, rule (F2) handles functions with a variable number of parameters.

In our example, using the previously established facts and parts of derived Cast relation computation result, rule (F1) derives target sets for indirect calls `call1`, `call2`, and `call3`: using Cast(0, 2) – TargetSet(*call1*, `f1`), using Cast(6, 10) – TargetSet(*call2*, `f2`), and using Cast(13, 14) – TargetSet(*call3*, `set_callback`).

Indirect Call Context Transfer. Function calls propagate types by changing their contexts: the argument types propagate from caller to callee context, and the return type propagates from callee to caller context. Section 6.5.1 shows how to collect these propagations as Cast facts for direct calls, but for indirect calls, we only collected Cast facts from argument types to a TypeCall($n, t_a, call$) fact. Now that we have partial possible targets in TargetSet from the previous rules' application, we complete type propagation computation for indirect calls: We derive a type context propagation for every argument type from caller context to every possible target function's context and vice versa for return types.

Rules (IC1) and (IC2) calculate these context transfers from indirect call arguments to indirect call target parameters. This computation is based on a partial TargetSet result, and assumes that an indirect call actually calls all functions in its target set, which might be over-approximating. Rule (IC1) computes context transfers for indirect call arguments, while (IC2) does the same for return types. These rules derive each Cast as if an indirect call would be a direct call to each function from its target set. When generating facts for an indirect call e, we added Cast($N(t, c), N(t, e)$) for each argument, where $N(t, e)$ is also used in a TypeCall fact. From this fact, rule (IC1) derives Cast($N(t, c), N(t, f)$) for any function f the indirect call c can target, which is exactly what we would get for a direct call to f. Rule (IC2) does the same for return types.

These rules are necessary to handle higher-order functions (indirect calls with function pointer arguments), like `call4` in line 27, see Figure 6.5. As previously discussed, facts and derived Cast information allow us to obtain TargetSet(*call3*, `set_callback`) with rule (F1), yet we cannot derive a target set for *call4* yet. Still, we establish the fact Cast(15, 16) about the call argument of *call3*. Now applying (IC1) with these two facts, we get Cast(15, 18), connecting the call argument `f3` with the actual target `set_callback`. From Cast(15, 18), we use the other rules again (namely rule (T) for Cast and rule (F1)) to derive Cast(15, 17), Cast(15, 19), and a target set for `call4`: TargetSet(*call4*, `f3`). The final shape of the TargetSet relation is as follows: *call1* has `f1`, *call2* has `f2`, *call4* has `f3`, and *call3* has `set_callback` as their targets, respectively.

Multi-module Support. So far, we have focused on generating facts for and applying rules to a single module. To combine several modules when linking, we alpha-rename type/context pair identifiers in TypeContextPair and propagate these changes across all the other facts describing the module. Renaming preserves the identifier uniqueness required for the correct rules evaluation.

Dynamic Linking. We generate a *module summary* to support dynamic linking and loading

of code. It contains all facts from a program that might influence other modules. These facts are loaded with the code at runtime, combined, and the target sets are re-computed. Section 6.7 describes this process in more detail.

Supporting Other Forward CFI Approaches and Updates. The rule-based target set computation approach that we use for TyPro, is also general enough to support the encoding of other CFI schemes, facilitating their development and comparison. Moreover, the expressiveness of rule-based CFI encoding allowed us, for instance, to capture the core of the target set computation performed by Clang CFI [170] and IFCC [184] with only one rule for each of the approaches. In Figure 6.8, the rule (ClangCFI) compactly encodes that the scheme allows only the targets with the exact same types and number of arguments, while the rule (IFCC) relaxes the requirement of type matching. Obviously, for the example in Figure 6.1, the rule (ClangCFI) obtains empty results for all the calls, which is too restrictive and breaks the legitimate code. Rule (IFCC) is too permissive, resolving every call to all the functions as they all have only one argument. Section 6.9 details how the limitations of these approaches affect the security and stability of the programs.

Furthermore, TyPro's rule system supports updates, either as new rules or refinements, *without* changing other parts of the compiler.

6.6 Call Target Enforcement

After having computed the target sets of all indirect calls, we enforce the CFI policy at link time within a binary: indirect calls must transfer control only to the reduced set of targets.

To enforce that only functions from the computed targets in TargetSet can be called, we replace function pointers with function identifiers in the whole program. A function identifier is a unique number for each address-taken function. It has the same bit size as a pointer so that it can replace the function pointer in memory. This way, we must only alter its initialization and usage. We replace each use of an address-taken function that is not a direct call with this identifier. Next, we replace all indirect calls with a `switch-case` structure over the function identifiers. For each possible function in the target set, we generate a `case` matching the function ID and a direct call to the respective function in the body. In the default case, i.e., when the ID does not match any allowed function ID, we terminate the program to stop a detected attack. Figure 6.9 shows a simple program before and after transformation, in C language for simplicity.

To stay correct and precise, we must only allow functions that are actually valid call targets according to the C specification. First, the number of arguments must match, which is enforced by the argument number checks in rule (F1)—rule (F2) similarly handles functions with a variable number of arguments. Second, it must be possible to cast all arguments to the required type in the target function; otherwise, the call would be undefined behavior. Typecasts are usually possible if both types are an integer, pointer, or float, or have the same bit width. Last, if the call's return value is not discarded, it must be possible to cast the function's return value to the return value of the call expression. We check all these conditions before generating cases and confirm their validity experimentally in Section 6.9.1. Still, functions with incompatible types might occur due to overapproximation in our analysis.

Generating assembly code from `switch-case` statements is left to LLVM, which lowers them into different assembly constructs, from simple comparisons over binary trees to

```
1 typedef void (*fptr_t)(long);
2 void f1(long a) {}   // ID assignment: &f1=>3
3 void f2(long a) {}   // ID assignment: &f2=>4
4
5 void code_before_typro(bool use_alternative, long param) {
6     // take addresses of functions
7     fptr_t fp = use_alternative ? &f1 : &f2;
8     // call function addresses
9     (*fp)(param);
10 }
11
12 void code_after_typro(bool use_alternative, long param) {
13     // take IDs instead of function addresses
14     fptr_t fp = use_alternative ? (fptr_t) 3 : (fptr_t) 4;
15     // direct call based on IDs
16     switch ((long) fp) {
17     case 3:
18         f1(param);
19         break;
20     case 4:
21         f2(param);
22         break;
23     default:
24         abort();
25     }
26 }
```

Figure 6.9: Example showing a simple indirect call before and after transformation.

jumptables. Also, LLVM can run additional optimizations over the new statements, that might not be possible before our changes, e.g., direct calls can be inlined if the callee is only short or rarely used.

To improve performance, TyPro tries to assign ascending identifiers to functions that occur together in the same target set. Ascending numbers lead to dense identifier sets, which can be converted to efficient jumptables, facilitating the subsequent LLVM optimization passes. Furthermore, TyPro reserves the lower numbers (up to 3) for program-specific non-function constants (like `SIG_IGN` in libc).

For dynamic modules, as detailed in Section 6.7.3, a linked library generates function IDs and new switches at runtime, based on the updated analysis results.

6.7 Dynamic Modules

In contrast to many prior CFI schemes [170, 63, 74, 73, 118], TyPro can handle code loaded at runtime: dynamically-linked libraries or runtime loading of shared libraries. To this end, we require that the loaded modules are also protected by TyPro. Furthermore, a generic runtime library must be present (see Section 6.4.3). It combines type information from different modules at runtime, computing new target sets for indirect calls and updating the necessary checks. Thus, we have to export type information with every program and

ExternalSymbol : $\mathbb{S}$	(exported definitions)
InterfaceType : $\mathbb{S} \times \mathbb{N}$	(types per declaration)
External : $\mathbb{N}$	(pairs in module summary)

$$\text{ExternalSymbol}(f) \wedge \text{InterfaceType}(f, t_1) \implies \text{External}(t_1) \quad \text{(ES)}$$
$$\text{External}(t_1) \wedge \text{PointsTo}(t_1, t_2) \implies \text{External}(t_2) \quad \text{(E1)}$$
$$\text{External}(t_1) \wedge \text{StructMember}(t_1, t_2, _) \implies \text{External}(t_2) \quad \text{(E2)}$$
$$\text{External}(t_1) \wedge \text{UnionMember}(t_1, t_2, _) \implies \text{External}(t_2) \quad \text{(E3)}$$
$$\text{External}(t_1) \wedge \text{Cast}(t_1, t_2) \wedge \text{ICall}(t_2, _, _) \implies \text{External}(t_2) \quad \text{(EC)}$$
$$\text{External}(t_1) \wedge \text{ICall}(t_1, call, _) \wedge \text{TypeCall}(t_2, _, call) \implies \text{External}(t_2) \quad \text{(ECA)}$$
$$\text{FunctionPointer}(t_1, _, _, _) \wedge \text{Cast}(t_1, t_2) \wedge \text{External}(t_2) \implies \text{External}(t_1) \quad \text{(EFP)}$$
$$\text{FunctionPointer}(t_1, f, _, _) \wedge \text{External}(t_1) \wedge \text{InterfaceType}(f, t_2) \implies \text{External}(t_2) \quad \text{(EFI)}$$

Figure 6.10: Additional predicate definitions for dynamic module support, and additional rules for module summaries.

shared object. Therefore, we extend the target set analysis in Section 6.5 by a *module summary* which contains only the type/context propagation information that can influence the computation of other modules.

To compute a module summary, we add to our analysis the new relations and rules presented in Figure 6.10. The final result is established after computing the External relation, which intuitively contains all types that can be propagated to or from outside of the module. This relation is the index set of type/context and type/call pairs that must be exported in the module summary, if another module could contain the same pairs. For example, the summary includes the parameters of an exported function that generate the same type/context pair in every module it is imported to. The summary is the subset of all relations except TargetSet, containing *only* facts that refer to indices in the External relation.

6.7.1 Additional Input Generation

To determine which type/context pairs enter the module summary, we need additional facts collected from the source code, filling the relations ExternalSymbol and InterfaceType. They describe the C interface that a module exposes to other modules, containing similar information like header files in C. We iterate all declared symbols of a module, including both imported and exported symbols, and record their interface types. If the symbol is a global g of type t, we record a fact InterfaceType$(g, N(t, g))$. If the symbol is a function f with signature $rt\ f(a_1, \ldots, a_m)$, we record facts InterfaceType$(f, N(a_i, f))$ and InterfaceType$(f, N(rt, f))$. If the symbol is visible after linking, we add it to the relation, as ExternalSymbol(f) or ExternalSymbol(g).

6.7.2 Additional Type Analysis

Using additional input facts described in Section 6.7.1, we compute the set of pair indices that must be visible to other modules. This information is expressed with External relation.

It is computed using the rules shown in Figure 6.10 including the rule for External symbols (ES), a group of rules for type visibility ((E1), (E2), and (E3)), a group of rules for indirect calls' treatment (they are (EC) and (ECA)), and, finally, a group of rules for function pointers that should be referenced in the summary (namely, (EFP) and (EFI)). In the following, we explain these rules in more detail.

6.7.2.1 Initialization

The interface of imported or exported symbols must be external because other modules can have the same symbols and, therefore, facts about the same pairs in their relations. We capture this property of the imported and exported symbols with the help of (ES) rule.

6.7.2.2 Type Visibility

When a type is visible externally, its structure is also visible. If an external type is a pointer type, rule (E1) marks its pointee type as external. For struct and union types, rules (E2) and (E3) respectively mark their fields as external.

6.7.2.3 Indirect Calls

If there is a type transfer between an external entry and an indirect call, this indirect call could receive function pointers from other modules. Rule (EC) marks this call as external, so we include its type/context pair in the module summary. In addition, rule (ECA) marks the corresponding TypeCall pair as external, including them in the module summary.

6.7.2.4 Function Pointers

Facts referencing a function pointer are considered external if there is a type transfer to any external fact (as captured by rule (EFP)) because the function pointer could be transferred to another module at runtime. The pointed function could become accessible at runtime, even if its symbol is not exported; therefore, we have to include its interface in the module summary: Rule (EFI) marks all arguments' types and the return value types as external.

After running the computation, External references the type/context pairs necessary for runtime TargetSet computations. For the module summary, we filter the entire fact set to contain only those pairs. Exporting only filtered facts greatly reduces the file size of the summary and the runtime of target set computations during dynamic loading.

6.7.3 Dynamic Call Target Enforcement

Our approach and its enforcer also support dynamic linking, which requires additional processing. Using the information obtained by the analysis described in Section 6.7.1, we can see at link time which target sets might need expansion later. If an indirect call has an associated index in External, there might be valid targets from other modules at runtime. When building the switch for such a call at link time, we do not add an error handler to its default case but add a new direct call, which we call the *trampoline*. While the trampoline target defaults to the error handler, the runtime library can overwrite it if necessary. If

the target sets are amended at runtime, the trampoline target will point to a new switch, handling the additional targets.

At runtime, the third component of TyPro, the runtime library, updates the target sets of all external indirect calls if necessary. The runtime library is a small C++ library that contains the presented algorithms and a custom just-in-time compiler. When modules are loaded during startup or at runtime, the runtime library loads their module summary (provided by the analysis discussed in Section 6.7.1) from a read-only section. The module summaries from all loaded modules are combined into a single set of facts. If necessary, the runtime library will re-run the target set computation from Section 6.5.2 on the collected module summary and will eventually add new targets to the existing target sets. If new targets appear in the target set of an indirect call, the runtime library will generate a new switch-case statement. It just-in-time compiles a new function with a switch over the function identifier and one case for each new target. Finally, the trampoline of the indirect calls is assigned to this newly generated function. If a module now calls a function with an identifier assigned by another module, the newly generated switch will dispatch the desired function, without allowing attackers to execute the arbitrary functions.

We designed the runtime library with security against memory corruption in mind. An attacker might try interfering with the module summaries or the target set computation to weaken TyPro's protection. To prevent these attacks, the runtime library exclusively uses memory from a custom, protected heap. This heap is isolated from the remaining program; no memory or pointers are shared with the main program. The heap is read-only by default and is only writeable during analysis and JIT compilation. An attacker can only interfere at the exact point when a library is loaded in a different thread and only if they leak a pointer to the heap.

An attacker might also try to tamper with the just-in-time compilation to inject custom code into the JITing area. The runtime library avoids this risk by compiling twice: A fresh part of the JIT area is made writeable, but not executable for the first compilation. Afterward, the generated code is made readonly. In a second pass, the JIT checks that the memory contains the expected instructions only, detecting any attempted attack on the JIT area. The generated code finally gets executable if the checks succeed. The runtime overhead of the second compilation pass is negligible compared to the analysis time. With these two defenses, we are confident that the runtime library does not introduce new security risks.

6.8 Implementation

We build the TyPro prototype as an extension of the Clang/LLVM 10 compiler toolchain. It targets 64-bit x86, ARM, and MIPS, generating binaries without additional dependencies on hardware features or operating system. TyPro can produce fully protected binaries with a protected musl libc [120], or protected binaries linking against an unprotected GNU libc (see Section 6.8.1 for details).

In our prototype, we instrument the code generation of Clang to collect facts (as discussed in Section 6.5.1 and Section 6.7.1) along with the compilation of the program. The collected facts are stored together with the original LLVM IR code in an object file. The generated IR itself is not altered.

After applying several optimization steps outlined in Section 6.8.2, we extract the facts

from all IR files seen in our modified version of LLVM's linker `lld`. Finally, we encode the optimized set of facts and the rules (from Section 6.5.2 and Section 6.7.2) in datalog for the Soufflé Logic Solver [71, 149] which we leverage to compute the TargetSet relation containing the targets for each indirect call.

We implemented a runtime library that enforces dynamically-loaded targets as a standalone C++ library without dependencies on LLVM or other non-standard libraries. It loads the serialized facts from multiple modules, runs the target set computation, and generates switches with a built-in just-in-time compiler. We perform all operations lazily, i.e., only after the first indirect cross-module call, preventing unnecessary computation. The library is 1.3 MB large and can either be shared or included in a protected musl libc.

6.8.1 C Standard Libraries

As mentioned previously, our approach requires all dynamically linked libraries to go through the same processing as the program using it, i.e., the analyses discussed in Section 6.5 and Section 6.7. However, many related works [170, 63, 74, 73, 118, 184] exclude the C standard library, for the following reasons: The standard GNU libc is not compilable by the Clang compiler, contains plenty of (typeless) inline assembly, and communicates with the Linux kernel over a syscall interface that cannot be altered.

Instead of excluding the standard library, TyPro uses and protects musl libc [120], which is compatible with Clang. Only when functions are sent to the kernel (signal handlers), TyPro resolves the identifiers back to function pointers before transferring them. A protected musl libc can be statically linked or used as a regular shared library.

Alternatively, programs relying on the GNU standard library can optionally link against an unprotected libc. In this case, TyPro resolves identifiers back to function pointers before transferring them to the libc, avoiding compatibility issues. To resolve a function identifier, a switch-case construct is emitted (similar to the one described in Section 6.6), returning actual function addresses instead of direct calls. With this method, TyPro-protected programs can use the system's unmodified standard library without breaking compatibility. This method could also be used to link with other *unprotected* libraries, assuming it is known *a priori* which library will be unprotected.

6.8.2 Optimizations

After the facts' extraction, we perform some minor but crucial optimizations by omitting unnecessary facts for the final target computations. These optimizations do not change the result of the computation but are essential for reasonable performance. First, we only collect a TypeContextPair fact if its identifier is used in at least one another relation. If we omit a type, we also do not generate additional facts relevant to this type definition, e.g., PointsTo or StructMember facts. TypeContextPair facts unused in other relations cannot be used in any rule and are therefore irrelevant for target set computation. Second, we omit primitive C types that are smaller (in bits) than a pointer, e.g., `char` or `void`. It is impossible to convert a function pointer to or from these types, nor can they participate in pointer aliasing or other rule-covered C structs; they are therefore irrelevant for target set computation. The most prominent example is the type `void`, which will never appear in our fact set, in contrast to `void*`, which has the same size as a function pointer and will appear in facts.

Third, we collapse chains of direct casts, e.g., the expression "`(fptr_int) ((void*) &f1)`" will be seen as one cast from `fptr_long` to `fptr_int`.

Running the computation on fact sets of larger programs is very time-consuming, in particular when the facts of all input files are merged. Therefore, we use an optimization based on *equivalences* to reduce the input size for the datalog solver drastically. If two TypeContextPair facts are *equivalent*, these type/context pairs can be merged without changing the result of the target set computation, making the input fact set smaller and the computation faster. In our implementation, two simple patterns indicate equivalence:

- If we extracted facts Cast(n_1, n_2) and Cast(n_2, n_1), then type/context pairs n_1 and n_2 can be merged.
- If we extracted Cast(n_1, n_2) facts and both type/context pairs n_1 and n_2 are pointer type (i.e., we extracted also PointsTo(n_1, n'_1) and PointsTo(n_2, n'_2) facts), then type/-context pairs n'_1 and n'_2 can be merged.

Merging propagates along with the structure of its types: if two facts of pointer type merge, their referenced type's facts also merge. And if two facts of struct or union type merge, their field facts also merge. We found this optimization to improve the computation runtime considerably while not changing the result of the target set computation.

6.9 Evaluation

We evaluate TyPro using three criteria. In Section 6.9.1, we measure *correctness*. Programs must not break, i.e., target sets must always include (at least) the correct targets. In Section 6.9.2, we evaluate *security*. Security is largely determined by the extent to which attack surface is reduced, i.e., how many unused targets are still included in the target set. Finally, in Section 6.9.3, we measure TyPro's *performance* in terms of runtime slowdown and size overhead.

Evaluation datasets. We evaluate each criterion on two datasets. First, we use the well-known SPEC CPU 2006 benchmark suite [58, 161]. In particular, we consider all nine SPEC programs written in pure C and use indirect calls, namely, *bzip2*, *gcc*, *gobmk*, *h264ref*, *hmmer*, *milc*, *perlbench*, *sjeng*, and *sphinx3*. These SPEC programs are also used in most related work [34, 211, 184, 51, 73, 98, 85, 63, 74, 129, 186, 45, 131], which allows for an easy comparison with TyPro. In contrast, the newer SPEC CPU 2017 [160] hasn't been used in any related work so far.

