UNIVERSITY OF CALIFORNIA,
IRVINE

Binary Recompilation via Dynamic Analysis and the Protection of Control and Data-flows Therein

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Joseph Nash

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ardalan Amiri Sani
Professor Alexander V. Veidenbaum

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Page

ACKNOWLEDGMENTS

I would like to profusely thank my advisor, Prof. Michael Franz. He has been instrumental in guiding me towards the systems security and compiler fields, which are so suitable to my sensibilities. He provides an extremely positive outlook to the research direction and process that made my time in grad school a pleasure. His deft conversation and writing skills leave me consistently inspired to improve my own.

I would like to thank the postdoctoral researchers at I have worked with at UCI, Dr. Yeoul Na, Dr. Per Larsen, Dr. Stijn Volckaert, Dr. David Gens, and Dr. Adrian Dabrowski. They are a wellspring of fresh ideas and best practices and their contributions tremendously assisted my research. In addition, I would like to thank my friends and fellow students in the lab for being great colleagues and for making graduate school much more enjoyable. Brian, Julian, Mohaned, Prabhu, Taemin, Anil, Alex, Dokyung, Paul, Mitch, Matt, Fabian, and Chinmay —it has been a pleasure working with you!

Thank you to my colleagues and collaborators at institutions outside UCI, especially Scott Carr, Nathan Burow, Taddeus Kroes.

Finally, I would like to give special thanks to my committee members, Prof. Ardalan Amiri Sani and Prof. Alex Veidenbaum, and my advancement committee members Prof. Harry Xu and Prof. Steve Gross for their time and consideration.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-10-C-0237 and FA8750-15-C-0085, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211, CNS-1513837, CNS-1464155, CNS-1513783, CNS-1657711, IIP-1520552 and CNS-151383; as well as gifts from Intel, Mozilla, Oracle, and Qualcomm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

VITA

Joseph Nash

EDUCATION

Doctor of Philosophy in Computer Science **2020**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2017**
University of California, Irvine *Irvine, California*

Bachelor of Science with Honors in Engineering Physics **2014**
University of Illinois at Urbana-Champaign *Urbana, Illinois*

RESEARCH EXPERIENCE

Graduate Student Researcher **2015–2020**
University of California, Irvine *Irvine, California*

Technical Intern **Summer 2018**
Northrop Grumman Corporation *Beavercreek, Ohio*

Research Assistant **Summer 2016**
Oracle Labs *Redwood Shores, California*

Mathematical and Computational Systems Biology Trainee **2014–2015**
University of California, Irvine *Irvine, California*

Undergraduate Research Assistant **2012-2014**
University of Illinois at Urbana-Champaign *Urbana, Illinois*

Student Undergraduate Laboratory Intern **Summer 2012, Summer 2013**
Argonne National Laboratory Argonne, Illinois

TEACHING EXPERIENCE

Teaching Assistant **Spring 2016**
CS 143A: Operating Systems
University of California, Irvine *Irvine, California*

Teaching Assistant **Winter 2017**
ICS 53: Principles in System Design
University of California, Irvine *Irvine, California*

Teaching Assistant **Spring 2012, Spring 2013**
Physics 102: College Physics: Electricity, Magnetism and Modern Physics
University of Illinois at Urbana-Champaign *Urbana, Illinois*

REFEREED PUBLICATIONS

1. Anil Altinay*, **Joseph Nash***, Taddeus Kroes*, Prahbu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Herbert Bos, Cristiano Giuffrida, Michael Franz. “BinRec: Attack Surface Reduction Through Dynamic Binary Recovery." In *Proceedings of the Fifteenth EuroSys Conference (EUROSYS).* 2020
   * Equal Contribution Joint First Authors

2. Taddeus Kroes, Anil Altinay, **Joseph Nash** , Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, Cristiano Giuffrida. “BinRec: Attack Surface Reduction Through Dynamic Binary Recovery." In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST).* 2018

3. Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, **Joseph M. Nash**, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, Michael Franz. “Hardware Assisted Randomization of Data." In *Proceedings of the 21th International Symposium on Research in Attacks, Intrusions and Defenses (RAID).* 2018

4. Nathan Burow, Scott A. Carr, **Joseph Nash**, Per Larsen, Michael Franz, Stefan Brunthaler, Mathias Payer. “Control-Flow Integrity: Precision, Security, and Performance" In *ACM Computing Surveys.* 2017

ABSTRACT OF THE DISSERTATION

Binary Recompilation via Dynamic Analysis and the Protection of Control and Data-flows Therein

By

Joseph Nash

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael Franz, Chair

Legacy binaries need to continue functioning even when no source code has been preserved, to support the workflows of government and industry. The binaries often lack recent improvements in compiler design and software engineering practices, causing them to be slower and less secure than modern binaries. Binary rewriting seeks to patch, optimize, instrument, or harden binaries to bridge this gap, but existing practice is limited by the underlying static analysis. We created a framework, BinRec, to use dynamic analysis to lift binaries to LLVM IR then recompile them, which overcomes the limitations of static analysis.

The protection of software against memory corruption exploits has a rich history, which this thesis both systematizes and extends. We present a study of the performance, precision, and security of control-flow integrity (CFI). Data-only attacks can bypass CFI, and so we present a defense against these attacks. The application of these hardening techniques to binaries deserves further study, and we discuss the extent to which security hardening can be applied to recompiled binaries.

This dissertation presents building blocks for the securing of legacy binaries using dynamic analysis, which we hope will become a dominant paradigm in the secure software ecosystem of tomorrow.

Chapter 1

Introduction

The fact of the matter is there are many binary only programs being used today. Source code modification is not suitable for these cases, for example the source code or toolchain to produce the binary may no longer be available. This could happen if the source code license expires while the right to distribute the binary persists. Alternatively, the distribution of software in binary form increases the likelihood users will have a binary but its source code becomes lost when the vendor goes bankrupt. The downside of legacy binary software is it does not have the benefit of recent improvements and knowledge. These improvements could be in algorithms, toolchains, hardware, best practices, or use patterns. Even modern binaries may not be optimized for a particular use case.

A famous example in this space is when Microsoft patched a buffer overflow in their equation editor. But they did not change the source code. They did a surgical binary patch where they inserted a bounds check by hand to prevent the overflow. We don't know why they chose to go this route, but according to the news site ArsTechnica, this was third-party code which Microsoft didn't have the license to adjust the source [3]. In addition to patching, practitioners like to reverse engineer, re-optimize, or harden binaries for security. In all these cases, they are faced with a fundamental challenge, analyzing the binary. Analyzing a binary is quite hard. There are many properties of binaries one can attempt to understand, including what instructions are executed, where do functions start and end, what is the control flow of the binary [12]. On the way to understanding these properties, an analysis must answer questions like " Is this sequence of bytes code, or data?" Horspool and Marovac showed 40 years ago [93] that separating code and data is undecidable, but that has not stopped countless projects attempting binary analysis since then.