Our second dataset consists of 7 larger real-world programs, commonly used in related work [45, 51, 63, 73, 74, 85, 98, 186]. It includes the web servers *Apache*, *lighttpd* and *nginx*, the FTP servers *pureftpd* and *vsftpd*, a cache server *memcached*, and the database *redis*. These programs use libraries like *libpcre*, *zlib*, *libevent*, *lua* and more, that we also compiled with our protection.

Compilation setup. We compile all programs and their respective libraries with TyPro enabled, using full optimization (`-O3`) and having link-time optimization enabled. If possible, we preferred static linking, so, when compared, no disadvantage for related work without dynamic linking support (like Clang CFI) was introduced.

For comparison, we compile all programs using the same optimization settings with an unmodified Clang 10 compiler, default libraries, and Clang CFI [170] in normal mode (`-fsanitize=cfi-icall`) and generalized mode (`-fsanitize=cfi-icall` and `-fsanitize-cfi-icall-generalize-pointers`). Moreover, we also compared TYPRO to IFCC [184] and simulate CFGuard's [101] policy.

6.9.1 Correctness

A CFI scheme must not break existing code to foster adoption in practice. While manual code changes could resolve incompatibilities, this is not feasible in many cases.
SPEC. We run the SPEC CPU benchmarks with all input sets and verify the output. TYPRO handles all nine programs without any failure, as shown in Table 6.1. In contrast, Clang CFI causes crashes in *gcc* and *hmmer* computing too narrow target sets, which we verified manually. These two benchmarks failed even in Clang CFI's generalized mode with relaxed type checking. These failures were observed only post-mortem as crashes, and it might be hardly feasible to alter the source of such big projects as *gcc* to respect the typing judgments. From the related work [85], we also know that MCFI [129] crashes on *perlbench* and *gcc*, while IFCC [184] passes all benches. The lack of dynamic linking does not impact related work here, as SPEC does not use it.
Real-world. TYPRO successfully compiled all seven programs without introducing errors, as shown in Table 6.2 (i.e., no false negatives). Also, IFCC [184] passed all tests but with much larger target sets, which we address in Section 6.9.2. But Clang CFI failed on four programs (*lighttpd*, *nginx*, *pureftpd*, and *redis*). Its generalized mode resolved the errors in *nginx*, but not the remaining three. We checked all introduced problems by hand—dynamic linking caused none, and all were due to functions that do not precisely match call types. Clearly, Clang CFI is too restrictive; and any extension to support *vararg* would break Clang CFI's equivalence class model. This in turn would break Clang CFI's target set enforcement, which relies on equivalence classes.
Unit Testing. To further test TYPRO, we created a set of over 220 hand-crafted unit tests, checking the correctness of TYPRO's support for different aspects of the C language, triggering corner cases, and checking correct interaction with the C standard library. We verified using QEMU that these tests also succeed on ARM and MIPS. With these tests and the various applications, we are confident that TYPRO can handle any standard-conformant C program (within limits discussed in Section 6.10).

6.9.2 Security

As detailed in Section 6.6, TYPRO takes all indirect calls in a C program and converts them to a set of well-typed direct calls. Therefore, no indirect call instructions remain in the compiled program, making arbitrary jumps impossible. However, attackers with memory corruption capabilities can tamper with the function identifiers that replaced function pointers in memory. Even in this case, they cannot invoke arbitrary code—only the execution of a minimal and limited set of functions is possible. Furthermore, the generated code is inherently safe against concurrent modifications and potential time-of-check/time-of-use (TOCTOU) vulnerabilities for two reasons: First, during computation, no intermediate values are spilled onto the stack. The branching happens in registers that are inaccessible to

Table 6.1: Average number of call targets per indirect call on SPEC. TyPro compared to Clang CFI, IFCC, CFGuard and MCFI. ✗: benchmark fails (CFI too restrictive). ○: best security per benchmark. ●: best security among compatibility-preserving schemes.

	Clang CFI [170]	Clang CFI (generalized)	TYPRO	IFCC [184]	CFGuard [101]	Clang CFI (data: [85])	MCFI [129] (data: [85])
400.perlbench	17.71 ○	51.99	22.32 ●	180.79	821.00	22.03	23.27 ✗
401.bzip2	1.00 ○	1.00 ○	1.00 ●	1.00 ●	2.00	1.00 ○	1.00 ○
403.gcc	9.19 ✗	34.44 ✗	24.98 ●	365.12	1192.00	8.91 ✗	32.63 ✗
433.milc	2.00 ○	2.00 ○	2.00 ●	2.00 ●	2.00 ●	2.00 ○	2.00 ○
445.gobmk	631.50	631.50	631.50 ●	749.12	1786.00	600.84 ○	605.51
456.hmmer	9.00 ✗	18.00 ✗	2.78 ●	19.00	19.00	10.00	10.00
458.sjeng	7.00 ○	7.00 ○	7.00 ●	7.00 ●	7.00 ●	7.00 ○	7.00 ○
464.h264ref	2.24	2.34	2.24 ●	10.95	42.00	2.06 ○	2.06 ○
482.sphinx3	5.00 ○	5.00 ○	5.00 ●	5.00 ●	5.00 ●	5.00 ○	5.00 ○
avg. %	(base)	+41.7%	+0.6%	+157.5%	+379.8%	+1.8%	+18.4%

Table 6.2: Average number of call targets per indirect call on various real-world server applications.

	Clang CFI	Clang CFI (gen.)	TYPRO	IFCC	CFGuard
httpd	14.69 ○	41.78	36.19 ●	462.59	2267.00
lighttpd	5.99 ✗	10.93 ✗	11.26 ●	50.32	257.00
memcached	1.99 ○	2.36	2.01 ●	14.88	85.00
nginx	16.53 ✗	56.08 ○	102.28 ●	240.72	758.00
pureftpd	1.00 ✗	1.00 ✗	1.00 ●	3.00	15.00
redis	10.03 ✗	44.19 ✗	48.06 ●	247.50	1136.00
vsftpd	3.33 ○	3.33 ○	3.33 ●	6.00	35.00
avg. %	(base)	+90.8%	+102.3%	+772.3%	+4102.6%

attackers. Second, even if manipulation would be possible, there is no arbitrary call that an attacker could try to reach, only direct calls for each function in the target set. The only indirect jumps in the program come from the compiler itself, as introduced for jumptables. However, the compiler properly bound-checks these jumps, making them irrelevant for control flow security. Thus, our switch-based target enforcement indeed limits the surface for code-reuse attacks.

The same arguments hold for the target checks between dynamic modules. Both statically-compiled and JIT-compiled switch statements are safe on their own. They are connected by a pointer in read-only memory, which attackers cannot tamper with. Only when new switches are built during load time the pointer is temporarily made writeable. However, dynamic modules are often loaded at startup before any input is processed to minimize the risk further.

Finally, we aim to understand the security benefits of finding minimal call target sets. Smaller target sets leave the attacker with fewer choices and fewer gadgets. As suggested in related literature [18, 85], we report absolute numbers of possible targets for indirect calls. We rely on CSCAN [85]'s metric of computing the average number of targets per indirect call,

considering only indirect calls that are reached during the execution of the program. Because CSCAN's measurement approach is incompatible with our function identifier and direct call approach, and because CSCAN showed problems with different compiler optimizations, we implemented an equivalent computation for this metric, collecting target sets at compile-time and indirect calls at runtime. We evaluate TyPro and compare it to Clang CFI [170] in both normal and generalized mode, IFCC [184] and CFGuard's [101] policy, all with the same compiler version and optimization settings. We do not compare to Intel CET's indirect branch tracking [136], whose precision is likely worse than CFGuard for programs without JIT compilers. Because Clang CFI and IFCC do not support musl libc, we link all programs against an unprotected GNU libc for this comparison, in line with related work. Results with musl libc are similar.

SPEC. Table 6.1 reports the average number of targets for SPEC programs. Clang CFI builds the smallest target sets but breaks programs. In contrast, TyPro has slightly more targets (0.6% on geometric mean) but keeps all programs intact without modifications. Moreover, TyPro builds smaller target sets than Clang CFI's generalized mode, which still breaks programs at a worse precision. IFCC and CFGuard do not break anything but build larger target sets than TyPro, on average 2.5× / 4.8× the size. We also compare our results to MCFI [129], analyzed by CSCAN [85]. TyPro has a slightly better precision than MCFI, and its policy works even on unmodified programs. Note that CSCAN used a different compiler version and likely different, unknown optimization settings for their results. Furthermore, it is unclear if it uses patches to SPEC. For transparency, we also show CSCAN's results for Clang CFI, which vary from us by 1.8%. If linked with musl libc, the programs use 1–15 additional indirect calls from musl, with 2–17 average targets. On average, over all programs, calls inside musl have 7 targets.

Real-world. Table 6.2 reports the average number of targets on several real-world programs frequently used in related literature. Again, Clang CFI builds the smallest but often incorrect target sets. Clang's generalized mode builds 91% larger but still too narrow target sets. TyPro has more allowed targets on these larger programs than Clang (+102%, +6% more than generalized mode), but the target sets do not cause crashes. Compared to other correct solutions like IFCC or CFGuard, the number of possible targets has greatly reduced—TyPro has less than one-quarter of IFCC's targets and 4.8% of CFGuard's targets while still keeping programs intact.

Having said this, TyPro's target set computation is still an approximation working exclusively on the extracted type information. The algorithm can overapproximate the possible targets to prefer correctness and speed over precision but never under-approximates targets. In specific cases, sophisticated attacks on really large programs like Control Jujutsu [38] or Control-Flow Bending [18] might still be possible, even in the presence of a perfect CFI policy. But from the experiments, we conclude that TyPro hits a sweet spot between correctness and security.

6.9.3 Performance

To facilitate wide deployment, we now demonstrate that TyPro does not impose significant overheads on protected programs.

Performance Overhead. We used the SPEC CPU 2006 benchmark and our real-world

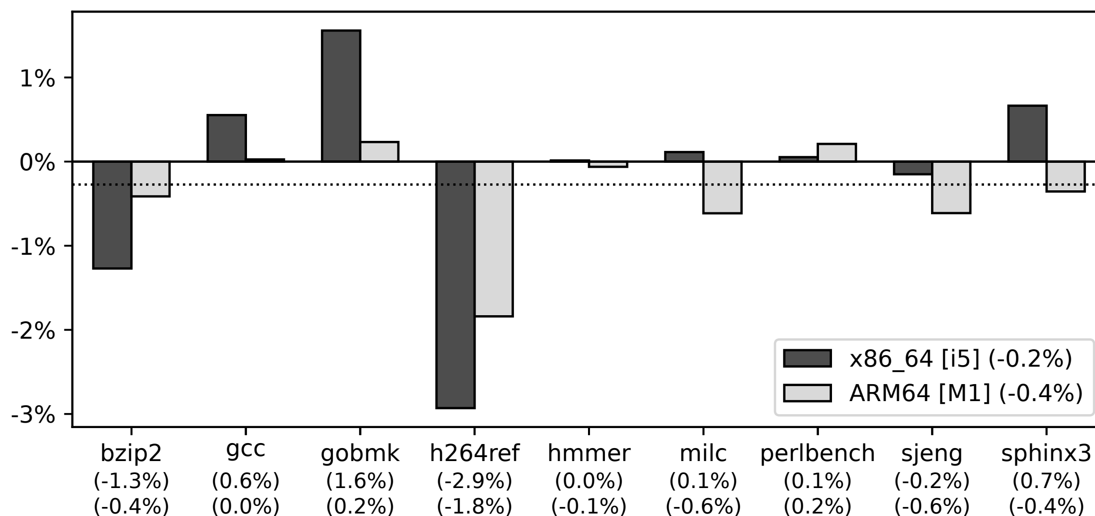


Figure 6.11: Runtime overhead of TyPro on SPEC 2006 benchmarks. Average overhead is −0.3% (i.e. programs get faster).

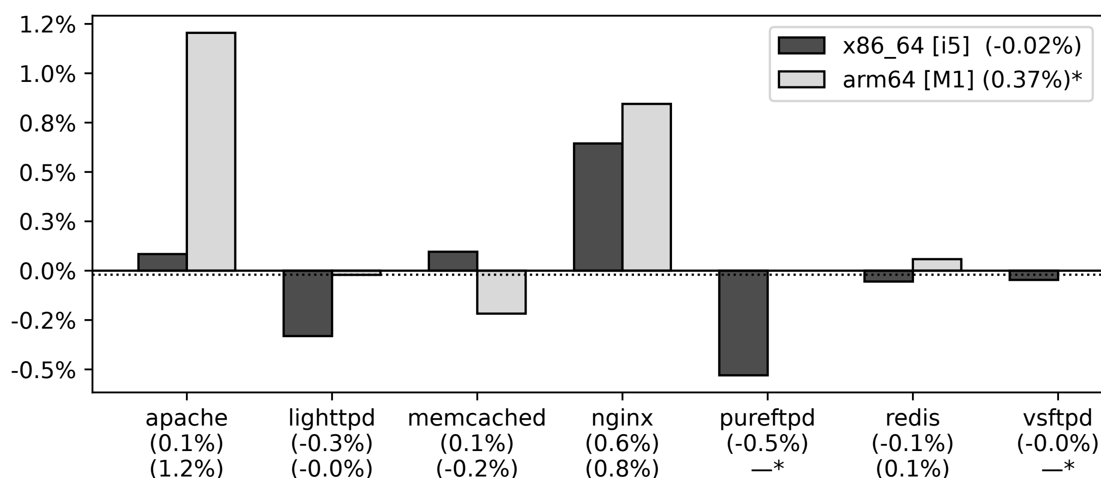


Figure 6.12: Runtime overhead on real-world applications. Average overhead is zero. (*) Unreliable FTP benchmarks were excluded on ARM.

servers to evaluate the performance impact. We measured the server's performance with ab [165] (webservers), memaslap [212] (memcached), redis-benchmark [145] (redis) and ftpbench from pureftpdlib [146] (FTP servers). Experiments ran on a Dell workstation with Intel Core i5-4690 CPU (4 × 3.5 GHz, no hyperthreading) and 32 GB of RAM. The operating system was Debian 10 "Buster" with kernel 4.19. To measure SPEC's performance on 64-bit ARM, we used an Apple M1 chip, 16 GB Ram, and asahi Linux with kernel 5.17. We used the "performance" CPU governor, disabled CPU boost, and applied cpuset to minimize the impact of environment and operating system on the measurements. We repeated experiments at least 10×. The standard deviation on SPEC was at most 0.76%, and 0.2% on average. Unfortunately, we had to exclude two inconsistent benchmarks (FTP servers on ARM):

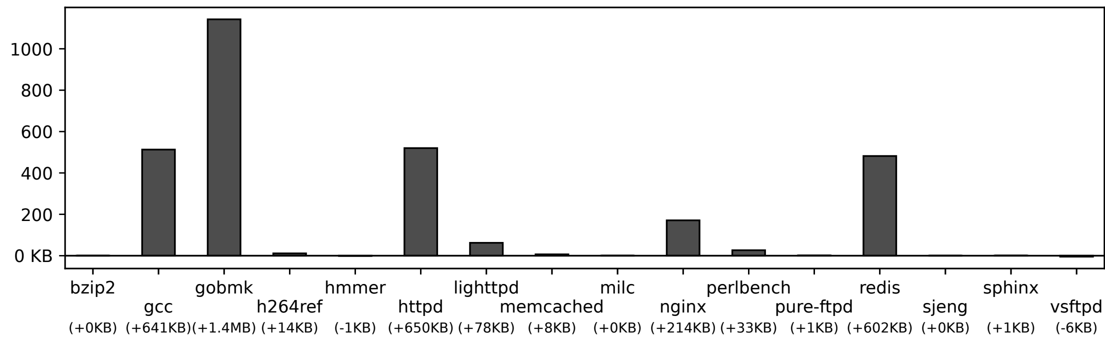


Figure 6.13: Additional size of SPEC and other example programs on x86 (in KB).

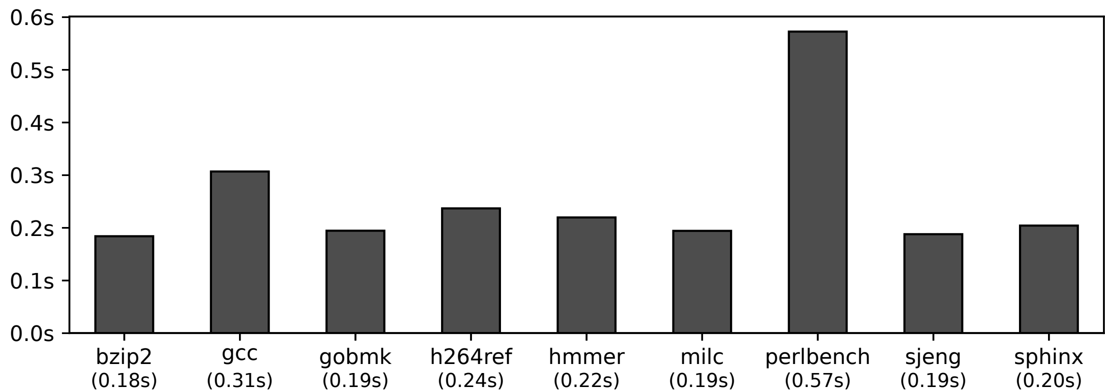


Figure 6.14: Runtime computation time for dynamic linking against musl libc.

averaged results varied between ±0.5% with a standard derivation of up to 3%. Lacking hardware, we did not evaluate performance on MIPS.

We present our findings in Figure 6.11 and Figure 6.12. Protected programs get between 1.6% slower and 2.9% faster. The mean overhead is −0.3% on SPEC, i.e., programs get slightly faster, and zero on real-world applications. TYPRO has a higher overhead in programs with large target sets (like *gobmk* with >600 valid targets, or nginx) and shows negative overhead in programs with small target sets (like *bzip* and *h264ref*). For many programs, the measured changes are within the standard deviation, and no real overhead is measurable.

Space Overhead. When comparing program size, we observed that the compiled binaries sometimes get larger. The generated switches and direct calls need more instructions than a single indirect call. Most programs are a few kilobytes larger after protection. In the worst case, gobmk gets 1.4 MB larger (40%). Figure 6.13 shows the additional binary size of all tested programs. On average, programs get 9.1% larger. We expect that this space demand does not prevent a CFI scheme from broader adoption, except if resources are scarce, such as for embedded systems.

6.9.4 Dynamic Loading

Dynamic loading support is an important feature for any CFI system such as TYPRO. All SPEC programs can dynamically link against a protected standard library (musl libc). At runtime,

computation and just-in-time compilation take between 0.18s (bzip2) and 0.57s (perlbench), with 0.25s on average. Target set computation and JIT compilation happens lazily—only when compile-time dispatchers can't handle a function ID, the runtime computation is triggered. Therefore most SPEC benchmarks can avoid runtime computation completely because they do not exchange function pointers with the C standard library. Figure 6.14 shows the maximal runtime computation, with lazy evaluation disabled.

In particular, *redis* nicely demonstrates TYPRO's dynamic loading capabilities: it has a `MODULE LOAD` command that can load arbitrary shared objects at runtime. We tested this command on a protected redis instance with various protected modules and verified that they load and are usable. Even though redis had one of the largest fact sets in our tests, recomputing the target sets after a module load was a matter of seconds. Given that target sets can be cached and the JIT compiler's runtime is negligible, we consider this support and its performance practical.

6.10 Limitations & Discussion

We now discuss the limitations of the current TYPRO prototype.

First, the target set computation slows down compilation. Our non-parallelized current prototype finishes the analyses in a few seconds for small programs, in a few minutes for medium programs, and less than an hour for exceptionally complex programs like nginx. While we believe that this build time is not necessarily a deal-breaker in times when software is built by CI/CD servers, our prototype can still be improved for speed. Likewise, dynamic loading and re-computing target sets at runtime can take up to a few seconds. Assuming that the dynamically-loaded modules do not change frequently, TYPRO could cache the computed target sets to speed up this process massively.

Second, like other CFI systems, our approach has limited compatibility with unprotected libraries. As soon as function pointers are exchanged with other libraries, these libraries have to be also protected. Usually, one could recompile these libraries with protection, but there might be cases where recompilation is impossible. In fact, TYPRO already has semi-automated support for function pointers exchanged with unprotected libraries. But developers would have to mark functions imported from unprotected modules so that the compiler can instrument calls accordingly.

Third, TYPRO does not yet support inline assembly or C++, which is frequently combined with C code. We believe relevant use of inline assembly is rare, apart from standard libraries. Most occurrences do not need analysis, e.g., the assembly in musl libc. While some parts of C++ are already supported (like lambda functions), we would need extensions for full support: First, we would have to collect class layout and inheritance information during type and context collection. Second, we would need additional rules for type propagation over inheritance. Third, we would compute target sets for C++ virtual dispatch. Fourth, we would rewrite virtual dispatch similar to NOVT [P1].

6.11 Related Work

TyPro positions itself in a wide range of existing CFI systems. Our evaluation has demonstrated that TyPro stands out as the most precise software-only forward CFI system that retains compatibility even with large programs. TyPro is fully open source and can protect any C software that LLVM can compile without requiring code modifications or special hardware support. Furthermore, we believe the proposed function type propagation is novel at the conceptual level. In the following, we will survey related work and briefly mention how TyPro differs from these proposals.

Software-Only Forward CFI. So far, Clang is the only compiler that has a strong forward CFI solution built-in. Clang CFI [170] checks a target function's type at indirect calls, preventing type-mismatched functions from being called. MCFI [129] is a similar protection with support for dynamic loading and linking. πCFI [131] extends MCFI with runtime information; functions are not allowed to be targeted before they are referenced once at runtime. IFCC [184] checks not function types but argument numbers, providing compatibility with legacy programs at the price of much larger target sets. We have demonstrated that these schemes are either too restrictive and break programs or are too permissive. Finally, Microsoft Control Flow Guard [101] is an even weaker protection: an indirect call can target the start of any function. All these systems are context-insensitive, i.e., they consider only the indirect call and function pointer at runtime, similar to TyPro.

Hardware-Assisted CFI. To accelerate CFI, researchers recently proposed hardware-assisted CFI schemes. In particular, Intel PT [66] was used in PT-CFI [51], PITTYPAT [34], CFIMON [202], μCFI [63], PathArmor [186] and GRIFFIN [45]. Furthermore, researchers proposed TSX-based CFI [118] and CFI-LB [73] based on Intel's transactional memory extensions, while OS-CFI [74] combines Intel MPX and TSX. CCFI [98] uses Intel's AES-NI instructions to perform cryptographic CFI checks. Finally, Intel CET [155, 136] contains a hardware-based, fast, but very imprecise forward CFI scheme. In contrast to TyPro, these systems require special hardware and potentially changes to the OS kernel, hindering the deployment of protected applications in many settings. Some of these systems are context-sensitive, i.e., they consider additional runtime information to determine correct call targets at the cost of increasing complexity and performance penalties.

Binary-Only CFI. Protecting pre-compiled executables resolves the dependency on source code. In particular, Opaque CFI [112], CCFIR [210], binCFI [211], Lockdown [138], TypeArmor [187] and CFIMon [202] enforce a CFI scheme without available source code. In contrast to source-based schemes, these CFI solutions lack precise typing and flow information. They rely on approximative reconstructions, making them less precise.

CFI for JITed Code. There are special-purpose CFI schemes targeting dynamically generated code. RockJIT [130], JITScope [207] and DCG [159] protect the execution of JIT-compiled code against control flow attacks. These schemes nicely extend TyPro because our current prototype only covers C code at compile time.

CFI Evaluations. Next to CFI schemes, researchers proposed frameworks that evaluate and compare CFI schemes. CSCAN [85] analyses CFI-protected binaries, counts the number of possible targets for each indirect call, and checks the added runtime checks for weaknesses. When we evaluated TyPro, we compared it with CSCAN-provided results. Similarly, the LLVM-CFI framework [119] is a toolkit to evaluate different policies inside LLVM. A

survey [16] by Burow et al. compares CFI schemes without requiring a specific compiler.
Type Analysis. Recently, Multi-Layer Type Analysis [95] (MLTA) has been proposed—a type analysis system designed to improve the precision of a simple base analysis (like the one used in Clang CFI). In contrast to TYPRO, MLTA relies on a base analysis and can only reduce the computed target sets of this base analysis. Fixing errors coming from the inaccurate base analysis, our main contribution, is out of scope for MLTA. Older work [104] uses an inexpensive and imprecise analysis to build a call graph for code browsing tools. Recent tools [105, 78] use local type information and casts near memory allocations to determine types of heap objects, countering memory reuse and information leakage vulnerabilities.

6.12 Conclusion

We presented TYPRO, a forward CFI scheme for C programs. TYPRO protects indirect calls in legacy, real-world programs without requiring manual effort. Even multi-module programs, dynamically loaded at runtime, can be protected. TYPRO's type-based approach has a precision comparable to state-of-the-art solutions used in production-grade compilers but does not underapproximate indirect call targets, leaving all protected programs intact. TYPRO successfully targets the sweet spot between security and compatibility. On average, TYPRO does not impose any performance overhead and only moderate binary size increase. Consequently, TYPRO enables CFI deployment for legacy and modern real-world applications.

6.12.1 Future Work—Switchpoline

Together with NOVT from Chapter 5, TYPRO served as the basis of Switchpoline [S1]. Switchpoline is the first automated, software-based Spectre-BTB mitigation for ARMv8 CPUs, where the typical x86-based defenses like retpoline do not suffice.

On ARM, every indirect forward control flow transfer is potentially vulnerable to Spectre-BTB. Switchpoline is a compiler that builds applications without *any* indirect jump instruction. With their precondition invalidated, Spectre-BTB attacks can no longer occur. To this end, Switchpoline builds on TYPRO's enforcement and NOVT, which can rewrite any language-level indirect jump into a series of direct jumps. Switchpoline further extends this approach and rewrites all possible indirect jumps into direct ones, either at compile-time or runtime. Switchpoline supports both static and dynamic linking.

Switchpoline successfully prevents Spectre-BTB in user-space applications with a negligible mean performance overhead of 0.46% measured in the SPEC CPU 2006 benchmark. Moreover, unlike many x86-specific mitigations, Switchpoline is compatible with existing orthogonal defenses, such as (hardware) CFI or Spectre-PHT mitigations. Given the growing market share of ARMv8 devices, Switchpoline is an important short-term mitigation to prevent widespread exploitation of Spectre-BTB.

Availability

Our prototype has been released as Open-Source Software; it is available on Github:
`https://github.com/typro-type-propagation/TyPro-CFI`

7
Cali: Compiler-Assisted Library Isolation

7.1 Motivation

Software libraries can freely access the program's entire address space, and also inherit its system-level privileges. This lack of separation regularly leads to security-critical incidents once libraries contain vulnerabilities or turn rogue. We present CALI, a compiler-assisted library isolation system that *fully automatically* shields a program from a given library. CALI is fully compatible with mainline Linux and does not require supervisor privileges to execute. We compartmentalize libraries into their own process and kernel namespace context with well-defined security policies. To preserve the functionality of the interactions between program and library, CALI uses a Program Dependence Graph to track data flow between the program and the library during link time. We evaluate our open-source prototype against three popular libraries: *Ghostscript*, *OpenSSL*, and *SQLite*. CALI successfully reduced the amount of memory that is shared between the program and library to 0.08% (*ImageMagick*) – 0.4% (*Socat*), while retaining an acceptable program performance.

7.2 Problem Description

Programs extend their own logic with external libraries, which ease developers' life by offering APIs that abstract from common tasks. Whereas convenient and common practice, linking third-party libraries imposes a significant security risk. Libraries execute in the context of the main program and thereby can freely access the program's entire address space and inherit the program's system-level privileges. At the same time, libraries often contain risky functionality, such as parsers, that do not really need to use the entire program's privileges and address space. This lack of privilege separation and memory isolation has led to numerous critical security incidents.

We can significantly reduce this threat surface by isolating the library from the main program. In most cases, a library (i) neither requires access to the entire program's address space, (ii) nor needs the entire program's privileges to function correctly. In fact, even complex libraries such as parsers require only limited interaction with the program or system. Conceptually, there is thus little need to grant an untrusted library access to the program or critical system privileges. Basic compartmentalization principles thus help to secure a program from misuse by untrusted code. First, *memory isolation* shields the program's sensitive memory from untrusted code parts (e.g., libraries). Second, *privilege separation* reduces the set of privileges of untrusted code parts.

Two recent attempts have proven that library isolation fosters memory isolation *and* privilege separation. Sandboxed API [49] assists developers in isolating the library into its own process (memory isolation) by providing isolation primitives that can be adapted to program-to-library interfaces. To enforce privilege separation, Sandboxed API allows the developer to configure seccomp-BPF [181] filter rules. Similarly, RLBox [124] lets developers split Firefox's libraries into different processes and again uses seccomp-BPF for confinement. Both systems provide primitives to ease library isolation for developers but still require significant manual code changes ("*Migrating a library into RLBox typically takes a few days [...]*" [124]). Given the plenitude of programs and libraries, even such reduced manual effort will severely hinder the wide deployment of library isolation.

The core challenge of automated library isolation is the inherent historic assumption that

programs and libraries share the same address space. Any attempt to split this address space (e.g., into different processes) breaks the underlying semantics if not dealt with accordingly. Passing data across contexts is trivial with primitive data types but quickly becomes challenging for complex objects. PtrSplit [89] is the first general attempt to tackle this issue by marshalling complex data types whenever they cross boundaries. Methodologically, PtrSplit tracks the bounds of complex objects to learn the necessary size information that is required to copy complex objects. While such an analysis allows for automation, (i) deep copies create significant memory and performance overhead (essentially duplicating objects passing to/from libraries), (ii) parallel computation on copies will lead to data inconsistencies, and (iii) PtrSplit's analysis cannot handle type casts. Especially the latter restriction is problematic in practice, as libraries commonly make use of type casts which undermine PtrSplit's analyses. For example, LLVM's `memcpy` cannot be tackled because operands are cast to `void*`, the same holds for generic callback arguments found in various common libraries.

7.3 Contributions

In this chapter, we present CALI, a compiler-assisted library isolation system that uses *shared memory* to allow a secure and efficient interaction between a program and its libraries. While placing a library in its own context immediately creates clear security boundaries, without further precautions, it breaks the program functionality. That is, programs and libraries were designed with the idea in mind that they share the same address space—an assumption that is broken by isolating the library. Consequently, using shared memory makes the use of pointer marshaling obsolete, dropping all its associated disadvantages. CALI's core challenge is preserving the library invocations' full functionality while minimizing the program parts that are exposed to the library. A naïve solution could place the entire program memory in shared memory that is accessible to the program *and* the library, which gained little security as the library had full write access to the main program's data (including pointers). Instead, CALI leverages a Program Dependence Graph (PDG) [40] to infer which memory allocations will (potentially) be passed from the program to the library. To this end, during link time, the PDG observes and propagates data flows crossing the security contexts. CALI then places the according memory regions in shared memory and isolates the remaining memory in the application and library processes, respectively.

CALI is the first system that *fully automatically* shields a program from libraries. CALI is compatible with mainline Linux, does not require supervisor privileges, and has an acceptable space and performance overhead. CALI compartmentalizes untrusted libraries into their own contexts with well-defined security policies. We provide both privilege and memory isolation by placing libraries in their own process that operates in per-compartment kernel namespaces. We apply seccomp filters to minimize the system interface of the library.

We implemented CALI based on LLVM and published its source code.[1] We evaluate the functionality, space, and performance overhead of our prototype against three popular libraries: *Ghostscript* (tested with *ImageMagick*), *OpenSSL* (tested with *Socat*), and *SQLite* (tested with *Filezilla*). CALI reduced the amount of memory that is shared between the

[1] `https://github.com/cali-library-isolation/Cali-library-isolation`

program and library to 0.08% (*ImageMagick*) – 0.4% (*Socat*). Not a single function pointer is shared, such that the threat surface is greatly reduced. CALI's compartmentalization has an acceptable performance overhead when limiting libraries to their least privilege in both memory and system access.

To summarize, our contributions are as follows:

- We present a fully-automated compiler-based separation between trusted and untrusted program logic using compartmentalization and data flow tracking.
- CALI transforms any program to a protected version that can be deployed on mainline Linux without additional requirements about the hypervisor, operating system, or hardware.
- The strict security boundaries of the resulting programs significantly reduce the threat surface, while our three examples show that the performance overhead is acceptable.

7.4 Background and Related Work

Program compartmentalization is the foundation of two related research directions with orthogonal goals. *Memory isolation* hides certain secrets (e.g., keys) in program memory from other parts/compartments. Typically built on top of memory isolation, *privilege separation* [140] also limits system access of certain code parts (compartments) to the least privilege. Either way, compartmentalizing an application is a two-step process: First, an application needs to be separated into two or more distinct code parts (cf. Section 7.4.1, "Compartmentalization"). These code parts must be able to communicate with each other and keep the original application's functionality. Second, the now distinct code parts need to be isolated from each other and executed in different contexts (cf. Section 7.4.2, "Isolation Primitives"). Isolation must guarantee the integrity of the trusted context even if the untrusted context acts maliciously.

We will provide background information and discuss related work in the following. To assist in this discussion, Figure 7.1 provides an overview of key related works, categorized into the two dimensions of the degree of automation (y-axis) and their provided security guarantees (x-axis). CALI aims to fill a gap by providing strong security guarantees (both memory isolation *and* privilege separation) and, at the same time, offering an unprecedented level of automation.

7.4.1 Compartmentalization

The few past manual efforts to split programs into compartments (e.g., Google Chrome [168], OpenSSH [43]) have shown that splitting a program into isolated parts is a tedious and error-prone task. This motivated research on several compartmentalization libraries that aid developers in this process. For example, Privman [76] can be used to split applications in a privileged server and an unprivileged client, requiring changes to the source code: Any library interaction needs to be rewritten manually, and any memory transfer needs to be performed by hand. Privman is merely a library providing the isolation primitives. In addition, Google recently published their framework "Sandboxed API" [49], which

can be used in C++ programs to isolate C libraries in a compartment process with only limited capabilities. To apply Sandboxed API, each and every usage of the library needs to be rewritten manually, which is neither trivial nor convenient. A similar framework is RLBox [124], with a much stronger type system enforcing additional security properties. Thus, summarizing, while all these libraries aid developers significantly, they still do not entirely eliminate the manual analyses and efforts.

In order to lower the manual effort of compartmentalization, researchers proposed various assistance tools that give additional information on where and how to separate the application. These tools are based on annotation-guided program analysis [53, 201, 90, 14] or dynamic analysis [11, 91]. Unfortunately, the strict dependence on source code changes (or annotations) implies that only experienced developers with deep knowledge of the source code and library interactions can compartmentalize software. This lack of automation does not scale for the wide variety of open-source off-the-shelf software.

Researchers have already identified this urgent need for automation. For example, Codejail [200] can separate program privileges *without* forcing developers to rewrite or know the program code *in the program*. However, to separate a program, Codejail requires that every library function needs to be "described" by a developer. To this end, wrapper functions must be written by hand and must specify every single memory transfer between program and library. Furthermore, Codejail's memory isolation is much weaker than related systems. In particular, the untrusted library can read arbitrary memory from the trusted program (hence bugs like Heartbleed cannot be contained). A solution requiring similar developer effort is known for Android apps (CompARTist [65]). While these wrappers can be written once for a library and then be used in many different projects without hassle, Codejail still leaves open the hard work for developers.

Two existing approaches come close to the degree of automation we envision, PtrSplit [89] and SOAAP [53]. Yet there is one fundamentally hard challenge for automating compartments: data flows of non-primitive objects (e.g., pointers, structs) between the compartments. "PtrSplit" [89] uses static analysis to separate annotated variables and related code. In contrast to prior work, PtrSplit can—assuming code annotations by a skilled developer—infer a separation boundary automatically. Furthermore, as PtrSplit tracks data flows, it marshals complex objects for IPC communication between the compartments. While this significantly boosts automation, it still requires code annotations, which in turn require program knowledge. Moreover, while PtrSplit provides memory isolation, it cannot limit the privileges of compartments. Facing these challenges, SOAAP even completely left open isolation (and the required automation) open for future work.

CALI automates compartmentalization to the highest degree possible. Given a program and its libraries, CALI fully automatically splits the program from (a developer-specified subset) of libraries. Developers do not have to care about library interfaces, nor have to know at which parts of their program a library has been integrated. The resulting compartments provide memory isolation *and* privilege separation. CALI only assumes a policy that guides the privileges of each program part, which could be derived automatically [47, 32, 46, 190].

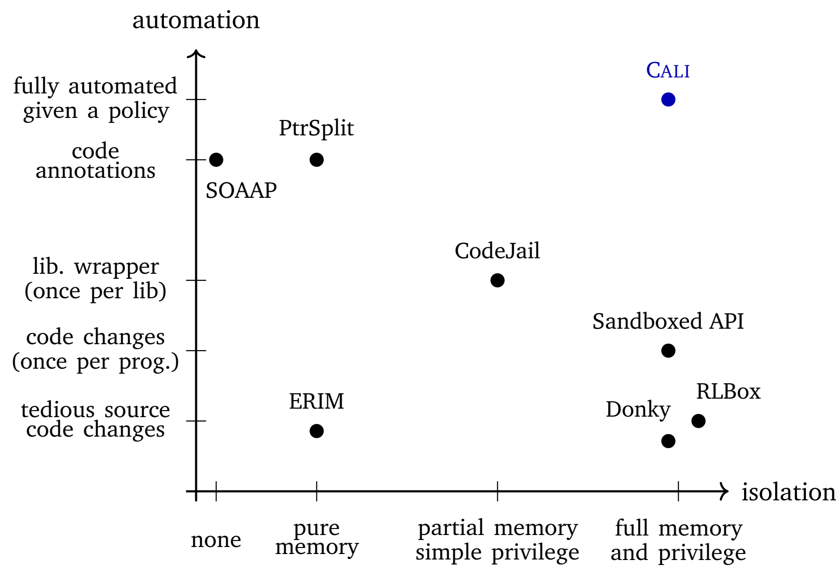


Figure 7.1: Overview of recent program isolation schemes.

7.4.2 Isolation Primitives

Once a program has been compartmentalized, we have to use certain isolation primitives to protect the program parts from each other. That is, assuming well-defined compartment boundaries and interfaces, how can we efficiently enforce isolation between the compartments? To this end, multiple isolation principles can be used, most of which focus only on memory isolation. The conceptually simplest approach is *Software Fault Isolation* (SFI) methods like NaCl [203] ("upro" [128]) or WebAssembly [54] (RLBox [124]), which compile untrusted program parts into sandboxes. SFI requires the source code of the program and all libraries, and it comes with significant restrictions in functionality—not all programs can be compiled to SFI schemes (e.g., JIT compilers). To solve this problem, other systems used existing OS functionality like executing compartments using different Unix users [76]. To further enhance capabilities, the OS kernel can be modified [11, 61, 88, 163, 191], however sacrificing compatibility with unmodified kernels. In the same spirit, researchers leveraged virtualization extensions of modern CPUs and introduce hypervisors for memory isolation (e.g., "SeCage" [91], "TrustVisor" [99], and "Libsec" [141]) Recently, researchers further boosted isolation primitives with hardware-specific features (e.g., CPU extensions) [185, 23, 83, 52, 150], which again reduces broader applicability. All these works demonstrate that special OS or hardware features can be elegantly used to boost additional memory isolation, which is important when aiming to protect certain regions of sensitive data. However, very few of these approaches are compatible with (e.g., process-based) privilege separation. Additionally, they require certain features or adoptions, which hinders a wider isolation deployment in the wide world of resource-constrained devices (think of IoT).