The two broad categories of binary analysis are dynamic and static. Dynamic analysis runs the binary, static analysis does not. Within these categories, different projects make design choices based on the properties of their input binaries (such as presence or absence of symbol information), and their goal. Binary recompilation uses binary analysis to perform program instrumentation and transformation and in addition produce a standalone executable. Some popular recent static works in the space are *Secondwrite* [9], *Superset Disassembly* [18] and *McSema* [68]. Yet even after years of development effort, these tools and others like them face serious shortcomings from their fundamental roots in static analysis. The use of heuristics mean these approaches are tuned to work for a particular use case, cannot be relied on in general [18].

Dynamic analysis provides a way to overcome the limitations of static binary rewriting. To that end, we created BinRec [8, 103]. BinRec takes dynamic traces through a binary and lifts them into LLVM IR. We produce a well formed LLVM IR module that is accessible to off-the-shelf LLVM optimizations and transformations. Then we produce an executable which faithfully executes the functionality of the original application, along with the desired transformations. We use BinRec to successfully run the SPEC 2006 integer benchmark suite. In addition we demonstrate BinRec's ability to apply off the shelf compiler transformations like SafeStack, and Address Sanitizer. The rewritten binaries we produce have good performance, about 29% relative to optimized input binaries, which is much lower than the comparable static recompilation tools (>100% overhead).

One of the main applications for dynamic recompilation is retrofitting on hardening against memory corruption attacks. There is a continual arms race between attackers and defenders in the memory corruption battle [185]. The attacks can be classified into control flow altering attacks, and attacks which stay within the intended control flow of a program [32, 38] (aka Data-only attacks). To prevent control flow altering attacks, we have Control-flow Integrity [5]. In this thesis, we include a systematization of knowledge of the CFI literature, including an analysis of its performance, precision, and security in Chapter 4. We explore in depth the control flow protection we can afford to binaries rewritten with BinRec.

As we will discover in Chapter 3, rewriting a binary in a sense creates an emulator for the original binary. As a consequence, the control flow of the original binary becomes data-flow in the rewritten binary. It is then important to consider defenses against data-only memory corruption attacks. We present a defense against data-only attacks which uses hardware acceleration to perform context-sensitive data space randomization (DSR) [21] in Chapter 5.

Since the publication of our work on CFI [28], Chapter 4, further work has brought to light the compatibility issues faced by CFI implementations [203]. Most CFI schemes need to exclude some programming constructions, and external code compatibility is a significant design point. We find external code is also a significant concern for data space defenses and binary rewriting, and we address those issues in this thesis.

In summary, the dissertation addresses the following research questions, and makes these contributions.

1.1 Research Questions

- **How does dynamic analysis compare with static analysis for binary rewriting?**
- **What are the limits of the protection against memory corruption we can apply to binaries? Can we protect both data and control flows?**

1.2 Contributions and Outline

This dissertation makes the following contributions, organized by chapter:

Chapter 3

- To the best of our knowledge, we create the first binary lifting framework that employs dynamic program analysis, trace merging, and incremental recovery to lift programs to a compiler-level intermediate representation. Our prototype successfully handles stripped, real-world release binaries. It is available at `https://github.com/securesystemslab/BinRec`.
- We show that the BinRec prototype robustly recovers all SPEC INT 2006 benchmarks without heuristics, the first lifting framework to do so. We also show that these recovered binaries outperform those that are successfully lifted by state-of-the-art lifting tools.
- We evaluate the efficacy of dynamic binary lifting in three application domains: i) Binary re-optimization, leveraging alias analysis tailored to the lifted IR resulting in improved performance in non-optimized binaries. ii) Binary hardening through CFI and compiler-level transformations such as Address Sanitizer and SafeStack. iii) Binary de-obfuscation through successful recovery of partial program semantics in virtualization-obfuscated binaries.

Chapter 4

- We create a systematization of CFI mechanisms with a focus on discussing the major different CFI mechanisms and their respective trade-offs.
- We introduce a taxonomy for classifying the underlying analysis of a CFI mechanism.

- We present both a qualitative and quantitative security metric and the evaluation of existing CFI mechanisms along these metrics.
- We include a detailed performance study of existing CFI mechanisms.

Chapter 5

- We propose the first context-sensitive data space randomization scheme, which offers greater security guarantees than prior solutions by dynamically choosing context-specific encryption keys based on the results of a context-sensitive analysis.
- We describe an ISA extension that efficiently supports our DSR scheme in hardware, and is also general enough to support all prior DSR designs.
- We implemented our DSR scheme and ISA extension and show that it achieves high precision with low overhead.

Chapter 2

Background

Binary analysis and the subsequent use of the analysis has seen tremendous academic interest. Likewise source based security transformations have a rich literature, and it is important to be familiar with both to understand what security we can achieve by recompiling binaries. We discuss here some of the foundational concepts. Additional background is in the beginning of Chapter 3 and Chapter 5. Additionally, Chapter 4 is a systematization of control-flow integrity knowledge and is a contribution of this thesis, so we discuss CFI only briefly in this background chapter.

2.1 Goals of Binary Analysis

Those outside the original implementation of software may be faced with the challenge of understanding the purpose and implementation of a binary in a process known as reverse engineering [41]. Their purpose may be to reimplement the functionality, or often to find vulnerabilities and construct exploits against the software. Binary patching seeks to change some portion of a binary. This could be to remove a bug, use a new instruction in the hardware, replace one function with another, or add a security check. Software patching can be seen as a limited form of software rewriting. The ratio of rewritten code to unmodified code typically determines whether patching techniques with limited scope are used, or 'high-touch' rewriting techniques are used instead. Binary rewriting is constituted by program wide analysis, and program wide transformation. Rewriting can be used for optimization [26], program hardening, wide-scale functionality modification, or analysis. All of these goals are facilitated by binary analysis.

2.2 Fundamentals of Binary analysis

Binary analysis comes in two fundamental categories, dynamic and static. There have been hybrids of the two approaches. The strengths and weaknesses are quite distinct between the categories.

Static analysis is any analysis that does not execute the code. For an excellent recent discussion of static analysis, see Andrisse [12]. The principal limitation of static analysis is that the detranslation of programs is equivalent to the halting problem, i.e., undecidable [93]. Static analysis turns to heuristics, which are specialized for a certain domain of input binary and desired application [18]. These heuristics enable the analysis to converge, but will give false positive or false negative results. It is essential to match the false result characteristics of a particular static analysis with the requirements of the application. We identified 5 critical challenges for the static analysis used for binary rewriting and discuss them in Section 3.2. Please see Section 3.9 for a review of static binary rewriting as well as related dynamic techniques such as Dynamic Binary Translation (DBT).