7.5 General Overview

7.5.1 Compiler-Assisted Library Isolation

We now describe CALI, a compiler-assisted compartmentalization privilege separation solution. Our primary use cases are developers or package maintainers who link common libraries into a (Linux-based) application written in a native language like C or C++. The library is given in binary form only, source code is only available from the main program—a pretty common scenario if a third-party library is proprietary or closed-source.

We envision that one of these libraries contains a severe vulnerability, giving attackers full control over the entire program. The developer wants to limit the damage that can occur from such a vulnerability. In particular, we want to protect *private information* (from application memory or files, like stored passwords) and *system integrity* (no system modifications like backdoors). We do not want to protect from *Denial-of-Service* (DoS) attacks that crash the application—*fault recovery* is out of scope. We aim, however, to mitigate resource exhaustion attacks that block the whole system (e.g., memory exhaustion, fork bombs) by limiting the compartment's computation and memory resources.

The application should not lose portability; it must continue working on any system where it worked before modification. No system modification is desired, and super-user permissions are neither available at installation time nor runtime.

Threat model. We envision that a library contains a severe vulnerability that gives attackers full control over the entire process that runs library code, including arbitrary code execution. We do not consider attacks against the kernel, the hardware, or micro-architectural attacks. We also assume a sound permission configuration ([47, 32, 46, 190]). We will discuss our assumptions on and mitigation of cross-compartment vulnerabilities in Section 7.8.4.

Compared to our basic attacker model from Section 2.3, we assume that code execution is possible, but the vulnerability is located in a library.

7.5.2 Overview

CALI performs privilege separation by isolating less-privileged code parts into their own context. While this concept is generic, in the following, we will stick to our main use case of library isolation. CALI automatically handles the interaction between the application and library, which, unlike before, now requires inter-context communication. CALI then reduces the permissions of the library context to the minimally required privileges and memory.

Our protection is applied at application build-time. We build the application using the LLVM toolchain with *link-time optimization* (LTO). When linking the application, all source code files are available as LLVM bitcode. In the first step, we perform a static, interprocedural analysis over the whole application. We determine where and how the library is used and what resources, like memory regions, it needs from the main application. In the second step, we rewrite the main application. We replace all library calls with calls to stub functions that take care of the transition between application and library context. To this end, we make the minimal necessary program memory regions accessible for the library context. Finally, we add a small static library that initializes this context.

After linking, the result is a normal Linux binary that runs on any Linux system without additional dependencies. The required interaction of the developer is minimal: Switch the

compiler toolchain to Clang/LLVM with LTO, add a compiler flag to enable our system, and specify the permissions applied to the library context in a simple format (see Figure 7.8 for a concrete example of a permission configuration). Furthermore, our design guarantees compatibility. Our context implementation uses primitives (processes, namespacing, seccomp, and semaphores) that are readily available in mainline Linux.

7.6 Shielding Compartments

CALI creates a compartment for all libraries that should be isolated. A compartment must fulfill three criteria: First, it must provide *privilege isolation* to prevent attackers from inflicting damage on the computer system. To this end, we must be able to constrain access to files, networks, and other resources (including e.g., computation power and memory) to contain vulnerable libraries successfully. Second, a compartment must provide *memory isolation* to protect the privileged main application (e.g., its pointers) from an attacker in the library compartment. Third, a compartment must preserve the functionality of a program without requiring modifications to the library. Calls to the library must continue working, and memory chunks that are passed from the program to the library must be accessible from the compartment and vice versa.

7.6.1 Basic Compartment Structure

For compatibility reasons, we create compartments based on processes. For each library compartment, we fork a new library process from the main process just before the libraries are initialized. This process is restricted in its permissions and only shares selected memory regions with the main program. Communication between compartments happens over shared memory, semaphores, and an anonymous socket. The library compartment processes sleep until a library function is invoked. Once woken up, they execute the called library function on behalf of the main process and return the result. They terminate when the main process terminates.

Such library compartments do not undermine memory deduplication; the isolated library is still a *shared* library mapped copy-on-write. Multiple processes sharing the library will require one copy in physical memory only. Also, nested libraries are not negatively affected by this scheme. If a library loads other libraries, these will be executed in the context of the loading library, inheriting its reduced privileges.

In principle, we can create compartments for any library. Having said this, we do not isolate standard libraries (e.g., libc). They are the usual interface to access the underlying system. Therefore, standard libraries would need all privileges the main program needs. Any restrictions on the standard library's permissions would also restrict the main application. In our design, each context has its own standard library. It does not matter if a library uses raw syscalls or libc—both execute from the library context with identical privileges.

7.6.2 Shared Memory

We create a segment of shared memory for each compartment and map it in both the main and library process. This memory is mainly used to allocate memory chunks that

are accessed by both processes. To organize this memory, we build drop-in replacement versions for `mmap`, `mremap`, and `munmap` handling memory from this shared memory pool page-wise. These functions, called `shm_mmap`, etc., also synchronize memory mapping between both processes.

We support dynamic memory allocation (e.g., `malloc`) by using a modified version of glibc's heap implementation working with shared memory. In this modified version, we remove the main arena (which cannot be shared) and replace all calls to the `mmap` family with calls to the `shm_mmap` family. Next, we utilize glibc's per-thread heaps to build per-process heaps, preventing concurrency issues between the main and the library process. Following this principle, we have drop-in replacements for all essential memory-allocating functions. If a chunk of memory needs to be shared with the library, we only need to replace its allocation with the appropriate shared allocation. This way, no memory needs to be copied between processes, improving the efficiency of memory transfers. Any memory outside of this shared memory pool is not accessible by the library.

7.6.3 Library Calls

For each library function called by the main program, we create a replacement function in the main program and a handler in the library compartment. All calls to library functions are passed to the library process using a custom IPC-based protocol. We rewrite all calls to the library with calls to this replacement function. This function stores all call-by-value arguments (numbers, pointer values, but not the memory pointed to) in shared memory and signals the library process using a semaphore. The library process invokes the handler, which loads arguments from shared memory and invokes the original function. Once the function returns, the result is stored back in shared memory, and the main process receives a signal using a second semaphore. Finally, the replacement function in the main process loads the return value from memory and returns. This design is entirely transparent to program and library, as long as all pointer arguments point to shared memory (which we will ensure in Section 7.7). In rare cases where libraries call the program (e.g., when the program overrides exported symbols from other libraries), we handle these calls analogously.

7.6.4 Callbacks, Signals, and File Descriptors

Sometimes libraries expect a callback that they will execute once an event occurs, or programs receive function pointers from a library that they will call later. In our context, this is dangerous. Callbacks allow one process (e.g., the library) to trigger the execution of code in another process (e.g., the main program), inheriting the program's privileges. To keep up the isolation between processes, we employ a strict policy: Callbacks are executed in the process they come from (where they have been defined).

Whenever a library call transfers a function pointer, we create a replacement function on the fly and store the original pointer in a lookup table. When invoked, the replacement function invokes the call in the other process.

This design complies with the usual structure of code containing callbacks. Functions passed from the program to the library were typically written in the program's code base and require access to the program's internal data. Executing them in the program context thus preserves compatibility. Functions returned and defined by the library might depend

on library-internal data and should stay in the library compartment for compatibility and security reasons. The library cannot invoke arbitrary code; the function pointers it passes execute in the library process.

Signals are handled using this callback mechanism: When a signal handler is registered in one compartment, the handler is synchronized with all other compartments. When a signal is caught, the handler executes in the compartment that registered it. If an uncaught signal terminates one process, the others also terminate.

File descriptors are handled differently. We detect them using static analysis (Section 7.7.7), not by type. When a descriptor is passed as a function call argument or return value, the other side gets full access to the descriptor. A duplicate is handed over on a shared socket, and FD numbers are adjusted between the processes. When the descriptor gets closed in one process, the descriptor is also closed in the other one. In Linux, most system resources are represented by file descriptors, and correct synchronization ensures a synchronized view of the system.

7.6.5 Isolation

Isolation between processes is provided by OS primitives present on any up-to-date Linux system. The exact isolation can be specified by an isolation policy and might depend on runtime data (e.g., environment or program arguments). The policy is given by the developer at compile-time, Figure 7.8 shows an example policy. In the following, we describe several isolation mechanisms that we deploy using a modified version of `nsjail` [50].

We put each compartment process in a new *mount namespace*, mount all accessible directories to an empty folder, and finally use *chroot* to jail the compartment into this directory. We utilize a *user namespace* to execute chroot without requiring higher privileges or capabilities. As a result, the library compartment process sees a file structure similar to the real system, but it contains only folders if access has been allowed by the policy. If only read access is desired, we mount the folder with the read-only attribute.

We use a *network namespace* to prevent a library compartment from communicating with other machines if not allowed by the isolation policy. Next, we use a *PID namespace* to protect other processes running on the system. Additionally, subprocesses spawned by the library will not continue running after the library compartment has been shut down. After forking the compartment process, we drop Linux capabilities or super-user rights the main program might have, according to the isolation policy.

The isolation policy can specify constraints on the computing resources used by the library compartment, enforced using *rlimit*. These constraints prevent DoS attacks on the system, like consuming all available memory, blocking all CPU cores, or “fork-bombs.” They are not meant to prevent application DoS (e.g., program crashes).

Finally, we apply a *seccomp* policy to restrict the set of system calls the library compartment can call.

7.6.6 Threading, Forks, and Concurrency

Our prototype has limited support for concurrency: While it does not break the semantics of threading and forking processes and, in fact, also works for multi-threaded or multi-processing programs, our locking mechanism serializes all threads at the library interface.

Therefore, concurrency is not an issue, as only one thread can call a library at a time. To get the full performance of concurrent execution, the library compartment process must be enabled to spawn its own threads, mirroring the threads in the main program process. For new threads, the communication structure must be cloned as well. The same structure can handle forking programs. In contrast to other work (that uses thread-like primitives for isolation), this extension does not impact security. However, we refrained from extending our prototype to multiple threads because this is not required to show the general feasibility of our approach.

Concurrently running code from multiple compartments opens up another problem: An attacker in one compartment could modify shared memory while another compartment uses this memory, possibly leading to memory corruption in other compartments, effectively weakening the introduced isolation. These issues are called *double-fetch bugs* [152, 192]. Other systems like PtrSplit or Sandboxed API avoid this problem; they copy memory instead of sharing it. However, this approach breaks existing software with legitimate use cases of shared memory: for example, most implementations of synchronized collections or spinlocks rely on shared memory; they cannot be used in multiple compartments with traditional methods.

CALI solves this issue by providing three modes of operation: In the *default mode*, concurrent access is allowed (to not break existing software). We consider the security impact of concurrent access rather low; manual inspection of the shared memory usage in our three example programs did not show vulnerable memory usage. Related work [152] can be used to counter potential double-fetch bugs.

If the program is clearly not concurrently accessing memory, CALI can be used in *mprotect mode*. After a library function has finished executing, the shared memory is set read-only in the library process. Before the next library function call, the shared memory is set writable again. A custom seccomp filter is installed to prevent attackers from changing the protection of shared memory manually. With this extension, an attacker in the library process can only modify shared memory while a library function is being executed. Even if the attacker has started additional threads, the security level is equal to memory-copying solutions. The downside of this mode is that only one thread can execute a library function at a time.

If the library is not concurrent itself (e.g., does not spawn threads or processes), CALI can be used in *non-concurrent mode*. The library process uses a seccomp filter to prevent forks, clones, or thread-spawning syscalls. Without these privileges, an attacker cannot run code outside of library calls: After a library call returns and the main compartment continues execution, the single library thread is blocked until the next library call is requested. This mode does not have additional overhead but prevents libraries from using concurrency.

7.7 Compiler-Assisted Separation

To automatically split an application into two parts, we have to know which memory chunk is used both by the program and the library, and which memory is used exclusively by the main program. We call chunks *common memory* if both compartments use them. Ideally, all other program memory should not be taken from the shared memory pool, as it otherwise might leak data to the library compartment. Having said this, sharing “too much” memory only weakens security guarantees and does not break functionality.

One core challenge is that we need to know *at allocation time* if a memory chunk is *common memory*. After memory has been allocated, moving the chunk might interfere with legacy code, e.g., pointers to the chunk scattered over the program would need to be updated. Similarly, we also have to identify all (potentially indirect) calls to library functions. They determine which memory is going to be accessible by the library.

To gather all this information, CALI uses *inter-procedural static analysis* on the compiled LLVM bytecode of the main program. In the first step, we use a *program dependence graph* (PDG) with similarities to the one proposed by Liu et al. [89], tracking the data flow of memory chunks. We detect all calls to library functions and tag all memory allocations that might reach library functions. In a second step, we rewrite all these memory allocations to use shared memory, generate replacement functions for used library functions, and finally, rewrite all calls to library functions with calls to these replacements.

7.7.1 Background: Call Graphs and SCCs

To schedule our analysis operations on the program, we use the *strongly connected components* (SCC's) of the call graph. A strongly connected component of a graph $G = (V, E)$ is a maximal subset of vertices V' where a path between all vertices exists ($\forall v_1, v_2 \in V'.\ v_1 \rightarrow^* v_2$). The graph formed by all strongly connected components in a call graph has nice properties: First, functions that can call each other in a recursive way (nested recursion) are contained in the same SCC. Every other function is in its own SCC of size 1. Second, because all recursive functions are contained in joined SCCs (one per recursive function group), the call graph of all SCCs is acyclic (circles in a call graph indicate recursion, which only happens inside SCCs). Third, traversing the SCC callgraph bottom up traverses all functions in a callee-before-caller order. Traversing the SCC callgraph top-down traverses all functions in a caller-before-callee order. To reduce the average SCC size, we only consider calls that can transfer memory by reference (not only by value). We ignore calls if all parameters and the return value are constant or of primitive type (e.g., `int`, `char`), because they are irrelevant for our following analysis.

Our analysis traverses a SCC callgraph of the whole program in a bottom-up fashion and analyzes all functions in a SCC at once. If we encounter a function call, it either targets a function we already analyzed or a recursive function in the same SCC that we are just analyzing. At a later stage, we will traverse the SCC callgraph top-down to propagate information from calls to called functions.

7.7.2 Analysis Phase: Overview

In the analysis phase, we mainly need to determine which memory allocations generate *common memory* (that later needs to be shared). The analysis consists of three phases:

1. Creation Phase: We construct a PDG containing information about intra-procedural data flow. We mark memory allocations and locations of common memory.
2. Reachability Phase: For each function group (callgraph SCC), we determine the reachability between memory allocations, common memory expectations, the function's arguments, and return value. We store the result in a function summary in the PDG, which is used when calls to this function are analyzed (inter-procedural data flow).

```
1  struct X { long one; long two; };
2
3  // Library function we need to compart
4  void libfunc(int *err, char *input, long *output);
5
6  // Main program
7  void main() {
8      struct X *x1 = new_struct(13L);
9      struct X *x2 = new_struct(37L);
10     struct X *x3 = update(x2);
11     char *buffer = malloc(1024);
12     lib_wrapper(buffer, x3);
13 }
14
15 struct X *new_struct(long init) {
16     struct X *s = malloc(sizeof(struct X));
17     s->one = init;
18     return s;
19 }
20
21 struct X *update(struct X *x) {
22     x.one = 18;
23     return x;
24 }
25
26 int lib_wrapper(char *buffer, struct X *x) {
27     int err;
28     libfunc(&err, buffer, &x->two);
29     return err != 0;
30 }
```

Figure 7.2: Example program passing memory to a library.

3. Specialization Phase: We check for functions that should return pointers to common memory depending on their usage. For example, wrappers around `malloc` like `calloc` or `new` should only generate common memory if their result is later passed to the library. However, these functions are called from many other functions. To keep a precise result, we thus clone these functions (including their containing SCC). The cloned functions will return common memory, while the original functions will not. Call sites are adjusted, and reachability analysis is repeated.

Figure 7.2 shows an example program we will use to illustrate the analysis. It consists of two struct instances, where one of these structs is used in a library, and the other is not. The reference to that struct passes multiple functions before being used as a library function argument. Figure 7.3 shows the LLVM translation of the program. The full PDG with all analysis results is given in Figure 7.7, and an excerpt illustrating the most important aspects is shown in Figure 7.4.

```
1  %struct.X = type { i64, i64 }
2
3  define void @main() {
4    %1 = call %struct.X* @new_struct(i64 13)          ; x1
5    %2 = call %struct.X* @new_struct(i64 37)          ; x2
6    %3 = call %struct.X* @update(%struct.X* %2)       ; x3
7    %4 = call i8* @malloc(i64 1024)                   ; buffer
8    %5 = call i32 @lib_wrapper(i8* %4, %struct.X* %3)
9    ret void
10 }
11
12 define %struct.X* @new_struct(i64) {
13   %2 = call i8* @malloc(i64 16)
14   %3 = bitcast i8* %2 to %struct.X*                 ; s
15   %4 = getelementptr %struct.X* %3, i64 0, i32 0
16   store i64 %0, i64* %4, align 8
17   ret %struct.X* %3
18 }
19
20 define %struct.X* @update(%struct.X*) {
21   %2 = getelementptr %struct.X* %0, i64 0, i32 0
22   store i64 18, i64* %2, align 8                    ; one=18
23   ret %struct.X* %0
24 }
25
26 define i32 @lib_wrapper(i8*, %struct.X*) {
27   %3 = alloca i32, align 4                          ; err
28   %5 = getelementptr %struct.X* %1, i64 0, i32 1
29   call void @libfunc(i32* %3, i8* %0, i64* %5)
30   %6 = load i32, i32* %3, align 4
31   %7 = icmp ne i32 %6, 0                            ; err!=0
32   %8 = zext i1 %7 to i32
33   ret i32 %8
34 }
```

Figure 7.3: Simplified LLVM code of the example in Figure 7.2.

7.7.3 PDG Construction

In contrast to other PDGs [89, 40], our graph does not need control dependence. The construction of our graph is based on the LLVM bytecode of the program. This bytecode is in *static single assignment* (SSA) form, meaning that every LLVM value gets assigned only once (at definition time, typically as the result of an instruction) [92]. Our PDG is based on LLVM values, and every value is represented by a PDG node. Nodes are additionally tagged with the type of the value and the function where it is contained, so every node is a three-tuple: $(value, type, function)$. We add similar nodes for all global variables and function arguments.

To trace actual memory chunks, we inspect the values further. We disassemble every complex data type (e.g., pointers, structs) into its basic data types. These so-called "subnodes" represent single members of structs or memory referenced by a pointer type in the graph. The rules to create subnodes are given in Figure 7.5: For every pointer type, we create a subnode

representing the memory pointed to, and connect it with a "pointer $\xrightarrow{\text{Deref}}$ memory" edge. For every field in a struct or union type, we create a subnode and connect it with a "struct $\xrightarrow{\text{Part}_i}$ field" edge (where i is the index of the field). We repeat this algorithm on all new subnodes up to a configurable recursion depth of 5. Limiting the depth prevents the analysis from being trapped in endless recursion induced by recursive structs. In theory, this limitation might lead to a loss of precision (when data flow is hidden deep in the subnodes). However, in all our example programs, a recursion depth of 3 was sufficient to cover all necessary information. This configurable limit should hold for most programs and can be adapted if required. We also limit the number of struct members to 32 for performance reasons. In our examples, we did not see any impact on the analysis precision by this restriction. Figure 7.4 shows an example subgraph for the second argument of `lib_wrapper` (the struct pointer `x: struct X*`). The subnodes represent the dereferenced struct of that pointer (`:struct X`), further dissected into its fields (`X.one`, `X.two`).