Dynamic analysis observes what a binary does during execution by using hardware or software instrumentation. An example software based analysis is a dynamic binary translation system (DBT) such as DynamoRIO [26] or PIN [119]. An example of a hardware mechanism for dynamic analysis is Intel Processor Trace [159]. On top of these capabilities, authors can examine program properties of interest such as instructions, call graphs, memory usage, or any number of higher level properties. The principal limitation of dynamic analysis is it needs to execute code paths of interest. Techniques such as fuzzing [126, 169] and symbolic execution try to explore more paths through a program, but are inefficient. Alternatively the execution through one carefully selected path can be recorded [70].

Symbolic execution [31, 99] is a technique developed to find bugs in software. It finds inputs for each possible code path in a program by replacing concrete values in memory with symbolic expressions. Concolic execution [43, 80] takes concrete input as a starting point, and maintains the concrete value as well as a symbolic expression for each data item. It can generate new, valid concrete values using the symbolic expression at any program statement. Concolic execution straddles the gap between static and dynamic analysis. With either concolic or symbolic execution, the principal limitation is due to the runtime cost of path explosion. Path explosion occurs because the number of paths through a program is exponential in the number of statements.

2.3 Static Analysis Properties

We present here some topics from static analysis that are relevant to both binary and source code analysis.

We are particularly interested in static analysis that identifies indirect calls/jump targets. Researchers refer to this kind of static analysis as points-to analysis. We refer the interested reader to Smaragdakis and Balatsouras [180]. Many compiler optimizations benefit from points-to analysis. As a result, points-to analysis must be sound at all times and therefore conservatively over-approximates results. The program analysis literature (e.g., [89, 90, 137, 180]) expresses this conservative aspect as a *may*-analysis relationship: A specific object "may" point to any members of a computed points-to set.

The following orthogonal dimensions in points-to analysis affect precision:

- *flow-sensitive* vs. *flow-insensitive*: this dimension states whether an analysis considers control-flow (sensitive) or not (insensitive).

- *field-sensitive* vs. *field-insensitive*: this dimension states whether an analysis considers struct members separately (sensitive) or not (insensitive).
- *context-sensitive* vs. *context-insensitive*: this dimension states whether an analysis considers various forms of context (sensitive) or not (insensitive). The literature further separates the following context information sub-categories: (i) call-site sensitive: the context includes a function's call-site (e.g., call-strings [175]), (ii) object sensitive: the context includes the specific receiver object present at a call-site [125], (iii) type sensitive: the context includes type information of functions or objects at a call-site [181].

Both dimensions, context and flow sensitivity, are orthogonal and a points-to analysis combining both yields higher precision.

Context-Sensitivity Figures 2.1d – 2.1f show the effects of context sensitivity on points-to analysis. In Figure 2.1d we see that the function `id` is called twice, with parameters of different dynamic types. Context-insensitive analysis (Figure 2.1f), does not distinguish between the two different calling contexts and therefore computes an over-approximation by lumping all invocations into one points-to set (e.g., the result of calling `id` is a set with two members). A context-insensitive analysis, considers a function independent from its callers, and is therefore the forward control-flow transfer symmetric case of a backward control-flow transfers returning to many callers [137]. Context-sensitive analysis (Figure 2.1e), on the other hand, uses additional context information to compute higher precision results. The last two lines in Figure 2.1e illustrate the higher precision by inferring the proper dynamic types `A` and `B`.

```
Object o;
o= new A();
...

o= new B();
```

(a) Flow-sensitivity example.

$o \rightarrow \texttt{A}$

$\ldots$

$o \rightarrow \texttt{B}$

(b) Flow-sensitive result.

$o \rightarrow \{\texttt{A}, \texttt{B}\}$

(c) Flow-insensitive result.

```
// identity function
Object id(Object o) { return o; }

x= new A();
y= new B();
a= id(x);
b= id(y);
```

(d) Context-sensitivity example.

$x \rightarrow \texttt{A}$

$y \rightarrow \texttt{B}$

$a \rightarrow \texttt{A}; \quad id_1 \rightarrow \texttt{A}$

$b \rightarrow \texttt{B}; \quad id_2 \rightarrow \texttt{B}$

(e) Context-sensitive result.

$x \rightarrow \texttt{A}$

$y \rightarrow \texttt{B}$

$a \rightarrow \texttt{id}; \quad id \rightarrow \texttt{A}$

$b \rightarrow \texttt{id}; \quad id \rightarrow \{\texttt{A}, \texttt{B}\}$

(f) Context-insensitive result.

Figure 2.1: Effects of flow/context sensitivity on precision.

2.4 Memory Corruption

After analyzing a binary, practitioners are faced with the decision of what transformation to perform. In our work, one of our main goals is to add security by means of hardening transformations. In the next section we include some background on memory corruption to set up the context for Control Flow Integrity, and Data Space Randomization. Work on CFI and DSR is a contribution of this thesis, so those are discussed in their own chapters.

Due to the use of unsafe languages (C, C++), software is prone to temporal and spatial memory safety violations [15]. Out-of-bounds array access is an example of a spatial error, and heap use-after-free is an example of a temporal error. The presence of these errors gives “attackers” the capability to read or write memory they should not have access to. Measures have been taken to detect and eliminate these memory errors [185], but they are still present in modern software. Software hardening and countermeasures have come into practice to prevent adverse consequences given these vulnerabilities. Defenders are in a continual race to introduce new hardening schemes, and attackers to bypass them.

Code injection allows attackers to inject executable bytes of their choosing and cause them to be executed. A classic example is the stack smashing attack [115]. Code injection today is largely prevented by Data Execution Prevention[11]. Since attackers could not introduce their own code, they turned to re-purposing already existing snippets of code in vulnerable programs [161]. The most popular way to reuse code in an attack is known as Return Oriented Programming (ROP) [171]. Attackers push a sequence of code addresses to the stack. At each address are a few instructions the attacker would like to execute, ending in a return instruction that invokes the next attacker controlled address on the stack.

Randomization-based or integrity-based mitigations have been used to combat code reuse exploits. Layout randomization-based approaches make make the attacker's job more difficult by making the locations of reusable pieces of code unpredictable. Address Space Layout Randomization (ASLR) is deployed on all commodity systems, and it randomizes the base address of libraries [148]. There are also more finely-grained code layout randomizations, which raise the bar for adversaries [107]. In either case, memory disclosure may allow attackers to bypass the mitigation.

Control-flow Integrity is an integrity based code reuse defense which enforces control-flow to stay within the intended control-flow of a program. Yet attackers have learned to exploit a program while staying within the restrictions of CFI. If these attacks exploit a difference between the programmer's intent, and the precision of applied control flow protection, they are known as Control-flow Bending attacks [32]. If these attacks do not use any illegitimate control flow edges, they are known as data-only attacks [38]. Data only attacks have been shown to be Turing-complete [95].