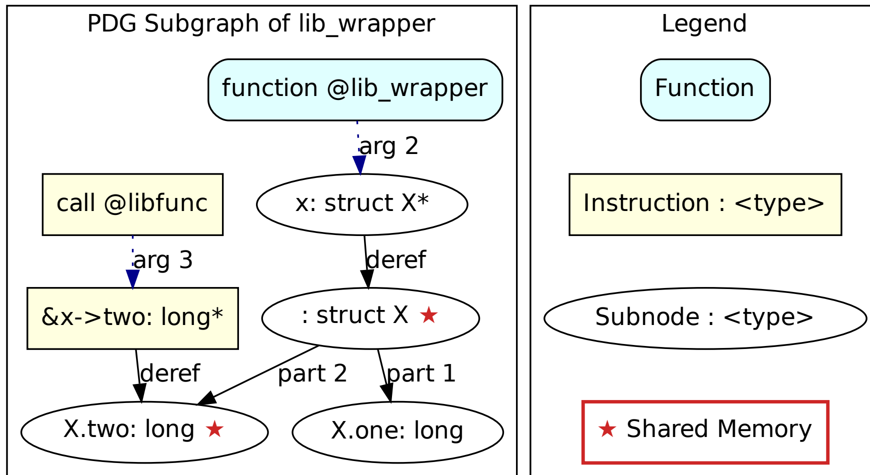


Figure 7.4: Excerpt of the PDG from the example program. The full graph is given in Figure 7.7.

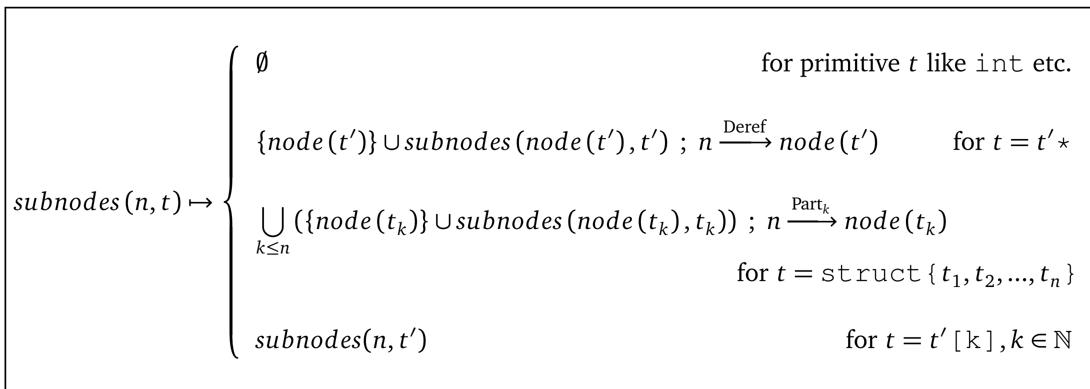
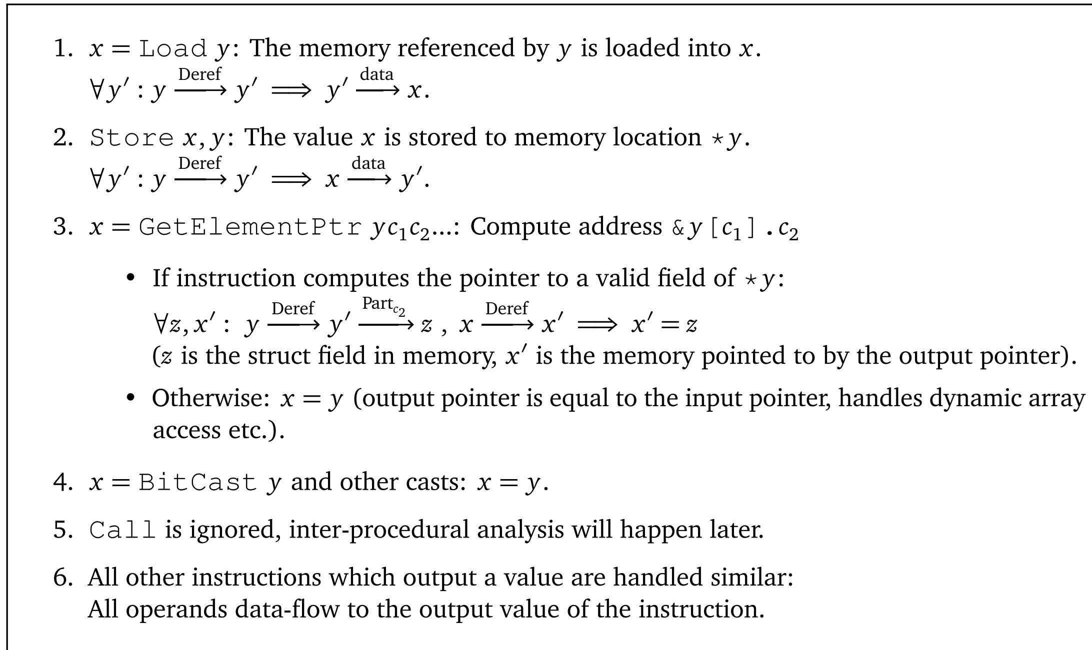


Figure 7.5: Recursive algorithm to generate subnodes of a value node n with type t. $node(t)$ creates a new subnode with type t.

7.7.4 Data Flow in PDGs



1. $x = \texttt{Load}\ y$: The memory referenced by y is loaded into x.
   $\forall y' : y \xrightarrow{\text{Deref}} y' \implies y' \xrightarrow{\text{data}} x$.
2. $\texttt{Store}\ x, y$: The value x is stored to memory location $*y$.
   $\forall y' : y \xrightarrow{\text{Deref}} y' \implies x \xrightarrow{\text{data}} y'$.
3. $x = \texttt{GetElementPtr}\ y c_1 c_2 \ldots$: Compute address $\texttt{\&}y\,\texttt{[}c_1\texttt{]}\,\texttt{.}c_2$
   - If instruction computes the pointer to a valid field of $*y$:
     $\forall z, x' :\ y \xrightarrow{\text{Deref}} y' \xrightarrow{\text{Part}_{c_2}} z\ ,\ x \xrightarrow{\text{Deref}} x' \implies x' = z$
     (z is the struct field in memory, x' is the memory pointed to by the output pointer).
   - Otherwise: $x = y$ (output pointer is equal to the input pointer, handles dynamic array access etc.).
4. $x = \texttt{BitCast}\ y$ and other casts: $x = y$.
5. `Call` is ignored, inter-procedural analysis will happen later.
6. All other instructions which output a value are handled similar:
   All operands data-flow to the output value of the instruction.

Figure 7.6: Rules to determine data flow for LLVM instructions.

We next extend the PDG with edges representing intra-procedural data flow. If the LLVM value of a node or subnode n might carry over to another node n' in any possible execution of the program, we assume a data flow from n to n' and add an edge $n \xrightarrow{\text{data}} n'$. A typical example is a load from memory: Reading `err` for the return statement in `lib_wrapper` is `%6 = load i32* %3` in LLVM, we summarize the load as a data edge from the subnode `err` of `%3` (representing the referenced memory) to the output value `%6`.

As a major difference from previous PDGs [89], we use *data equality* to capture pointer aliases. If two (sub)nodes represent the same value storage (i.e., the same memory location is referenced by two pointers), we consider them to be *data-equal*. If one node's associated value gets updated, the other node's value will also change. We could handle this situation with bi-directional $\xrightarrow{\text{data}}$ edges, but merging significantly reduces the size of the graph and improves the runtime of all further operations. Merging nodes might introduce cycles into the subnode graph if subnodes connected to each other are merged (e.g., linked list structure). The subnodes of a value no longer form a tree, which is vital for handling recursive data structures. Merging nodes also eliminates nodes introduced by type casts.

We use six rules for LLVM instructions that describe the data flow between inputs and output of an instruction (Figure 7.6). To be on the safe side regarding functionality, our rules might over-approximate data flow but should not under-approximate it.

Data flow between nodes x and y ($x \xrightarrow{\text{data}} y$) is propagated recursively to its subnodes by two simple rules:

1. If there is data flow between two structs x and y, then there is also data flow between their members:

$x \xrightarrow{\text{data}} y \wedge x \xrightarrow{\text{Part}_i} x' \wedge y \xrightarrow{\text{Part}_i} y' \implies x' \xrightarrow{\text{data}} y'$

2. If there is data flow between two pointers x and y (pointers might alias), then we consider the referenced memory data-equal:
$x \xrightarrow{\text{data}} y \wedge x \xrightarrow{\text{Deref}} x' \wedge y \xrightarrow{\text{Deref}} y' \implies x' = y'$

Rule 1 expresses that copying a struct from one location to another implies that all struct members are copied, too. Rule 2 can best be explained with the running example. In function `main`, data flows from `%2` to `%3`. Therefore, the referenced memory is the same (node '`: struct X`' representing the struct's memory), as `%3` is an alias of `%2`.

7.7.5 Reachability Analysis

After these inferences, the PDG contains all values in a program and their intra-procedural relations. Equipped with this PDG, we can determine if a memory allocation must produce common memory for a library call in the same function. If there is a data flow path from the memory allocation output (*source*) to a library call's arguments (*sink*), then the memory must be shared. Such data flow problems can be modeled as reachability over $\xrightarrow{\text{data}}$ edges in the PDG.

First, we use data flow to determine which indirect calls might call a library function (find sinks). Then, we determine which nodes are common memory (reach a sink) within a function. Finally, we extend this to an inter-procedural analysis. We describe these three steps more detailed in the following.

7.7.5.1 Determine indirect library calls

To decide if a call needs common memory, we need to decide if it could potentially call a library function. While this is trivial for direct calls, it is not obvious for indirect calls. Some programs, for example, build a `struct` of library function pointers as an exchangeable "interface" against a library. We search for paths backward in the PDG from the called function address (sink) to a library function (source). The backward path from sink to source must only consist of $\xleftarrow{\text{data}}$, $\xleftarrow{\text{Part}_i}$ and $\xrightarrow{\text{Deref}}$ edges. From the indirect call address (sink), we follow $\xleftarrow{\text{data}}$ edges backward that lead us towards the origin of the address value. We also follow $\xrightarrow{\text{Deref}}$ edges in the forward direction (pointer to memory) to get from function addresses to actual functions (sources). $\xleftarrow{\text{Part}_i}$ is used to cope with compiler optimizations.

This analysis might over-approximate which call might invoke a library function to preserve program functionality. Having said this, we did not observe such an over-approximation in the three real-world programs in our evaluation.

7.7.5.2 Intra-procedural Reachability Analysis

At this level, we know which function calls invoke library functions. All their call arguments must point to shared memory later, so we first mark everything referenced by an argument of these calls as *common memory*. More formally, we mark everything that can be reached from a call argument using only (and at least one) $\xrightarrow{\text{Deref}}$ edge. We will use this mark

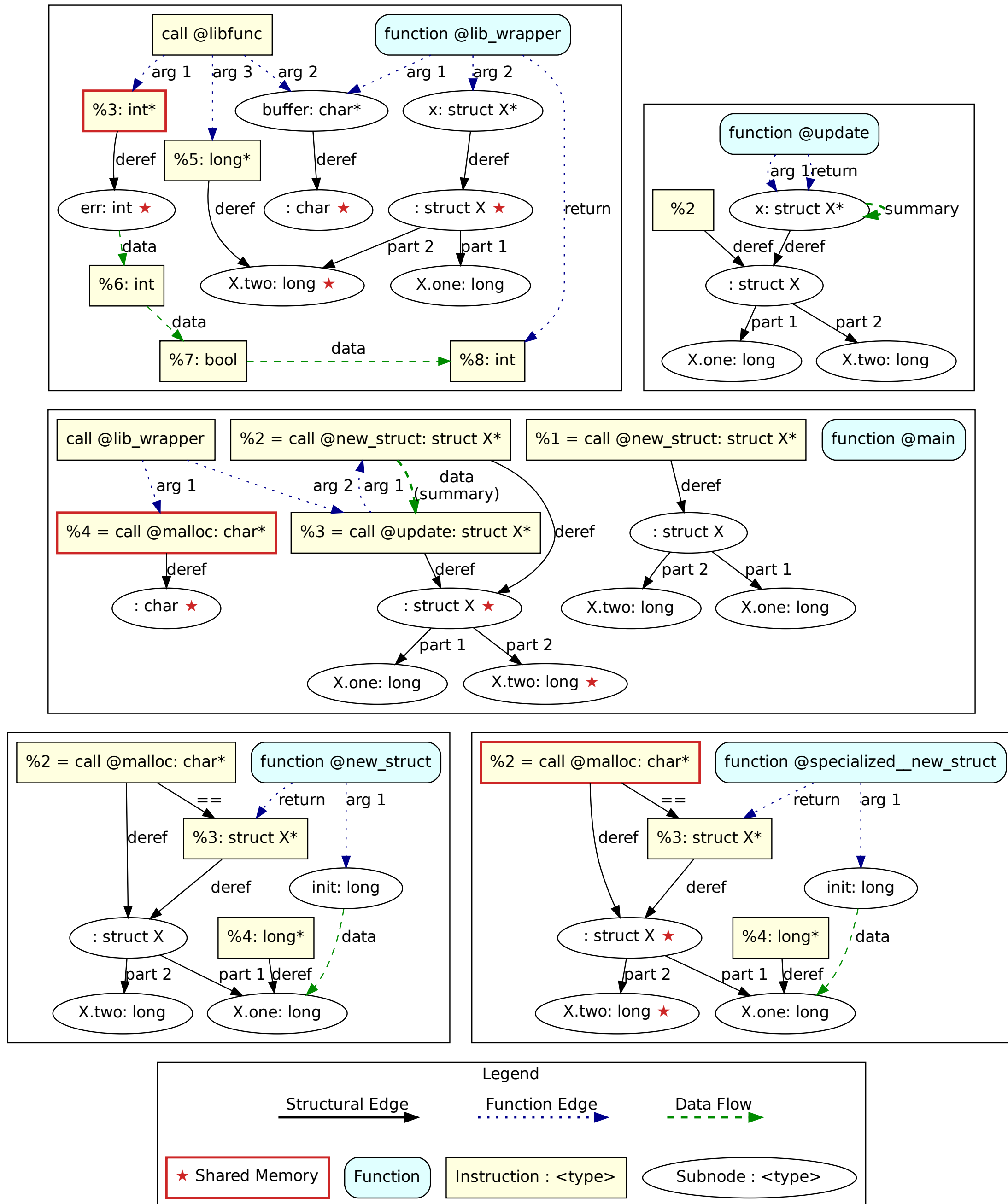


Figure 7.7: The full Program Dependence Graph from the example program, after all analyses have been applied.

as taint and propagate it backward, marking all nodes that might reach a library call. If that mark reaches the memory produced by a memory allocation, this allocation needs to be rewritten. Formally, an allocation instruction x needs to return shared memory iff $\exists x' : x \xrightarrow{\text{Deref}} x' \land \text{marked}(x')$.

In our example in Figure 7.2 (and Figure 7.3), we first mark the dereferenced arguments of the call to `libfunc` in `lib_wrapper` with a star, which includes `err: int`, the memory referenced by parameter `buffer` and `x->two`. The struct behind pointer `x` must be shared, but detecting this requires further analysis:

To propagate the common memory mark, we follow all $\xleftarrow{\text{data}}$ and $\xleftarrow{\text{Part}_i}$ backward edges and mark every node we can reach. Formally, if $x \xrightarrow{\text{data}} y$ and y is marked, then x needs to be marked because there is a path from x to a library call. Marks are also propagated to and from global variable nodes. We do not need to follow $\xrightarrow{\text{Deref}}$ edges here, as all possible dereferences have already been marked by the initial marking step. In our example, we need to mark the `struct X` node, given that struct member `two` needs to be shared and the struct members reside together in memory (Figure 7.4).

7.7.5.3 Inter-procedural Reachability Analysis

The analysis so far handles all cases where allocation and library call are in the same function. We extend the analysis first to a group of functions, then to the whole program. To cover recursion, we analyze all functions in a SCC (see Section 7.7.1) together at once. We resolve all `Call` instructions targeting functions within the same SCC. We connect the function's argument nodes with the parameter values from all actual calls (with a $\xrightarrow{\text{data}}$ edge), and we connect the value of the `Ret` instruction in the callee with the result value of all `Call` instructions in the callers. Once we re-run the data flow analysis, it covers data flow between all functions in this SCC, possibly over-approximating (because no call context sensitivity is given in our algorithm).

Analyzing functions SCC-wise is a good trade-off between whole program analysis and function-wise analysis. Analyzing all functions at once is usually not feasible in acceptable time, and we cannot afford to lose context sensitivity on all functions. Function-wise analysis is much faster but cannot handle nested recursive functions. The SCC-wise analysis of our algorithm picks the best of both worlds: For non-recursive functions, it boils down to function-wise analysis; only in the case of nested recursive functions is SCC-wise analysis slower (but much more precise).

To extend SCC-wise analysis to full inter-procedural analysis, we traverse the SCC call graph bottom-up and run the SCC-wise analysis on each SCC. Due to the properties of a SCC call graph, we visit callees before callers, and recursion only occurs within a SCC. That is, if we encounter a `Call` instruction, it points either to a function within the same SCC or it points to a function in an already analyzed SCC.

When analyzing a SCC, we create a *summary* for each contained function, similar to parameter trees from [89]. A summary captures all possible data flow and indirect function invocations between arguments, return value, and global variables used in a function, including shared memory markers. When we later see a call to that function, we insert the precomputed summary, ignoring the full graph for the function itself.

In our example graph (Figure 7.7), we have built a summary edge for `update`. In

`main`'s call to `update`, we copy the summary edge between `x2/%2` and `x3/%3` (and unify the dereferenced structs). We also copy the three memory markers from the `lib_wrapper` to the arguments of its call. With this information, we can reason that the `malloc` call in `main` must be shared because its memory will be passed to a compartment (`buffer`).

7.7.6 Function Specialization

At this point, our analysis cannot handle functions allocating (potentially shared) memory and returning its reference. Examples are `calloc` and the `new` operator from C++. Both internally use malloc to allocate memory and return a reference to initialized memory. In our example, `new_struct` allocated memory that must (`x2`) or shouldn't (`x1`) be shared. The naïve solution (propagating the marks in both directions) would mean that *all* calls to `new_struct` would return shared memory. If that happens to `calloc`, the majority of memory used in the program might be affected—a prohibitive over-approximation.

We tackle this problem using *function specialization*, which is executed after the reachability analysis. We first determine if a function could potentially create and output memory chunks. Next, we check if any calls to this function require these chunks to be common memory (i.e., if the function must return shared memory for some calls). If so, we clone the function. The original function is unmodified, and the clone (the specialized function) will create shared memory. To this end, we traverse the SCC call graph top-down: For each SCC, we identify the memory output nodes. A memory output node is a subnode of the return value that is reachable over at least one $\xrightarrow{\text{Deref}}$ (function returns a pointer) or a subnode of an argument node reachable over multiple $\xrightarrow{\text{Deref}}$ edges (function stores a pointer in a reference-passed variable). For each call to a function in this SCC, we relate the actual call argument subnodes with the function argument subnodes and relate the call result subnodes with the function return subnodes based on the subnode graph structure. If any of the related nodes are marked as common memory, we specialize (clone) the whole SCC. We copy all marks from all calls to the argument nodes of the specialized version and re-run the reachability analysis, forcing the function to output common memory. All calls that contributed markers are pointed to the respective specialized function.

In our example, the function `new_struct` is called once with tagged memory output (`x2` from the second call in `main`). We create a copy `specialized__new_struct` and copy two markers to the specialized function. We see that `malloc` in the specialized version must return shared memory. We thus update the second call in `main` to call `specialized__new_struct`. As the end result, just one struct in `main` is in shared memory (`x2 / %2`), while the other one (`x1 / %1`) is not.

Updating calls inside a specialized function might require further callees to be specialized. To this end, we iterate the SCC call graph in caller before callee order, including both original and cloned SCCs.

Function specialization potentially increases the program size. To reduce the space overhead, we schedule two LLVM passes after specialization: "Dead Global Elimination" removes the old function if all calls get specialized, and "Merge Functions" unifies cloned functions that have not been changed.