The defenses against data-only attacks are comparatively less mature than those defending against code reuse. In general, defenses against data-only exploits try limit the amount of data which is writable or readable by a particular memory accessing instruction. We have seen randomization based and integrity based approaches, in a parallel manner to the control flow protections. Integrity based defenses include Hardware-Assisted Data-flow Isolation [182], and in the randomization space, there is Data Space Randomization[22]. Both these approaches suffer from a lack of precision, meaning the number of data items which are potentially readable or writable from a given instruction is larger than necessary to correctly execute the program. In addition to precision limitations, existing approaches to DSR suffer from something that is a challenge is many domains, securely storing secrets, namely encryption keys. In Chapter 5, we correct these flaws with DSR to create a state of the art data-only attack mitigation.

2.5 Memory Corruption Defenses for Binaries

Binary rewriting has been explored as a method to make software more secure. The two main approaches for increasing the security of a binary are attack surface reduction, and hardening. Attack surface reduction is a form of program specialization that removes unneeded code or code paths from a program or API [123, 173]. It makes use of the fact that attackers can find code reuse gadgets in statically or dynamically dead code. The dead code can safely be removed from the binary, so it is no longer available to attackers.

Binary hardening is another name for the application of exploit mitigations to a binary. It is in general more difficult to harden a binary than a source program because there is less semantic information available in the binary. The security consequence of the analysis difficulty is that typically, static binary analysis based solution solutions restrict attackers less than the source based versions of that hardening transformation. Approaches such as BinCFI [212] work on stripped binaries, but need to use heuristics to over approximate the set of program behaviors which should be allowed. Other approaches rely on symbol tables and debug information to determine more precisely what binary behaviors should be allowed [111, 211]. On the other hand, through the use of dynamic information, binary hardening transformations can be *more* precise than the source level counterparts, as we accomplish in BinRec.

The overhead of mitigations is always critical to their deployment prospects. Multiverse [18] is flexible enough to apply precise hardening transformations like shadow stacks to binaries, but the associated overhead means the system is not likely to be deployed in practice.

The evaluation of security properties in rewritten binaries is non-trivial. Any runtime component or instrumentation added to a binary is potentially new attack surface. Several binary rewriting systems including our system, BinRec, and McSema [68] rewrite a program by means of constructing an emulation of the original binary. As a consequence, what was control-flow in the original binary becomes data-flow in the rewritten one. We explore this concept in section 3.6.

Chapter 3

Dynamic Binary Lifting and Recompilation

Binary lifting and recompilation allow a wide range of install-time program transformations, such as security hardening, deobfuscation, and reoptimization. Existing binary lifting tools are based on static disassembly and thus have to rely on heuristics to disassemble binaries.

We present a new approach to heuristic-free binary recompilation which lifts dynamic traces of a binary to a compiler-level intermediate representation (IR) and lowers the IR back to a "recovered" binary. This enables rich program transformations, such as compiler-based optimization passes, on top of the recovered representation. We identify and address a number of challenges in binary lifting, including unique challenges posed by our dynamic approach. In contrast to existing frameworks, our dynamic frontend can accurately disassemble and lift binaries without heuristics, and we can successfully recover obfuscated code and all SPEC INT 2006 benchmarks including C++ applications. We evaluate the prototype, BinRec, in three application domains: i) binary reoptimization, ii) deobfuscation (by recovering partial program semantics from virtualization-obfuscated code), and iii) binary hardening (by applying existing compiler-level passes such as AddressSanitizer and SafeStack on binary code).

3.1 Introduction

Binary rewriting [69, 195, 196] has many applications such as post-installation program hardening [39, 121, 154, 191, 192, 212], (de)obfuscation [50, 200, 204], and reoptimization [61]. However, its effectiveness is limited in practice by the complexity of analysis and transformation in the absence of source code.

To overcome the limited expressiveness of assembly code, researchers introduced "binary lifting" which raises machine instructions to higher-level intermediate representations (IR) such as LLVM bitcode [9, 67, 68]. Binary lifting has the potential to capitalize on powerful compiler-level analysis and transformations already available in production compilers such as binary reoptimization. Despite its benefits, binary lifting has not seen widespread adoption in practice because existing approaches rely on static disassembly, which is fundamentally unable to accurately model indirect control-flow targets, differentiate between code pointers and data constants, or identify the boundary between data and instruction bytes [12, 93].

While heuristics have been used to successfully circumvent these limitations for certain binaries that adhere to specific assumptions [9, 196], binaries that are the target of analysis are typically release builds, stripped of symbols and debug information, and sometimes even intentionally obfuscated by vendors or malware authors. Code patterns found in such binaries easily violate these assumptions, e.g., handwritten assembly, highly optimized code, code produced by non-standard compilers, obfuscated or packed code, and even position-independent code, which is commonly used in shared libraries [18].

In contrast to static translation methods, dynamic binary translation (DBT) tools such as Pin [119], DynamoRIO [26] and Valgrind [136] analyze concrete executions of a target program, and thus can seamlessly handle all statically unknown components such as mixed code and data, and indirect control-flow targets. Unfortunately, the usability of existing DBT tools is limited for two reasons: first, they operate on the level of machine code, limiting the availability of complex analysis tools.

Second, the rewritten code in their output is tailored to the tool's runtime environment, and can not be reused for subsequent executions. In other words, any transformation on the binary has to be done again each time the program runs. This introduces performance and portability problems for instrumented applications.

We present a framework that employs dynamic analysis to lift binary code to LLVM IR in order to apply complex transformations, and subsequently *lowers* it back to machine code, producing a standalone executable binary which we call the *recovered* binary. To the best of our knowledge, BinRec is the first binary lifting framework based on dynamic disassembly, enabling lifting of statically unknown code for the first time. Additionally, BinRec is the first dynamic binary rewriting tool that persists its transformations in a standalone output binary.

Our main goal is to recover code that is opaque to static analysis. While our use of dynamic analysis solves this issue, it brings with it the problem of covering code that is not exercised during lifting: when dynamically lifting a program from a single trace, one only observes one out of many possible code paths. Hence, the recovered binary only supports code paths for which all control flow edges are present in the code path observed during lifting. A *control flow miss* occurs when the recovered binary reaches a code path that was not covered during lifting. Much like page faults are handled by a page fault handler in modern operating systems [52], control flow misses may be handled means of customizable handlers that may disallow the unknown control flow transfer by stopping execution. Alternatively, the handler may be configured to apply *incremental lifting*, allowing unknown edges and retrofitting the binary with the newly found code path. For optimization scenarios, the handler may even be left empty to allow for aggressive branch pruning, specializing the binary for a specific input format. Applications of our framework may select a handler that best suits their needs, for instance depending on whether unknown control flow is assumed to be malicious or not. The use of dynamic tracing enables us to produce recovered binaries with precise control-flow integrity (CFI). The allowable targets for any indirect control-flow are hence limited to the ones observed during (optionally incremental) lifting. We show that BinRec produces recovered binaries hardened with control flow integrity (CFI) with slowdowns of 0.98x – 1.29x, depending on the optimization level of the binary.

Crucially, we harness the power of existing IR-level compiler analyses and transformations on binaries where static lifting fails. Our evaluation on SPEC CPU2006 shows that BinRec successfully lifts code patterns in optimized input binaries that state-of-the-art static lifters such as McSema [68] and Rev.ng [67] cannot. To demonstrate the immediate benefits of lifting binary code to compiler IR, we show that BinRec improves performance of some of our non-optimized input binaries and successfully applies two security transformations available in LLVM—SafeStack [105] and AddressSanitizer [170]—to our lifted IR. In contrast to previous binary rewriting approaches, BinRec naturally enables these compiler transformations without any additional engineering effort. We also show that trace-based lifting enables us to recover partial program semantics of virtualization-obfuscated binaries, by combining IR-level analysis with readily available compiler optimizations.

3.2 Analysis Limitations for Static Binary Lifting

Analyzing binary code – or translating it to an accurate high-level representation that is better for analysis, transformation, and recompilation – is a challenging problem. The problem is compounded in cases where the binary code is encrypted or obfuscated. While many general problems of (static) disassembly have been well documented in the literature [12, 93], in this section we reiterate in detail some of the current unsolved challenges in the context of binary lifting and program transformation through static methods. We describe these challenges below and motivate our new dynamic approach by explaining why static, heuristic-driven approaches are inherently insufficient for lifting arbitrary binaries.

3.2.1 L1 Code and Data Separation

By default, stock compilers do not attach labels to the data and references they embed into a program. To distinguish code from data and references from constants, the appropriate labels must be inferred through program analysis. This problem is undecidable in the general case [93], so state-of-the-art analysis tools employ heuristics to approximate the correct label set [195, 196, 212]. A data value, for example, can be considered a code reference if it is aligned correctly, and if it represents a valid code address in the binary. However, value collisions occur frequently [195] and alignment is not mandatory on many platforms. A dynamic tool can accurately assign labels by observing how the CPU interprets the values it reads from memory.

3.2.2 L2 Indirect Control Flow

Indirect Control Flow transfers (iCFTs) may transfer control to one or more target locations depending on their execution context. Indirect calls are used to implement calls to function pointers in C code, which are even more prevalent in C++ code in the form of virtual functions. Indirect branches often implement switch statements and position-independent code (PIC). In PIC, all direct branches are replaced with indirect branches that add the offset at which the binary/library is mapped in memory, to the branch target.

Statically identifying all potential targets of iCFTs is, again, undecidable in the general case [93]. Static approaches do achieve high accuracy when identifying the potential targets of iCFT instructions that load their destination address from jump tables [66, 212]. Resolving indirect function calls and returns, on the other hand, remains a challenge. Wang et al. [196] argue handling iCFTs can be supported through their approach, but their prototype Uroboros does not handle iCFTs. The underlying analyses [66] used in Rev.Ng [67] claim 90-95% jump target recovery depending on architecture.

Meanwhile, Qian et al. [156] as well as Zhang and Sekar [212] use a lookup-table that translates original target addresses to the new addresses at run time, effectively resulting in a hybrid approach between static and dynamic rewriting. The table contains potential targets collected based on heuristics.

Dynamic tracing can reliably identify control flow targets as it follows the CPU to any jump target regardless of how the target address is computed.

3.2.3 L3 External Entry Points

Dynamic linking is prevalent in real world software, and it presents additional hurdles to binary analysis and rewriting. Analyzing and rewriting external libraries at a binary level is generally infeasible; this requires static linking for all the library code [67] and incurs significant overhead [18]. Without visibility of all the code, however, the control and data flow between program modules is only partially observable to binary analysis through the interface of external modules.

Such partial visibility can be a problem when a code pointer of the main module is passed as an argument to an external module and is used to re-enter the main module, e.g., callbacks. After binary rewriting, this code pointer will become invalid because the code layout changes. Some existing binary rewriters attempt to support such callbacks by implementing special case handlers for the interface of known libraries [9, 198]. However, they cannot correctly handle external callbacks through unknown interfaces. Multiverse instead implements run-time lookup tables to handle callbacks [18] as a generic but heavyweight solution to support unknown external entry points.

Dynamic tracing can easily capture such entry points by recording control flow transfers going in and out of the targeted code space, which enables performant, surgical control and data modification at these points.

3.2.4 L4 Ill-formed code

Manually written assembly code is not only used for optimization, but as an anti-debugging and anti-disassembly technique as well. While generated code is somewhat predictable, aggressive compiler optimizations can lead to similar ill-formed instruction constructs [12].

Overlapping instructions are a classic anti-disassembly technique [117] but occasionally appear in highly-optimized libraries too [12]. Selection control structures (e.g. switch/case) are lowered as *Inline data and jump tables* by some compilers. *Overlapping basic blocks, multi-entry functions, and tail calls* obscure the detection of function boundaries.

Dynamic tracing bypasses handling of ill-formed code during disassembly by observing the actual instructions executed by the CPU instead.

3.2.5 L5 Obfuscation

In addition to naturally occurring technical challenges, binary lifting approaches may have to deal with binaries that have explicitly been modified with the intent to obstruct analysis. While these obfuscation techniques are well documented [1, 2, 46], they still pose significant challenges in practice.

For instance, virtualizing obfuscators transform executable code stored in code sections into bytecode stored in data sections, and embed a virtual machine into the program to interpret the bytecode [1, 10]. In a program protected by such an obfuscator, the static code sections reveal little to no information about the behavior of the program. Other problematic obfuscation techniques include opaque predicates [47], control-flow flattening [46], and aliasing [194]. All these transformations can be used to artificially inflate the size and complexity of the program's control-flow graph to a point where static disassembly becomes intractable.

Dynamic lifting can revert all of these obfuscating transformations to some extent. In the case of virtualizing obfuscation, a dynamic lifting tool can capture the run-time semantics of the program in the form of executable code, which can then be transformed into an equivalent deobfuscated trace [50, 163, 174, 204]. In the other cases, a dynamic tool can remove dead code and spurious aliases.

3.3 Design

Our design overcomes the fundamental limitations identified in Section 3.2. We achieve this by leveraging *dynamic program analysis* to recover accurate disassembly of binaries which is then translated into a transformable, high-level intermediate representation (IR).

Figure 3.1 shows a high-level overview of our approach, consisting of three logical components: an extensible dynamic lifting engine and data collector, a transformation component that rewrites the IR code in a canonical way, and a back-end that compiles the transformed IR back to machine code and produces an executable binary. The lifting engine is extensible to support different execution driving paradigms. After running the canonicalization component, the full range of existing LLVM-based transformations can be applied to the client program.

3.3.1 Factors Significant to Dynamic Lifting

While our dynamic approach naturally sidesteps the limitations of static disassembly, it comes with its own set of challenges that need to be carefully addressed.