After running the function specialization pass, we finally have a PDG that knows for each memory allocation if it should produce shared memory or not.

7.7.7 Tracing File Descriptors

We trace file descriptors along with memory chunks but with much simpler rules. We use a list of known functions that return new file descriptors (e.g., `open`, `socket`). Calls to these functions are the *sources* of our data flow analysis and are tagged with FD. FD tags propagate forward along $\xrightarrow{\text{data}}$ edges only. During the reachability analysis phase, they are copied from callees to callers. During the specialization phase, they are copied from callers to callees. Function arguments of library functions are the *sinks* of this analysis; if a library function argument is marked with FD, it denotes a file descriptor that needs to be handled separately. This algorithm detects all file descriptors passed from the program to the library. A similar algorithm can be used to detect file descriptors that are passed from a library function to the program.

7.7.8 Rewriting Memory Allocations

In LLVM programs, we have three types of memory allocation: global variables, stack variables (`Alloca` instruction), and calls to memory-allocating functions. For each of these allocations, we can easily check if it must be shared: If the return value node of the instruction (a pointer) has a $\xrightarrow{\text{Deref}}$ edge to a subnode marked as common memory, then the allocation must be shared. We share calls to memory-allocating functions by replacing them with their shared counterpart (`shm_malloc`, see Section 7.6.2). We provide these replacements for all primitive memory-allocating functions. Higher ones will be resolved using function specialization. We share global variables by moving them to a special page-aligned section in the ELF binary, which will be mapped shared at runtime. We move shared stack variables to our shared heap. They are initialized and freed in the function prologue and epilogue, respectively.

After rewriting memory allocations, we have a program that can run in compartments without breaking functionality.

7.7.9 Data-Transferring Call Graph Analysis

To reduce the runtime of our analysis, we attempt to minimize the size of the SCCs. The more SCCs we have (and as smaller they get), the more calls can be analyzed with call context sensitivity. All our cross-function analyses attempt to track memory chunks using data flow analysis. Function calls that do not take or return memory references cannot move any memory chunks and, therefore, cannot compute interesting data flow. Thus, instead of a normal SCC call graph, we use a *data-transferring call graph* as the basis for the SCC computation. In this graph, we only consider calls that are able to transfer memory by reference (not only by value). We ignore calls if all parameters and the return value are constant or of primitive type (e.g., `int`, `char`, `float`, `double`). For ImageMagick, this relaxed notion reduced the maximal SCC size from around 500 to 23. With our data-transferring call graph, many of these calls are not considered, and the big nested recursive SCC is broken down into many small, easy analyzable SCCs.

7.8 Evaluation

We implemented our CALI prototype in C++ and evaluated it on three sample applications. We chose these applications to cover many different aspects: Different languages (C and C++), user interfaces or console, local and networking applications, and different code sizes. Furthermore, these programs link all libraries during the compilation phase (i.e., no dynamic loading) and thus perfectly suit the link-time passes of CALI. All applications use different, widely used libraries that contained severe vulnerabilities in the past:

ImageMagick is a large (453,000 LoC) image processing tool suite written in C. ImageMagick uses *Ghostscript* to read/write postscript and PDF files, which had some serious bugs in the past [20, 108]. We protect ImageMagick's `convert` utility, which is used to convert between file formats by isolating *Ghostscript*.

socat is an all-round utility for networking. Socat can create connections between almost any kind of endpoints. Its C code base is rather small (29,000 LoC). Socat can establish encrypted TLS connections using the *OpenSSL* library, which had several severe vulnerabilities in the past [110].

Filezilla is a popular FTP client with a wxWidgets GUI. Its large codebase (190,500 LoC) is written in C++ and scattered over different projects. Filezilla uses *SQLite* (with critical vulnerabilities in the past [109, 111]) to manage download queues and store known servers. SQLite is a popular library [183] (even incorporated in major operating systems) due to its permissive licensing. In 2019, a critical vulnerability appeared, which required patches in countless applications [109, 111].

We have chosen these example programs because they are widely known, use libraries with vulnerabilities in the past and cover different areas (computation, networking, and user interfaces). We evaluate the functionality of CALI on more programs taken from the most popular Debian packages [5].

7.8.1 Correctness Evaluation

We apply CALI on each of these applications and check if the resulting binary is still fully functional. We additionally instrumented each library interface to catch more subtle bugs.

ImageMagick's functionality can be verified using the provided integration tests (that call the protected `convert` binary). After the protection with CALI, we repeated all provided tests 50 times and found no difference in behavior. Next, we added additional tests: We chose eight popular image formats (including all formats handled by Ghostscript), prepared sample files for each format, and converted each format into each other one. All converted images were identical to the ones produced by an unmodified `convert` program.

socat does not provide integration tests. Therefore, we combined several socat instances using different types of connections, transferred large amounts of data between them (1000 connections, up to 1 GB per connection), and verified the transfer was working, and no data got changed. In detail, the socat "client" configuration reads data from a file, sends it over a TLS connection to a "server." This server is another socat configuration listening for TLS connections and using echo to send incoming data back. A third socat, our "proxy" configuration, was sitting in the middle, using a TLS server to read connections from the client and using a TLS connection to proxy incoming data to the real server. No socat

configuration showed a different behavior after being protected by CALI.

Testing *Filezilla* is tricky because no official tests are provided, and its GUI is hard to automate. We resorted to manual testing, using a protected Filezilla to connect to various servers and testing all the functions. We especially focused on the parts that used SQLite: the download queue and the server configurations. Again, we could not see any behavioral differences.

From our tests, we can conclude that CALI does not break the functionality of protected programs, given a sound security policy configuration.

To evaluate CALI on an unbiased selection of binaries, we analyzed the most popular binaries taken from Debian Popularity Contest [5] (top 300 packages). We select every binary that (1) links dynamic libraries besides the standard libraries that can run with reduced privileges, (2) can be compiled with Clang/LLVM 7, (3) comes with a working set of integration tests. We rebuild these binaries with CALI enabled and used their integration tests to verify that CALI did not break anything. We additionally instrumented the library interface to detect errors in memory sharing reliably, no errors occurred. We confirm that CALI works on all tested binaries, which are: dpkg-deb, dbus-daemon, man, mandb, accessdb, whatis, gpg, gpgv, gpgsm, scdaemon, xz, xzdec, fc-cache, fc-list. We isolated the most important libraries, namely libbz2, liblzma, libz, libexpat, libgdbm, libksba, libsqlite3, and libfontconfig. Each binary used up to 4 of these libraries. Depending on the quality of the provided integration tests, 35%–80% of all library call locations have been covered during these tests. Manual inspection revealed that the uncovered library callsides were mostly error handling or dead code. Additionally, some binaries had tests that did not require any library calls: sudo, gpg-agent, dirmngr, file, shared-mime-info, and pstree. CALI did not break any binary.

7.8.2 Usability Evaluation

CALI is designed to be easily deployable in real-world systems. To apply CALI, we just need to enable link-time optimization (`-flto`) and add our linker. In most build systems, it is sufficient to add CALI using a common environment variable: `LDFLAGS="-fuse-ld=cali -Wl,--cali-config=permissions.yaml"`. Next, we specify which libraries should be separated and which privileges they should have in a simple text-based config file (see Figure 7.8). For most applications, integration worked as simple as that.

Only in exceptional cases CALI needs additional information to handle corner cases. In our examples, ImageMagick uses a custom memory allocator instead of `malloc`. Here we need to configure which function allocates and deallocates the memory (2 lines in the configuration file, no source code changes required). Socat and Filezilla did not require any annotation. Overall, integrators thus do not need to know application internals, With this one exception, no deep knowledge of the application internals was necessary, CALI inferred all other information automatically.

For example, we show the full configuration file of ImageMagick `convert` in Figure 7.8. The contained Ghostscript library is limited to access only the folder where the input and output files reside, `/tmp` (where ImageMagick might put additional files), and its installation directory (where for example color profiles are located). The network is not available in this compartment. User and PID namespacing is enabled by default. The selectors describe

which code belongs in which context: At link time, ImageMagick consists of all `.o` files (and some static libraries). Not in a specific context (not named in any selector) is the standard library (libc); it can be called from both contexts and always executes in the calling context. Lines 7–9 mark the custom heap implementation as required in exceptional cases. Our prototype is able to auto-generate most parts of the necessary configuration file, for example, the selectors. Users must only specify permissions (the green highlighted lines) and possible custom heaps (the red lines).

```
1  ---
2  contexts:
3    main:
4      selectors:
5        - "*.o"
6        - "libMagick*.a"
7      function_behavior:
8        AcquireMagickMemory: malloc
9        RelinquishMagickMemory: free
10   library:
11     selectors:  # a list of libraries
12       - "libgs.so"
13     permissions:
14       readonly:
15         - "/var/lib/ghostscript"
16         - "/dev/urandom"
17       readwrite:
18           # folder containing input/output files in argv
19         - "$ARGV_FOLDERS"
20         - "/tmp"
21       network: none
```

Figure 7.8: Configuration file for ImageMagick `convert`. The permissions given by the user are highlighted in green. The red lines specify a custom heap implementation.

This high degree of automation is a major benefit of CALI, which addresses important aspects left open by others. To evaluate if this promised automation also holds in practice, we gave our prototype and documentation to two students: an undergraduate and a grad student in Computer Science. They were tasked to isolate four programs (dpkg, xz, socat, Filezilla) without further assistance. We made sure that the students did not know the program internals (e.g., source code) before handing out the tasks. They correctly isolated previously unknown programs in ≤ 45 min per program and in about 32 min on average. That is, after obtaining the source code, they obtained a well-isolated compiled program in about half an hour. The vast majority of this time was spent on developing and testing a sound permission set, which can be completely automated [47, 32, 46, 190]. This is a great improvement compared to RLBox ("a bit over two days" by developers with program knowledge). Most other related work did not evaluate the necessary human effort.

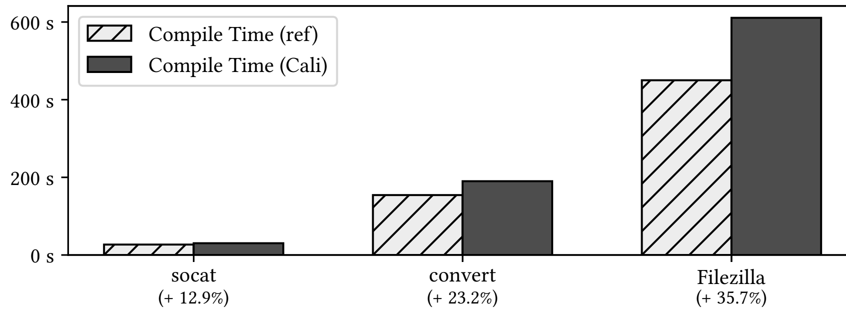


Figure 7.9: Compile time without and with Cali.

Table 7.1: Remaining shared memory allocations and the number of specialized functions in the main program.

Program	Shared Mem.	Mem. Chunks (shared/all)	Specialized functions	Increased code size
ImageMagick	0.078%	28 / 36101	67 / 5395	225 KB (+ 4.7%)
Socat	0.396%	15 / 3787	3 / 748	187 KB (+ 56.2%)
Filezilla	0.255%	48 / 18798	15 / 12348	186 KB (+ 2.8%)

7.8.3 Compilation and Size Overhead

Next, we evaluate the compilation overhead induced by the graph analyses of CALI, as shown in Figure 7.9 for our three sample applications. We compiled every application ten times from a clean source directory and measured the median compilation time of a pure LLVM-based build and a CALI-protected build. CALI adds 12–36% compilation time, usually just a few seconds, up to a few minutes, even for large projects such as Filezilla. In times where build servers are common, this overhead does not impede wide deployment.

The size of the protected binaries increases compared to the original version. After stripping, the protected binaries are around 186 KB to 225 KB larger (see Table 7.1), which is mainly because of our statically linked IPC library (up to 212 KB). For typical x86 architectures, a few hundred KB are no issue; in theory, we could also use a shared library.

We conclude that CALI is easily applicable to protect applications, even without detailed source code knowledge. Binaries produced by CALI do not change behavior and run in every context the unprotected binary would also run.

7.8.4 Security Evaluation

To assess the degree of security CALI provides, we have to answer two questions: (1) Is the compartmentalization and its compartment privilege system strong enough? (2) Under which circumstances can an attacker escape from a compartment?

The compartment privilege system is strong enough to prevent any influences of an attacker beyond the runtime of the program. The PID namespace ensures that all processes

spawned by the library compartment are killed on program termination. Containment policies can usually either revoke file system access of libraries or confine access to subparts only. Hence, neither sensitive data can be accessed nor persistent backdoors can be installed. Furthermore, attackers can only leak data if network operations are allowed. We enforce strict security policies on the isolated libraries:

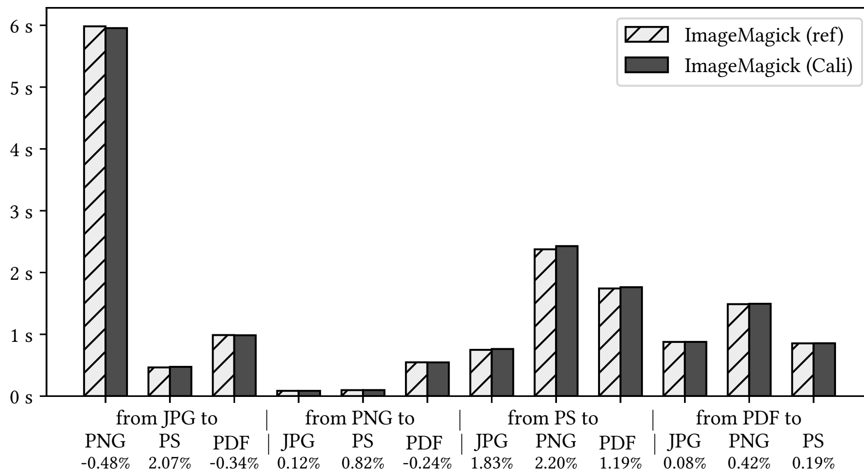
Ghostscript inside ImageMagick gets write access only to the folder with the input/output files (as named in the command line parameters) and the temporary directory `/tmp` (see Section 7.8.2). Additional read-only access is granted on its installation directory and `/dev/urandom`. No network communication is permitted. This isolation is quite close to the minimal required privileges: an attacker exploiting a library vulnerability can only tamper with files in the same folder as the output file. Our permission configuration file is shown in Figure 7.8.

OpenSSL inside socat can only read files given in command line parameters (e.g., certificate and private key) and the randomness devices. Nothing is writeable: an attacker exploiting OpenSSL cannot trigger any permanent changes on the system. We can even block general network access for this library because the program passes the file descriptor of an open socket to the library. The provided socket is the only network communication possibility of OpenSSL. Attackers can still access the certificate's private key (which the library must know in order to work) but have only very limited possibilities to leak it.

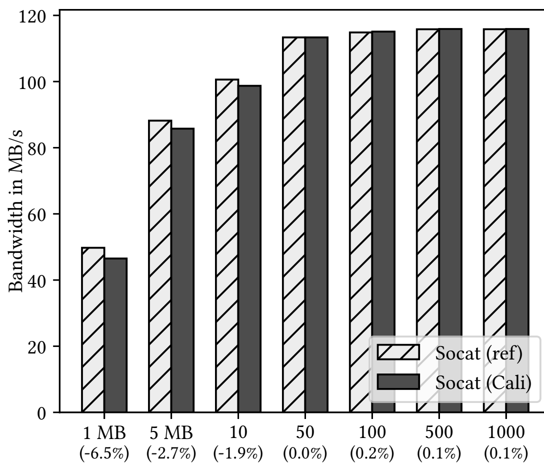
SQLite inside Filezilla can only access files in Filezilla's configuration folder. Network access can even be fully forbidden. An attacker can only mess with Filezilla's config but cannot do any further harm. The threat surface is thus minimal.

Attackers are further tightly bounded when they aim to leak information from shared memory or modify critical data structures in shared memory. Table 7.1 shows that CALI greatly reduces the number of memory allocations in the program that actually produce memory shared between the program and the library. Only a tiny fraction (< 0.4%) of all memory allocations produce chunks that are accessible to the library. Most of these memory allocations are essential to keep the program's functionality; thus, the information would have been passed to the library anyway.

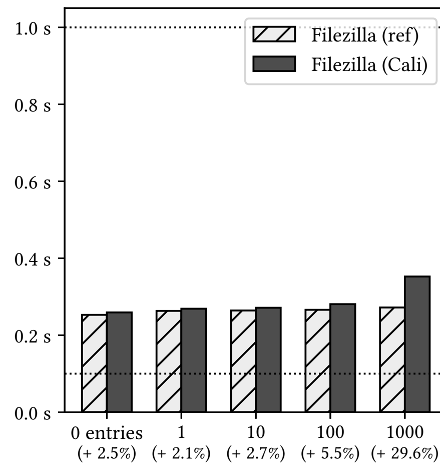
A limitation of our fully-automatic approach is the remaining risk of cross-compartment exploits: The exposed interface between privileged and unprivileged contexts might be vulnerable, e.g., if an attacker could store invalid values in shared memory or trigger callbacks with unexpected parameters or in an unexpected order. RLBox [124] approached this risk by proposing a restrictive C++ type system and requiring the developer to rewrite his source code to adjust to this type system. While protecting from cross-compartment exploits, this tedious work (multiple days) impedes usability, especially if users (e.g., repository maintainers) are largely unfamiliar with the source code. In contrast, CALI provides fewer security guarantees against cross-compartment exploits. Having said this, CALI already handles code pointers like callbacks as function arguments or return values (see Section 7.6.4), such that they cannot be abused to invoke arbitrary code execution in the privileged process. Furthermore, CALI detects and warns about function pointers in shared memory. All evaluated programs (including programs from popularity contest) either have no function pointers in shared memory, or these pointers are only called from within the unprivileged context—which is uncritical. In general, CFI can be used to harden against cross-compartment exploits, including a viable protection for C++ objects [193, 194]. CALI also provides two optional



(a) Runtime of ImageMagick `convert`. The isolated libraries are called to read or write PS/PDF files, conversions between JPG and PNG are affected only by startup overhead.



(b) `socat` file transfer bandwidth over TLS (note: higher is better in this case)



(c) Filezilla startup time, depending on transfer queue size. Dotted lines are noticeable delay (100 ms) and interrupting delay (1000 ms).

Figure 7.10: Performance impact of Cali.

protections against double-fetch bugs (see Section 7.6.6). Other work [62, 152, 192, 194] already suggests protections against different other similar bugs, which are compatible on our approach. We thus see CALI in the sweet spot between usability/automation and security guarantees. CALI already minimizes the risked interface (< 0.4% of memory is shared) and detects critical function pointers in the shared area—other improvements are left open to future work.

7.8.5 Performance Evaluation

Wider adoptions of a protection can only happen when its runtime overhead is negligible (typically below 5%–10% [164, 100]). We evaluate if CALI is fast enough for wider deployment.

All experiments were performed on an Intel Core i7-9700K CPU (8×3.6 GHz, no hyper-threading) with 64 GB of RAM. We use Ubuntu version 18.04 LTS with an unmodified Linux 5.4 kernel. Our compiler toolchain is clang/LLVM version 7. To avoid any influences on the benchmark, we set the CPU governor to "performance" and disable "turbo" CPU power states as well as ASLR. We force each program to use only one CPU core to prevent any unfair advantage our protected version might get; we use cpuset to ensure this CPU core is reserved exclusively for the program. If not stated, the standard deviation of the results was below 1% of the median.

Microbenchmarks: We attribute the runtime overhead that CALI introduces mainly to two factors. First, at startup, the protected program needs to set up additional data structures and start the library compartment process. Second, every call to the library forces the kernel to switch between the two processes twice (call and return). We measured these two overheads in a minimal program as a microbenchmark. Initialization at startup takes 2.2 ms on average. In these micro benchmarks, our compartment handles around 323,000 calls/second (3.1 µs overhead per library call). With the mprotect-based concurrency protection enabled, our overhead is 8 µs per library call.

Application benchmarks: We now evaluate the runtime overhead of CALI on our real-world applications. We measure the runtime of ImageMagick `convert` while converting four different image formats into each other. We chose JPG, PNG, PDF, and PostScript. As input files, we picked a camera picture (JPG), a website background (PNG), a test pattern (PostScript), and a sample PDF from W3C. When converting to pixel-based formats, we use a density of 300 dpi, a default value for printing. We run each conversion 100 times. Figure 7.10a shows the median conversion time. The runtime overhead is between 0% and 2.2% (geometric mean: 0.65%).

Next, we measure the network throughput of an encrypted TLS connection using `socat`, where OpenSSL is put in a compartment. Every packet is encrypted or decrypted in the isolated library, while the data is processed by the main program (resulting in ∼42,000 library calls per second). We transmit files with varying sizes over a common 1 Gbps local network. The TLS server answers with the content it receives (`echo`), so we test with symmetric up- and downstream. We run each experiment 100 times and take the median throughput, i.e., file size divided by transmission time. Our network throughput showed a higher standard deviation (around 3%) for short connections (files up to 10 MB). To avoid imprecise values, we repeated the affected experiments 1000 times. Figure 7.10b shows the achieved throughput. For files up to 10 MB, the connection cannot be fully saturated, neither with protected nor unprotected `socat`. The reasons are the TLS handshake, the TCP slow start algorithm, and application startup time. CALI adds ∼2 ms to the application startup, resulting in a throughput degradation of 1.9% to 6.5%. For large files or long-lived connections that face the initialization overhead just once, CALI's impact on network throughput is almost negligible (less than 0.1%).

Finally, we benchmark the impact of CALI on Filezilla, the only GUI program in our test setting. Filezilla uses SQLite to manage the download queue and server settings. The

runtime performance overhead at startup is highest since every single entry in the database is read. This leads to many library invocations, as SQLite induces one library call per table row and, for each row, one call per cell. We enqueue 0–1000 downloads in Filezilla and measure the startup time until the main window is fully displayed (`CMainFrame::OnActivate`). We repeat each experiment 500 times, as Filezilla's startup time shows a higher standard deviation (up to 20 ms). Figure 7.10c shows the median start times. Filezilla takes around 256 ms to start, and CALI adds 5 ms–80 ms depending on the queue size. With typical queue sizes up to 100 entries, the overhead is ≈3.2%. For an empty queue, the overhead is slightly higher, mainly due to the fact that there is less file I/O, and thus the initialization overhead becomes more prominent.

The user experience does not change, as the Filezilla startup response delay is already above 100 ms (lower dotted horizontal line) and thus noticeable, and even the protected case is way below the time that interrupts workflows (1000 ms, upper dotted horizontal line) according to the literature [107, 127].

While it is common to choose a standardized benchmark like SPEC CPU [161, 160] to allow for comparative evaluations, we cannot do so in our case. No program in the CPU benchmark uses third-party libraries.

7.9 Conclusion

CALI protects applications from vulnerabilities and backdoors in third-party libraries. CALI does not assume *a priori* expert knowledge of the program's source code and does not require source code changes. Its compartmentalization can be easily integrated into common build processes. Programs compiled with CALI are fully portable and do not require additional CPU features, OS modifications or superuser privileges. Isolated libraries can only access small, non-sensitive portions of the main program's memory (up to 0.4% in our examples), and only selective system access permissions remain.

Next to its primary use case, CALI can also separate different components *within* a program. Developers can use this feature to split their application into least-privileged components—fully transparently by recompiling their program with CALI. In fact, we support more source code languages than just C and C++. The underlying LLVM bytecode is independent of the source code language, and programs in other languages with LLVM frontend (Delphi, Rust, Go, Swift, and many more) could be separated by CALI with minimal adaptations.

Availability

Our prototype has been released as Open-Source Software; it is available on Github:
`https://github.com/cali-library-isolation/Cali-library-isolation`

8
Conclusion

Memory corruption attacks on C/C++ programs have been an unsolved problem for a long time. Although researchers proposed many countermeasures, attackers can still divert the control flow of applications and execute arbitrary code. With arbitrary code execution, attackers can compromise the vulnerable application's whole system with the permissions of the user that ran the application.

This thesis proposes practicable countermeasures against this problem that protect applications and systems against these threats. In particular, we answered two research questions:

(RQ1) How can we prevent the escalation from memory corruption to attacks on an application's control flow? Section 3.6 detailed control flow protections further. To preserve the integrity of an application's control flow, we must protect backward control flow and forward control flow, which is different for C and C++ applications. We analyzed existing backward CFI schemes in Chapter 4. After reviewing stack canaries, shadow stacks, and return address encryption on modern systems, we found that existing protections are applicable for real-world deployment. In particular, we can suggest further adoption of shadow stacks, which have a mean overhead of only 2.7%—much lower than expected. We also proposed two new forward CFI schemes. NoVT from Chapter 5 protects C++ virtual dispatch with negligible to negative overhead. TyPro from Chapter 6 protects C indirect calls with negligible overhead, even supporting dynamic linking. If combined, these two solutions cover forward CFI in C/C++ programs completely. If we combine both solutions with shadow stacks and the already existing RELRO mode (see Section 3.4), we can produce applications that cover the integrity of each control flow. The only remaining exception is inline assembly, which is so rare that it can be protected manually. In Chapter 6 and related work [S1], we already did this for musl libc.

All presented schemes are compatible with existing source code. Developers can apply the protecting compilers without any change to their C/C++ application, as long as it is already compatible with Clang. While implemented for Linux only, the methodologies are not dependent on a specific operating system or CPU architecture. The performance impact of our protections is also negligible—with the proposed protections, developers can get more secure software without drawbacks or manual effort. We expect this attractive offer will get more developers to use modern CFI systems and deliver more secure applications.

However, there might be situations where applying all tools is not possible, mainly when third-party libraries are involved. The tools mentioned above cannot recompile a protected version if no source code is available, just-in-time compilers can introduce unprotected code, and malicious code in supply-chain attacks might be designed to circumvent CFI's protection. Therefore, we asked the second research question:

(RQ2) How can we isolate application memory and underlying system from attackers that exploit a code execution vulnerability in a library? Program compartmentalization (see Section 3.7) is a known defense that prevents system corruption in case of attacks. However, this concept has been hard to apply because it has required manual effort and more profound knowledge of both application and isolation mechanisms. We solve this and other problems with CALI in Chapter 7. We provide automatic library isolation that requires minimal developer effort. Developers only specify the permissions that a library should have; CALI infers everything else. In particular, no source code knowledge or modification is necessary. While not as automated as our other solutions, we showed in our experiments

from Section 7.8.2 that even unfamiliar developers can quickly isolate libraries with a sound permission configuration. While CALI's performance overhead depends on the frequency of library interaction, we have shown that it is acceptable in many situations in the real world.

There is a remaining threat surface that compartmentalization cannot solve in general. Some libraries might require access to parts of the system. These parts might contain sensitive data. In rare cases, corruption of these parts is enough to compromise the user account, for example, the `.bashrc` file. Compartmentalization, in general, cannot differ if the library accesses such files on its behalf or because an attacker exploited it. Thus, we cannot prevent the misuse of existing library privileges. This issue is not CALI-specific—all library compartmentalization solutions face this problem. Further research in this direction is required: for example, solutions could drop some privileges after startup when they are not needed anymore.

To summarize, we have presented three tools (NOVT, TYPRO, and CALI) that protect C/C++ programs against different threats originating from memory corruption vulnerabilities. All tools are compatible with existing software, easy to apply for developers, and have a little to negligible performance impact. If extended with the existing, re-evaluated protections, they offer a strong level of protection for legacy applications. We released all tools as free and open-source software ready to use.

Further Research Directions

More engineering effort can improve the quality and the experience for developers that use our solutions in practice. First, NOVT and TYPRO could be combined into a single tool with a shared analysis. Switchpoline [S1] solves parts of this issue already. Next, one could add compatibility with unprotected code, making step-by-step adoption possible. While both NOVT and TYPRO already have some level of compatibility, if the developer configures it, this could be automated and extended. Finally, continuous maintenance is necessary to keep the tools up to date with compiler development.

Future compilers could combine CFI schemes with memory safety schemes. For example, a sound static analysis can check if a piece of C/C++ code is memory-safe or potentially vulnerable. In the next step, additional protection, such as CFI or isolation, could only be applied to potentially unsafe parts. Furthermore, compilers could use analysis techniques such as symbolic execution to determine conditions for a program's memory safety. At runtime, one could determine if an input meets these conditions: if it is safe, a fast, unprotected version of the program executes, otherwise, a hardened but slower version executes.

Further research on library isolation could help automate CALI's configuration step. For example, new tools could automatically infer the minimal necessary permissions of a piece of code, which in turn could be used to determine optimal isolation bounds and compartment configurations. More research on cross-compartment exploits is necessary to increase the security level of library isolation. For example, an automated version of RLBox's [124] interface sanitation could be invented.

Library isolation could be improved using artificial intelligence. AI systems could monitor the system access patterns of isolated code and detect and block unusual or suspicious accesses. Isolation could improve AI precision by ensuring that system accesses of different modules are distinguishable. While one can hardly guarantee stability in such a context, AI

could solve the issue of misused library privileges.

The presented solutions protect C and C++ applications only. Other memory-unsafe languages like Pascal and Fortran have similar problems that might be addressed. Even if a C/C++ application is built of protected code only, it can contain just-in-time compilers that produce unsafe and unprotected code at runtime. Many languages and ecosystems use just-in-time compilers, e.g., PHP, JavaScript, WebAssembly, and the JVM. While some protections already exist [130, 207], future research is necessary to provide a similar level of protection for just-in-time compiled code. Nevertheless, the interface between native and runtime-generated code needs protection because it is based on potentially dangerous indirect branches. More research on these topics will close the gap further toward fully protected legacy applications.

Finally, memory-safe languages like Rust can solve the underlying issue of all attacks presented in this thesis: memory corruption vulnerabilities. Further research could provide automated tools that help developers to adopt memory-safe languages in their projects. For example, researchers could propose methods that support developers when rewriting or extending existing applications, lowering the cost of memory-safe programming. In particular the interaction between Rust and legacy C/C++ code might need additional protection: researchers could find methods to ensure that this interaction does not break Rust's security guarantees. Also, researchers could isolate unverified C code in Rust applications similar to CALI. These methods can support incremental deployment of Rust in legacy applications, slowly replacing memory-unsafe code.

Finally, when memory-safe languages have taken the place that C and C++ have today, then memory corruption vulnerabilities and code execution attacks might finally vanish. Until then, this thesis provides viable mitigations.

Bibliography

Author's Papers for this Thesis

[P1] **M. Bauer** and C. Rossow. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In: *2021 IEEE European Symposium on Security and Privacy*. EuroS&P '21. Sept. 2021. DOI: 10.1109/EuroSP51992.2021.00049.

[P2] **M. Bauer**, I. Grishchenko, and C. Rossow. TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation. In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC '22. Dec. 2022. DOI: 10.1145/3564625.3564627.

[P3] **M. Bauer** and C. Rossow. Cali: Compiler Assisted Library Isolation. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. May 2021. DOI: 10.1145/3433210.3453111.

Other Papers of the Author

[S1] **M. Bauer**, L. Hetterich, C. Rossow, and M. Schwarz. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In: *Proceedings of the 2024 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. 2024.

Other references

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. 2005. DOI: 10.1145/1102120.1102165.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In: *2008 IEEE Symposium on Security and Privacy*. SP '08. 2008. DOI: 10.1109/SP.2008.30.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In: *18th USENIX Security Symposium*. SEC '09. 2009.

[4] P. Akritidis, Niometrics, Singapore, and U. of Cambridge. Cling: A Memory Allocator to Mitigate Dangling Pointers. In: *19th USENIX Security Symposium*. SEC '10. Aug. 2010.

[5] B. Allombert. *Debian Popularity Contest*. 2020. URL: https://popcon.debian.org/stable/by_vote.

[6] N. Almakhdhub, A. Clements, S. Bagchi, and M. Payer. μRAI: Securing Embedded Systems with Return Address Integrity. In: *27th Annual Network and Distributed System Security Symposium*. NDSS '20. Jan. 2020. DOI: 10.14722/ndss.2020.24016.

[7] arm. *Armv8.5-A—Memory Tagging Extension*. 2019. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[8] M. Backes and S. Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In: *23rd USENIX Security Symposium*. SEC '14. Aug. 2014.

[9] S. Bhatkar and R. Sekar. Data Space Randomization. In: *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by D. Zamboni. DIMVA '08. 2008. DOI: 10.1007/978-3-540-70542-0_1.

[10] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813691.

[11] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In: *5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '08. 2008.

[12] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In: *Fields of Logic and Computation II*. Springer, 2015. DOI: 10.1007/978-3-319-23534-9.

[13] D. Bounov, R. G. Kici, and S. Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In: *23rd Annual Network and Distributed System Security Symposium*. NDSS '16. 2016. DOI: 10.14722/ndss.2016.23421.

[14] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In: *13th USENIX Security Symposium*. SEC '04. 2004.

[15] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. 2008. DOI: 10.1145/1455770.1455776.

[16] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys* (Apr. 2017). DOI: 10.1145/3054924.

[17] N. Burow, D. McKee, S. A. Carr, and M. Payer. CFIXX: Object Type Integrity for C++. In: *25th Annual Network and Distributed System Security Symposium*. NDSS '18. 2018. DOI: 10.14722/ndss.2018.23279.

[18] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In: *24th USENIX Security Symposium*. SEC '15. Aug. 2015.

[19] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In: *23rd USENIX Security Symposium*. SEC '14. 2014.

[20] Carnegie Mellon University. *Ghostscript contains multiple -dSAFER sandbox bypass vulnerabilities*. 2018. URL: https://www.kb.cert.org/vuls/id/332928/.

[21] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In: *7th Symposium on Operating Systems Design and Implementation*. OSDI '06. 2006.

[22] J. Chen and J. Revels. Robust benchmarking in noisy environments (2016). arXiv: 1608.04295.

[23] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-Grained Execution Units with Private Memory. In: *2016 IEEE Symposium on Security and Privacy*. SP '16. May 2016. DOI: 10.1109/SP.2016.12.

[24] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. *Itanium C++ ABI*. 2021. URL: https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

[25] J. Corbet. *Using Rust for kernel development*. Sept. 2021. URL: https://lwn.net/Articles/870555/.

[26] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In: *12th USENIX Security Symposium*. SEC '03. Aug. 2003.

[27] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In: *7th USENIX Security Symposium*. SEC '98. 1998.

[28] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813682.

[29] T. H. Dang, P. Maniatis, and D. Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. 2015. DOI: 10.1145/2714576.2714635.

[30] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge Me If You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for X86 and ARM. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. 2013. DOI: 10.1145/2484313.2484351.

[31] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In: *9th European Conference on Object-Oriented Programming*. ECOOP '95. Aug. 1995.

[32] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '20. Oct. 2020.

[33] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. 2006. DOI: 10.1145/1134285.1134309.

[34] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. Efficient Protection of Path-Sensitive Control Security. In: *26th USENIX Security Symposium*. SEC '17. Aug. 2017.

[35] J. Edge. *"Strong" stack protection for GCC [LWN.net]*. 2014. URL: https://lwn.net/Articles/584225/.

[36] M. Elsabagh, D. Fleck, and A. Stavrou. Strict Virtual Call Integrity Checking for C++ Binaries. In: *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '17. 2017. DOI: 10.1145/3052973.3052976.

[37] H. Etoh and K. Yoda. Protecting from stack smashing attacks (Jan. 2000).

[38] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813646.

[39] X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '17. 2017. DOI: 10.1145/3092703.3092729.

[40] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987). DOI: 10.1145/24039.24041.

[41] A. Flores-Montoya and E. Schulte. Datalog Disassembly. In: *29th USENIX Security Symposium*. SEC '20. Aug. 2020.

[42] Free Software Foundation. *GNU Lesser General Public License (LGPL)*. Aug. 2002. URL: http://www.gnu.org/licenses/lgpl.html.

[43] M. Friedl, N. Provos, T. de Raadt, K. Steves, D. Miller, D. Tucker, J. McIntyre, T. Rice, and B. Lindstrom. *OpenSSH Release 5.9*. Sept. 2011. URL: https://www.openssh.com/txt/release-5.9.

[44] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACSAC '14. 2014. DOI: 10.1145/2664243.2664249.

[45] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In: *ACM SIGPLAN Notices*. ASPLOS '17. 2017. DOI: 10.1145/3037697.3037716.

[46] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '20. Oct. 2020.

[47] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis. Temporal System Call Specialization for Attack Surface Reduction. In: *29th USENIX Security Symposium*. SEC '20. Aug. 2020.

[48] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In: *21st USENIX Security Symposium*. SEC '12. Aug. 2012.

[49] Google. *google/sandboxed-api*. Aug. 2019. URL: https://github.com/google/sandboxed-api.

[50] Google. *nsjail*. 2021. URL: https://nsjail.dev/.

[51] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In: *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. 2017. DOI: 10.1145/3029806.3029830.

[52] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory. In: *2015 IEEE Symposium on Security and Privacy*. SP '15. 2015. DOI: 10.1109/SP.2015.8.

[53] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean Application Compartmentalization with SOAAP. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813611.

[54] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to Speed with WebAssembly. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '17. 2017. DOI: 10.1145/3062341.3062363.

[55] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection without Loose Ends. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC '15. 2015. DOI: 10.1145/2818000.2818025.

[56] HardenedBSD. *HardenedBSD - Introducing CFI*. 2022. URL: https://hardenedbsd.org/article/shawn-webb/2017-03-02/introducing-cfi.

[57] G. Heiser. *Gernot's List of Systems Benchmarking Crimes*. 2010. URL: https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html.

[58] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News* 34, 4 (Sept. 2006). DOI: 10.1145/1186736.1186737.

[59] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In: *2012 IEEE Symposium on Security and Privacy*. SP '12. 2012. DOI: 10.1109/SP.2012.39.

[60] A. Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic* 16, 1 (1951). DOI: 10.2307/2266412.

[61] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. 2016. DOI: 10.1145/2976749.2978327.

[62] H. Hu, Z. L. Chua, Z. Liang, and P. Saxena. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In: *Computer Security – ESORICS 2015*. 2015. DOI: 10.1007/978-3-319-24177-7_16.

[63] 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: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018. DOI: 10.1145/3243734.3243797.

[64] X. Hu and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In: *22nd Annual Network and Distributed System Security Symposium*. NDSS '15. Jan. 2015. DOI: 10.14722/ndss.2015.23297.

[65] J. Huang, O. Schranz, S. Bugiel, and M. Backes. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. 2017. DOI: 10.1145/3133956.3134064.

[66] Intel. *Intel® Architecture Instruction Set Extensions and Future Features*. May 2021. URL: https://www.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html.

[67] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, Dec. 2011. URL: https://www.iso.org/standard/57853.html.

[68] ISO. *ISO/IEC 14882:2020: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Dec. 2020. URL: https://www.iso.org/standard/79358.html.

[69] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In: *21st Annual Network and Distributed System Security Symposium*. NDSS '14. 2014. DOI: 10.14722/ndss.2014.23287.

[70] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In: *2002 USENIX Annual Technical Conference*. ATC '02. June 2002.

[71] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On Synthesis of Program Analyzers. In: *Computer Aided Verification*. Ed. by S. Chaudhuri and A. Farzan. CAV '16. 2016.

[72] P. Kehrer. *Memory Unsafety in Apple's Operating Systems*. July 2019. URL: `https://langui.sh/2019/07/23/apple-memory-safety/`.

[73] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng. Adaptive Call-Site Sensitive Control Flow Integrity. In: *2019 IEEE European Symposium on Security and Privacy*. EuroS&P '19. 2019. DOI: `10.1109/EuroSP.2019.00017`.