Coverage A fundamental challenge for any dynamic analysis is to drive execution through all desirable code paths [130]. Which code paths are desirable, however, depends on the type and goal

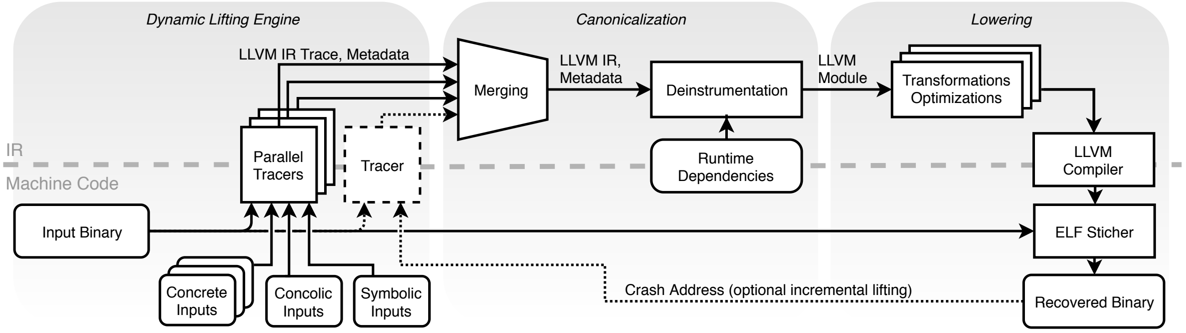


Figure 3.1: The steps of binary recovery: lifting to compiler IR, transformation on the IR, and lowering back to machine code.

of the analysis. To optimize binaries, for example, it is sufficient to explore the most frequently executed paths. For security hardening, it might be acceptable to explore only those paths reachable through trusted inputs and to prune all unexplored paths. For testing, the execution may need to cover all the code paths in the binary.

The paths the analysis covers depend on the set of inputs that drive execution, as is the case for any other dynamic analysis. We designed BinRec with configurable execution driving paradigms to accommodate a wide spectrum of applications (Section 3.3.2). We also merge multiple traces into a single transformable IR module, thereby recovering multiple sets of code paths (Section 3.3.3).

However, even with an ideal execution driver, the desirable control flow paths may not be fully exercised. This can lead the execution of the recovered binary to flow to code that was not covered during lifting, an event we refer to as a *control flow miss*. Lack of coverage can occur because the control flow of the program depends on implicit program inputs such as timing information, random numbers, and literal memory addresses. The coverage may also be incomplete because the concrete or symbolic inputs that achieve full coverage cannot be feasibly calculated. BinRec therefore handles control flow misses by means of customizable miss handlers, again, based on the application scenarios: The handler may be configured to disallow or ignore an unknown control flow transfer, or to incrementally recover the binary with the newly found code path (Section 3.3.5).

Scalability To dynamically disassemble or lift binary programs, they must be executed with concrete or symbolic inputs according to coverage considerations. Generating inputs to achieve maximum coverage is not only difficult, but may lead to path explosion for complex programs. To address this, we designed the dynamic analysis to record multiple, independent, traces of the binary (resulting from multiple executions of the binary with different inputs). We can merge the resulting traces at a later stage to increase global coverage. This design splits the analysis of complex, large binaries into smaller manageable chunks which can be lifted in a distributed and/or parallel infrastructure (Section 3.4.1).

In contrast to DBT approaches, which must recompute analysis facts and insert instrumentation on every run of a program, BinRec separates the analysis and transformation from the execution of the rewritten binary. In this way, we can do those heavyweight operations ahead of time, and have better performance of rewritten binaries.

3.3.2 Dynamic Lifting Engine

Execution Driver BinRec takes a multi-pronged approach, using several complementary methods, to drive dynamic execution. These methods use different types and sources of inputs. The first source of input to drive a program for dynamic lifting should be a test corpus exercising desired features. The more closely this corpus matches the real workload on the rewritten binary, the better. However, user-specified tests alone are unlikely to fully exercise all the code paths that should be lifted. Besides the obvious sources of explicit input to a program (command line, stdin), there can be implicit inputs that are much less obvious to users but still need to be accounted for. These can include address layout, timers, random number generators, interrupts and network packets. Even if users can specify the explicit inputs for every conceivable desired behavior of their specialized program, it is highly unlikely they would be aware of all the implicit inputs. We therefore turn to alternative techniques to produce specialized programs that are robust enough to function correctly in the presence of implicit input.

One potential solution to this problem is to drive execution through all or most of the program paths that depend on implicit input. Towards this solution, we drive some of the implicit inputs that cannot be triggered merely through explicit inputs. For example, we found an interesting case in the Perl interpreter where the control-flow depends on the virtual address space layout (specifically on the alignment of `argv` strings). We exercise this implicit input source by controlling the lengths of environment strings in a way that results in different `argv` alignments. Similarly, we enable address space randomization (ASLR) during tracing, to exercise more code paths dependent on the address space layout.

While achieving complete code coverage is not our objective, the users may still require nearly complete code coverage depending on the application. For this use case, BinRec supports concolic execution [43] to explore more code paths.

Alternatively, BinRec can take fuzzer-generated, concrete inputs to drive the binary lifting frontend. Concolic execution and fuzzing have complementary strengths and weaknesses [80, 209]. Fuzzing scales well to large programs, but has difficulty exploring all branches of complex conditional statements. Concolic execution is useful to drive execution through such conditionals. We found concolic execution to become untenable on programs with cryptography or hashing, such as SHA2. In those programs, the SMT solver becomes a bottleneck. The input generation interface is flexible and extensible, which allows the dynamic driver to be customized for a particular client application, and so explore program paths using the best methodology for the target.

Figure 3.2 shows how we symbolically execute Listing 3.1 by passing two symbolic arguments of one byte each, recovering all possible code paths.

Dynamic Data Recording We record dynamic data about the execution of each program path specified by the driver. This data is key to overcome the fundamental limitations of static binary

```
int main(int argc, char **argv) {
  if (argc == 3) {
    char a = argv[1][0];
    char b = argv[2][0];
    if (a == b) {
      puts("arguments are equal");
    }
  }
  return 0;
}
```

Listing 3.1: A simple C program that checks if the leading characters of two provided arguments are equal.