[74] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang. Origin-sensitive Control Flow Integrity. In: *28th USENIX Security Symposium*. SEC '19. Aug. 2019.

[75] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In: *Proceedings of the 22nd Annual Computer Security Applications Conference*. ACSAC '06. 2006. DOI: `10.1109/ACSAC.2006.9`.

[76] D. Kilpatrick. Privman: A Library for Partitioning Applications. In: *2003 USENIX Annual Technical Conference*. ATC '03. June 2003.

[77] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida. SoK: Benchmarking Flaws in Systems Security. In: *2019 IEEE European Symposium on Security and Privacy*. EuroS&P '19. 2019. DOI: `10.1109/EuroSP.2019.00031`.

[78] E. van der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. 2018. DOI: `10.1145/3274694.3274705`.

[79] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In: *11th USENIX Conference on Operating Systems Design and Implementation*. OSDI '14. 2014.

[80] M. Larabel. *Tesseract OCR Benchmark - OpenBenchmarking.org*. 2022. URL: `https://openbenchmarking.org/test/system/tesseract-ocr`.

[81] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. In: *2014 IEEE Symposium on Security and Privacy*. SP '14. 2014. DOI: `10.1109/SP.2014.25`.

[82] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. 2005. DOI: `10.1145/1065010.1065027`.

[83] H. Lee, C. Song, and B. B. Kang. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018. DOI: `10.1145/3243734.3243748`.

[84] V. Lextrait. *The Programming Languages Beacon*. Sept. 2022. URL: `https://www.mentofacturing.com/Vincent/implementations.html`.

[85] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu. Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by J. Ligatti, X. Ou, J. Katz, and G. Vigna. CCS '20. 2020. DOI: `10.1145/3372297.3417867`.

[86] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In: *28th USENIX Security Symposium*. SEC '19. Aug. 2019.

[87] Linux Authors. *cpuset(7) - Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man7/cpuset.7.html.

[88] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight Contexts: An OS Abstraction for Safety and Performance. In: *12th USENIX Conference on Operating Systems Design and Implementation*. OSDI '16. 2016.

[89] S. Liu, G. Tan, and T. Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. 2017. DOI: 10.1145/3133956.3134066.

[90] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan. Program-Mandering: Quantitative Privilege Separation. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. 2019. DOI: 10.1145/3319535.3354218.

[91] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813690.

[92] LLVM Project. *LLVM Language Reference Manual - LLVM 10 documentation*. 2019. URL: https://llvm.org/docs/LangRef.html.

[93] LLVM Project. *Benchmarking tips - LLVM 15.0.0git documentation*. 2022. URL: https://llvm.org/docs/Benchmarking.html.

[94] K. Lu, M. Backes, S. Nürnberger, and W. Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In: *23rd Annual Network and Distributed System Security Symposium*. NDSS '16. Feb. 2016. DOI: 10.14722/ndss.2016.23173.

[95] K. Lu and H. Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. 2019. DOI: 10.1145/3319535.3354244.

[96] J. Magee. *[cfe-dev] What do the different stack-protector levels protect in Clang?* 2017. URL: https://lists.llvm.org/pipermail/cfe-dev/2017-April/053662.html.

[97] G. Maisuradze, M. Backes, and C. Rossow. What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In: *25th USENIX Security Symposium*. SEC '16. Aug. 2016.

[98] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813676.

[99] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In: *2010 IEEE Symposium on Security and Privacy*. SP '10. 2010. DOI: 10.1109/SP.2010.17.

[100] Microsoft. *The BlueHat prize contest official rules*. 2012. URL: https://web.archive.org/web/20141111145816/http://www.microsoft.com/security/bluehatprize/rules.aspx.

[101] Microsoft. *Control Flow Guard - Win32 apps*. 2022. URL: https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[102] Microsoft. *Data Execution Prevention*. Feb. 2022. URL: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[103] Microsoft Corporation. *Microsoft Security Intelligence Report vol.16*. 2013. URL: http://download.microsoft.com/download/7/2/b/72b5de91-04f4-42f4-a587-9d08c55e0734/microsoft_security_intelligence_report_volume_16_english.pdf.

[104] A. Milanova, A. Rountev, and B. G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engineering* 11 (2004). DOI: 10.1023/B:AUSE.0000008666.56394.a1.

[105] A. Milburn, E. van der Kouwe, and C. Giuffrida. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In: *2022 IEEE Symposium on Security and Privacy*. SP '22. May 2022. DOI: 10.1109/SP46214.2022.00016.

[106] M. R. Miller and K. D. Johnson. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities. US 2012/0144480 A1. Microsoft Corporation. Patent number US 2012/0144480 A1. June 7, 2012.

[107] R. B. Miller. Response Time in Man-computer Conversational Transactions. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68. 1968. DOI: 10.1145/1476589.1476628.

[108] MITRE Corporation. *Artifex Ghostscript : Security Vulnerabilities*. 2019. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-10846/product_id-36469/Artifex-Ghostscript.html.

[109] MITRE Corporation. *CVE-2019-5018*. 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5018.

[110] MITRE Corporation. *Openssl : Security Vulnerabilities*. 2019. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-217/product_id-383/Openssl-Openssl.html.

[111] MITRE Corporation. *Sqlite : Security Vulnerabilities*. 2019. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-9237/Sqlite.html.

[112] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque Control-Flow Integrity. In: *22nd Annual Network and Distributed System Security Symposium*. NDSS '15. Feb. 2015. DOI: 10.14722/ndss.2015.23271.

[113] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*. Ed. by C. R. Ramakrishnan and J. Rehof. TACAS '08. 2008. DOI: 10.1007/978-3-540-78800-3_24.

[114] Mozilla. *Dromaeo: Javascript Performance Testing*. 2020. URL: https://wiki.mozilla.org/Dromaeo.

[115] Mozilla. *Kraken JavaScript Benchmark (version 1.1)*. 2020. URL: https://wiki.mozilla.org/Kraken.

[116] MozillaWiki Team. *Oxidation*. Nov. 2020. URL: https://wiki.mozilla.org/Oxidation.

[117] MozillaWiki Team. *Security/Sandbox*. Aug. 2020. URL: https://wiki.mozilla.org/Security/Sandbox.

[118] M. Muench, F. Pagani, Y. Shoshitaishvili, C. Kruegel, G. Vigna, and D. Balzarotti. Taming Transactions: Towards Hardware-Assisted Control Flow Integrity Using Transactional Memory. In: *19th International Symposium on Research in Attacks, Intrusions, and Defenses*. RAID '16. Springer. Sept. 2016. DOI: 10.1007/978-3-319-45719-2_2.

[119] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. Analyzing Control Flow Integrity with LLVM-CFI. In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC '19. 2019. DOI: 10.1145/3359789.3359806.

[120] musl authors. *musl libc*. 2022. URL: https://musl.libc.org/.

[121] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Notices* 44, 3 (Mar. 2009). DOI: 10.1145/1508284.1508275.

[122] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *SIGPLAN Notices* 44, 6 (June 2009). DOI: 10.1145/1543135.1542504.

[123] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10. 2010. DOI: 10.1145/1806651.1806657.

[124] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In: *29th USENIX Security Symposium*. SEC '20. Aug. 2020.

[125] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems* 27, 3 (May 2005). DOI: 10.1145/1065887.1065892.

[126] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. 2007. DOI: 10.1145/1250734.1250746.

[127] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[128] B. Niu and G. Tan. Enforcing User-space Privilege Separation with Declarative Architectures. In: *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing*. STC '12. 2012. DOI: 10.1145/2382536.2382541.

[129] B. Niu and G. Tan. Modular Control-Flow Integrity. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. 2014. DOI: 10.1145/2594291.2594295.

[130] B. Niu and G. Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. 2014. DOI: 10.1145/2660267.2660281.

[131] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813644.

[132] Octane Team Google. *Octane 2.0*. 2020. URL: https://chromium.github.io/octane/.

[133] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. 2010. DOI: 10.1145/1920261.1920269.

[134] A. One. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (Nov. 1996). http://www.phrack.com/issues.html?issue=49&id=14.

[135] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon? In: *2018 IEEE International Symposium on High Performance Computer Architecture*. HPCA '18. 2018. DOI: 10.1109/HPCA.2018.00032.

[136] B. V. Patel. *A Technical Look at Intel's Control-flow Enforcement Technology*. June 2020. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[137] PaX. *Address Space Layout Randomization*. Sept. 2004. URL: https://pax.grsecurity.net/docs/aslr.txt.

[138] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In: *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA '15. 2015. DOI: 10.1007/978-3-319-20550-2_8.

[139] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav. Compiler Optimization of C++ Virtual Function Calls. In: *USENIX Conference on Object-Oriented Technologies*. COOTS '96'. June 1996.

[140] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In: *12th USENIX Security Symposium*. SEC '03. 2003.

[141] W. Qiang, Y. Cao, W. Dai, D. Zou, H. Jin, and B. Liu. Libsec: A Hardware Virtualization-Based Isolation for Shared Library. In: *19th IEEE International Conference on High Performance Computing and Communications; 15th IEEE International Conference on Smart City; 3rd IEEE International Conference on Data Science and Systems*. HPCC/SmartCity/DSS '17. Dec. 2017. DOI: 10.1109/HPCC-SmartCity-DSS.2017.5.

[142] Qualcomm Product Security. *Pointer Authentication on ARMv8.3*. 2017. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[143] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. 2020. DOI: 10.1145/3320269.3384757.

[144] Red Hat. *Position Independent Executables (PIE)*. Nov. 2012. URL: https://www.redhat.com/en/blog/position-independent-executables-pie.

[145] Redis Authors. *Redis benchmark*. 2022. URL: https://redis.io/docs/reference/optimization/benchmarks/.

[146] G. Rodola. *pyftpdlib/ftpbench*. 2016. URL: https://github.com/giampaolo/pyftpdlib/blob/master/scripts/ftpbench.

[147] Rust Team. *Rust Programming Language*. 2022. URL: https://www.rust-lang.org/.

[148] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In: *11th Annual Network and Distributed System Security Symposium*. NDSS '04. 2004.

[149] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On Fast Large-Scale Program Analysis in Datalog. In: *Proceedings of the 25th International Conference on Compiler Construction*. CC '16. 2016. DOI: 10.1145/2892208.2892226.

[150] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In: *29th USENIX Security Symposium*. SEC '20. Aug. 2020.

[151] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In: *2015 IEEE Symposium on Security and Privacy*. SP '15. 2015. DOI: 10.1109/SP.2015.51.

[152] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs Using Modern CPU Features. In: *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '18. 2018. DOI: 10.1145/3196494.3196508.

[153] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In: *2012 USENIX Annual Technical Conference*. ATC '12. 2012.

[154] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In: *2005 USENIX Annual Technical Conference*. ATC '05. Apr. 2005.

[155] V. Shanbhogue, D. Gupta, and R. Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. 2019. DOI: 10.1145/3337167.3337175.

[156] H. Sidhpurwala. *Hardening ELF binaries using Relocation Read-Only (RELRO)*. Jan. 2019. URL: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.

[157] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. English. In: *2012 USENIX Annual Technical Conference*. ATC '12. June 2019. DOI: 10.5555/2342821.2342832.

[158] R. Smith. *Extreme Flow Guard (xFG) and Kernel Data Protection (KDP) Coming to Windows 10*. July 2020. URL: https://petri.com/extreme-flow-guard-xfg-and-kernel-data-protection-kdp-coming-to-windows-10.

[159] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and Protecting Dynamic Code Generation. In: *22nd Annual Network and Distributed System Security Symposium*. NDSS '15. 2015. DOI: 10.14722/ndss.2015.23233.

[160] Standard Performance Evaluation Corporation. *SPEC CPU® 2017*. 2017. URL: https://www.spec.org/cpu2017/.

[161] Standard Performance Evaluation Corporation. *SPEC CPU® 2006*. June 2020. URL: https://www.spec.org/cpu2006/.

[162] J. V. Stoep and C. Zhang. *Queue the Hardening Enhancements*. May 2019. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html.

[163] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel Support for Secure Process Compartments. *ICST Transactions on Security Safety* 2 (2015).

[164] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In: *2013 IEEE Symposium on Security and Privacy*. SP '13. 2013. DOI: 10.1109/SP.2013.13.

[165] The apache software foundation. *ab - Apache HTTP server benchmarking tool*. 2022. URL: https://httpd.apache.org/docs/2.4/programs/ab.html.

[166] The Chromium Projects. *Chromium*. 2022. URL: https://www.chromium.org/Home.

[167] The Chromium Projects. *Memory safety*. 2022. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[168] The Chromium Projects. *Secure Architecture*. 2022. URL: https://www.chromium.org/Home/chromium-security/guts/.

[169] The Chromium Projects. *Telemetry: Run Benchmarks Locally*. 2022. URL: https://chromium.googlesource.com/catapult/+/HEAD/telemetry/docs/run_benchmarks_locally.md.

[170] The Clang Team. *Control Flow Integrity - Clang 13 documentation*. 2021. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[171] The Clang Team. *AddressSanitizer*. 2022. URL: https://clang.llvm.org/docs/AddressSanitizer.html.

[172] The Clang Team. *Control Flow Integrity Design Documentation*. 2022. URL: https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.

[173] The Clang Team. *Introduction to the Clang AST - Clang 13 documentation*. Jan. 2022. URL: https://clang.llvm.org/docs/IntroductionToTheClangAST.html.

[174] The Clang Team. *MemorySanitizer*. 2022. URL: https://clang.llvm.org/docs/MemorySanitizer.html.

[175] The Clang Team. *SafeStack*. 2022. URL: https://clang.llvm.org/docs/SafeStack.html.

[176] The Clang Team. *ShadowCallStack - Clang 15.0.0git documentation*. July 2022. URL: https://clang.llvm.org/docs/ShadowCallStack.html.

[177] The Clang Team. *ThreadSanitizer*. 2022. URL: https://clang.llvm.org/docs/ThreadSanitizer.html.

[178] The Clang Team. *UndefinedBehaviorSanitizer*. 2022. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[179] The Flang Team. *Flang 15.0.0 (In-Progress) Release Notes - The Flang Compiler*. 2022. URL: https://flang.llvm.org/docs/.

[180] The kernel development community. *Building Linux with Clang/LLVM*. 2021. URL: https://www.kernel.org/doc/html/latest/kbuild/llvm.html.

[181] The Linux Kernel documentation. *Seccomp BPF (SECure COMPuting with filters)*. 2019. URL: https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

[182] The Santa Cruz Operation. *Relocation*. 2001. URL: https://refspecs.linuxbase.org/elf/gabi4+/ch4.reloc.html.

[183] The SQLite Project. *Most Widely Deployed SQL Database Engine*. 2019. URL: https://www.sqlite.org/mostdeployed.html.

[184] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In: *23rd USENIX Security Symposium*. SEC '14. 2014.

[185] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In: *28th USENIX Security Symposium*. SEC '19. Aug. 2019.

[186] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015. DOI: 10.1145/2810103.2813673.

[187] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In: *2016 IEEE Symposium on Security and Privacy*. SP '16. 2016. DOI: 10.1109/SP.2016.60.

[188] A. van de Ven. *Exec shield*. Aug. 2004. URL: https://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

[189] Vendicator. *Stack Shield: A "stack smashing" technique protection tool for Linux*. 2000. URL: http://www.angelfire.com/sk/stackshield/.

[190] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li. Mining Sandboxes for Linux Containers. In: *2017 IEEE International Conference on Software Testing, Verification and Validation*. ICST '17. 2017. DOI: 10.1109/ICST.2017.16.

[191] J. Wang, X. Xiong, and P. Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In: *2015 USENIX Annual Technical Conference*. ATC '15. 2015.

[192] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In: *26th USENIX Security Symposium*. SEC '17. Aug. 2017.

[193] Y.-P. Wang, X.-Q. Hu, Z.-X. Zou, W. Tan, and G. Tan. IVT: An Efficient Method for Sharing Subtype Polymorphic Objects. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019). DOI: 10.1145/3360556.

[194] W. Wang, X. Xu, and K. W. Hamlen. Object Flow Integrity. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. 2017. DOI: 10.1145/3133956.3133986.

[195] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: *2010 IEEE Symposium on Security and Privacy*. SP '10. May 2010. DOI: 10.1109/SP.2010.30.

[196] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy X86 Binary Code. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. 2012. DOI: 10.1145/2382196.2382216.

[197] Webkit. *JetStream2*. 2020. URL: https://browserbench.org/JetStream/.

[198] Webkit. *Speedometer 2.0*. 2020. URL: https://browserbench.org/Speedometer2.0/.

[199] Webkit. *SunSpider 1.0.2 JavaScript Benchmark*. 2020. URL: https://webkit.org/perf/sunspider/sunspider.html.

[200] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang. Codejail: Application-Transparent Isolation of Libraries with Tight Program Interactions. In: *Computer Security – ESORICS 2012*. 2012. DOI: 10.1007/978-3-642-33167-1_49.

[201] Y. Wu, J. Sun, Y. Liu, and J. S. Dong. Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In: *28th IEEE/ACM International Conference on Automated Software Engineering*. ASE '13. 2013. DOI: 10.1109/ASE.2013.6693091.

[202] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In: *2012 IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '12. 2012. DOI: 10.1109/DSN.2012.6263958.

[203] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In: *2009 IEEE Symposium on Security and Privacy*. SP '09. 2009. DOI: 10.1109/SP.2009.25.

[204] S. H. Yong and S. Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-11. 2003. DOI: 10.1145/940071.940113.

[205] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '10. 2010. DOI: 10.1145/1755688.1755707.

[206] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables: The SmallEiffel Compiler. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '97. 1997. DOI: 10.1145/263698.263728.

[207] C. Zhang, M. Niknami, K. Z. Chen, C. Song, Z. Chen, and D. Song. JITScope: Protecting web users from control-flow hijacking attacks. In: *2015 IEEE Conference on Computer Communications*. INFOCOM '15. 2015. DOI: 10.1109/INFOCOM.2015.7218424.

[208] C. Zhang, C. Song, K. Chen, Z. Chen, and D. Song. VTint: Protecting Virtual Function Tables' Integrity. In: *22nd Annual Network and Distributed System Security Symposium*. NDSS '15. Feb. 2015. DOI: 10.14722/ndss.2015.23099.

[209] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song. VTrust: Regaining Trust on Virtual Calls. In: *23rd Annual Network and Distributed System Security Symposium*. NDSS '16. 2016. DOI: 10.14722/ndss.2016.23164.

[210] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In: *2013 IEEE Symposium on Security and Privacy*. SP '13. 2013. DOI: 10.1109/SP.2013.44.

[211] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In: *22nd USENIX Security Symposium*. SEC '13. 2013.

[212] M. Zhuang and B. Aker. *memaslap - Load testing and benchmarking a server*. 2022. URL: http://docs.libmemcached.org/bin/memaslap.html.

Memory corruption attacks have haunted computer systems for decades. Attackers abuse subtle bugs in an application's memory management, corrupting data and executing arbitrary code and, consequently, taking over systems. In particular, C and C++ applications are at risk, while developers often fail or lack time to identify or rewrite risky parts of their software.

In this thesis, we approach this problem with compilers that protect applications without requiring code changes or developer effort. We cover all treated aspects in legacy applications: returns, indirect forward jumps in both C and C++, and immutable libraries. First, we re-evaluate existing return address protections. In particular, we show that most adaption-preventing arguments have become less critical in the modern world and that already existing solutions can be deployable in production. Second, we protect virtual dispatch in C++ applications from hijacking. We employ a type analysis and a compiler transformation that implements virtual dispatch efficiently without hijackable pointers. Third, we protect indirect calls to function pointers in C applications. We use a new type-based analysis to find indirect call targets and transform indirect calls into a secure and fast version with limited targets. Finally, we propose a method to isolate potentially vulnerable code, particularly unprotected closed-source libraries, into compartments with restricted access to its environment.

Universität des Saarlandes