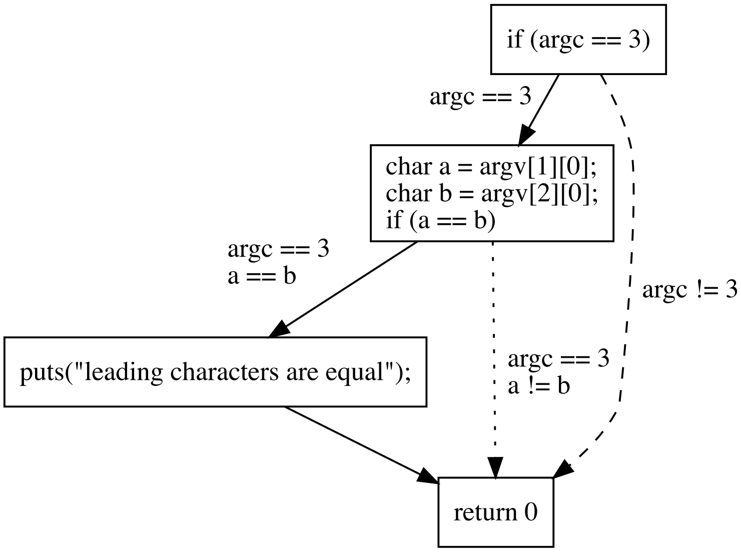


Figure 3.2: Symbolic execution of Listing 3.1. `argc`, `a` and `b` are symbolic values, causing the execution state to fork twice as represented by the different arrow styles. Edge labels show the constraints recorded in each execution state.

lifting as explained in Section 3.2. We currently record which instructions were executed, where the function boundaries are, and the observed targets of each branch instruction. We use this information to accurately disassemble binaries and produce canonical IR, as explained in Section 3.3.3. Our framework is extensible, so other data can be recorded to fill the needs of downstream transformations. The recorded data is fully accurate on paths which are exercised by the dynamic lifting engine, but we cannot reason about data that is not covered by the dynamic traces.

The front-end decodes and records each instruction executed by the client binary using the program counter. This procedure is agnostic to the static representation of the executable code and is therefore not affected by any intentional or unintentional differences between the static and dynamic (actual) instruction trace. Such differences would arise in the presence of unaligned, packed, or encrypted code. We therefore address L4 and some aspects of obfuscation L5. A tradeoff incurred by this design choice is a potentially slower lifting front-end. A scheme that dynamically records control flow, but that lifts disassembled basic blocks statically would occupy another point in the design space, and would sacrifice compatibility with non-standard binaries for faster lifting.

3.3.3 Canonicalization

Merging Traces BinRec can compose program traces generated over different runs using different execution driving paradigms. We implemented a technique to merge distinct traces into one specialized program which will behave correctly on all covered paths. In concrete terms, merging proceeds by lifting N instances of the target program in parallel. The different execution paths can be driven by fuzzing, concolic execution, or a chosen corpus of inputs. Then, we create one LLVM IR module from N LLVM IR modules using metadata we collected during lifting.

In a dynamic binary lifting system, execution traces serve a fundamental role in mapping between program input, program semantics, and lifted program structure. A lifted program based on exactly one dynamic trace is a version of the original program which has been specialized for exactly one input and execution path. The ability to merge multiple traces into one program is both a selling point for system users, and a desirable engineering capacity. Any unnecessary features of the original binary can then be removed from the attack surface of the recovered binary.

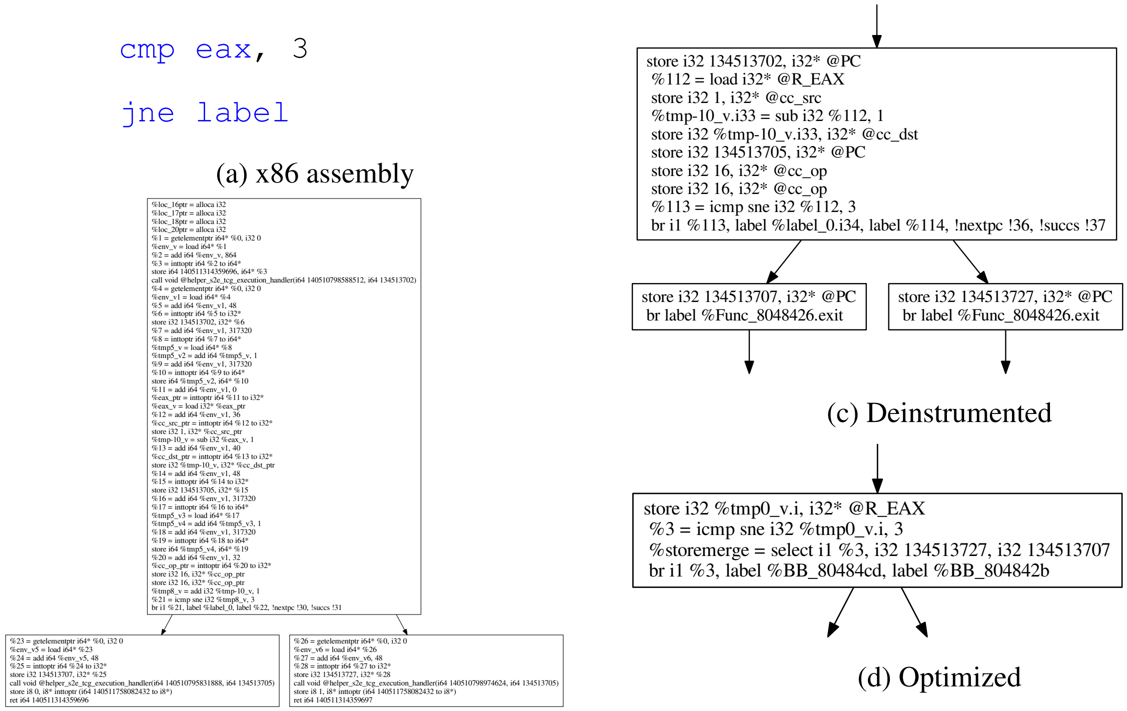
Merging depends on the ability to correlate the code and data addresses of one dynamic trace with another. In the case of position independent code, the addresses change from trace to trace, but are correlated by simple subtraction of the section base addresses. Traces from programs using fine grained code and/or data layout randomization (at load or run-time) could be merged using a specific mapping function taking the randomization seed as input. We leave the implementation of such correlation techniques as future work.

Trace merging relies on canonicalization of code to identify which instructions in different traces represent should be considered equal in the merged trace. We currently identify an instruction by the address where it is mapped during lifting. This mapping differs across invocations of programs containing position-independent code or fine-grained randomized code layouts [107]. To support trace merging for these programs, future work can record the mappings of code and data sections during lifting, and use them to canonicalize traces before merging them.

The code of the combined program is the union of all basic blocks observed in the merged traces. The allowed targets of each control flow statement in the combined program are the union of the observed targets for each observation of that branch in the merged traces.

It may be observed that this is a path-insensitive procedure. The resulting control-flow graph, before optimization, resembles the original program's CFG but lacks the nodes that were not executed while lifting. One could imagine an alternative, path-sensitive, reassembly technique, where only control flow paths exactly following one of the recorded traces are allowed. However, it is likely unprofitable to construct such a recovered program, as in effect this would be a tree traversal of the original program's control flow graph, and the resulting program would have a code size explosion. This method of combining program paths shares many similarites with the tail deduplication compiler optimization [120].

Removing Lifting Instrumentation BinRec uses an emulation-based dynamic lifting engine, which allows us to lift programs compiled for a different instruction set architecture than the host system. IR generated from such an emulation-based engine, however, is heavily instrumented to facilitate execution in a virtualized environment. This code cannot be used as a standalone program, unless we remove the instrumentation code. Our framework contains a *deinstrumentation*



```
cmp eax, 3
jne label
```

(a) x86 assembly

```
%loc_16ptr = alloca i32
%loc_17ptr = alloca i32
%loc_18ptr = alloca i32
%loc_20ptr = alloca i32
%1 = getelementptr i64* %0, i32 0
%env_v = load i64* %1
%2 = add i64 %env_v, 864
%3 = inttoptr i64 %2 to i64*
store i64 140511314359696, i64* %3
call void @helper_s2e_tcg_execution_handler(i64 140510798588512, i64 134513702)
%4 = getelementptr i64* %0, i32 0
%env_v1 = load i64* %4
%5 = add i64 %env_v1, 48
%6 = inttoptr i64 %5 to i32*
store i32 134513702, i32* %6
%7 = add i64 %env_v1, 317320
%8 = inttoptr i64 %7 to i64*
%tmp5_v = load i64* %8
%tmp5_v2 = add i64 %tmp5_v, 1
%9 = add i64 %env_v1, 317320
%10 = inttoptr i64 %9 to i64*
store i64 %tmp5_v2, i64* %10
%11 = add i64 %env_v1, 0
%eax_ptr = inttoptr i64 %11 to i32*
%eax_v = load i32* %eax_ptr
%12 = add i64 %env_v1, 36
%cc_src_ptr = inttoptr i64 %12 to i32*
store i32 1, i32* %cc_src_ptr
%tmp-10_v = sub i32 %eax_v, 1
%13 = add i64 %env_v1, 40
%cc_dst_ptr = inttoptr i64 %13 to i32*
store i32 %tmp-10_v, i32* %cc_dst_ptr
%14 = add i64 %env_v1, 48
%15 = inttoptr i64 %14 to i32*
store i32 134513705, i32* %15
%16 = add i64 %env_v1, 317320
%17 = inttoptr i64 %16 to i64*
%tmp5_v3 = load i64* %17
%tmp5_v4 = add i64 %tmp5_v3, 1
%18 = add i64 %env_v1, 317320
%19 = inttoptr i64 %18 to i64*
store i64 %tmp5_v4, i64* %19
%20 = add i64 %env_v1, 32
%cc_op_ptr = inttoptr i64 %20 to i32*
store i32 16, i32* %cc_op_ptr
store i32 16, i32* %cc_op_ptr
%tmp8_v = add i32 %tmp-10_v, 1
%21 = icmp sne i32 %tmp8_v, 3
br i1 %21, label %label_0, label %22, !nextpc !30, !succs !31
```

```
%23 = getelementptr i64* %0, i32 0
%env_v5 = load i64* %23
%24 = add i64 %env_v5, 48
%25 = inttoptr i64 %24 to i32*
store i32 134513707, i32* %25
call void @helper_s2e_tcg_execution_handler(i64 140510795831888, i64 134513705)
store i8 0, i8* inttoptr (i64 140511758082432 to i8*)
ret i64 140511314359696
```

```
%26 = getelementptr i64* %0, i32 0
%env_v6 = load i64* %26
%27 = add i64 %env_v6, 48
%28 = inttoptr i64 %27 to i32*
store i32 134513727, i32* %28
call void @helper_s2e_tcg_execution_handler(i64 140510798974624, i64 134513705)
store i8 1, i8* inttoptr (i64 140511758082432 to i8*)
ret i64 140511314359697
```

(b) Instrumentation after lifting

```
store i32 134513702, i32* @PC
 %112 = load i32* @R_EAX
 store i32 1, i32* @cc_src
 %tmp-10_v.i33 = sub i32 %112, 1
 store i32 %tmp-10_v.i33, i32* @cc_dst
 store i32 134513705, i32* @PC
 store i32 16, i32* @cc_op
 store i32 16, i32* @cc_op
 %113 = icmp sne i32 %112, 3
 br i1 %113, label %label_0.i34, label %114, !nextpc !36, !succs !37
```

```
store i32 134513707, i32* @PC
 br label %Func_8048426.exit
```

```
store i32 134513727, i32* @PC
 br label %Func_8048426.exit
```

(c) Deinstrumented

```
store i32 %tmp0_v.i, i32* @R_EAX
 %3 = icmp sne i32 %tmp0_v.i, 3
 %storemerge = select i1 %3, i32 134513727, i32 134513707
 br i1 %3, label %BB_80484cd, label %BB_804842b
```

(d) Optimized

Figure 3.3: Deinstrumentation of a small basic block from Listing 3.1 (a). Dynamic code lifting captures instrumented, decoupled code (b). Deinstrumentation shortens the code and adds explicit control flow instructions (c). After optimization, a single basic block remains in the IR (d).

component that eliminates dependencies on the run-time environment from lifted code, and merges all captured code together into a single LLVM module that is suitable for use in subsequent transformation passes and compilation into a standalone binary.

Whereas a program binary can explicitly use physical CPU registers and memory references, the lifted IR of a recovered program has an abstract representation of the memory model in the original binary. To handle this abstraction gap, we represent physical registers, stack and memory locations as objects in the high-level IR. This enables us to generate programs which contain two stacks and register sets. The native stack contains data such as register spills and return addresses, as well as any data we add while transforming the lifted program. The emulated stack and register set contain the data of the original binary. Generated code interacts with this emulated environment to reproduce the functionality of the original program. The emulated state cannot be fully optimized into native state due to the lack of semantic information about the size and lifetime of stack allocations.

To illustrate the state of the deinstrumented code compared with other stages of lifting, we recover a simple C program that prints whether two arguments are equal. We use symbolic execution to recover a complete control flow graph. The front-end initially lifts the binary to 630 LLVM instructions. 36 instructions remain after deinstrumentation and optimization, a reduction of one order of magnitude. We also compiled the example to LLVM IR using the Clang compiler, counting 33 instructions as a baseline for recovery. The 3 instructions of overhead in the recovered program are generic set-up code.

Design of IR

A key point of differentiation between binary rewriting approaches is how they represent the results of the binary analysis. On one extreme, there may be only local information and no stored state. For example, substituting one instruction for an equivalent one wherever it is found in the binary. On the other extreme, a binary could be decompiled into source code[4]. Compiling an binary is an inherently lossy procedure, so it is extremely difficult to recover a source representation from a binary without debug symbols or other additional information. In between these two approaches is to represent some properties of the binary under analysis in an intermediate representation(IR). The IR can be a very low level assembly like language, or it can have richer semantic constructs to represent program code. There are benefits on each side. A low level IR is closer to the original binary, so it is more suitable for producing minimal binary rewriting and maintaining a mapping between program points in the original binary and the rewritten one. With a high level IR, it is easier to perform large scale transformation. Another dimension of IR is its existing ecosystem.

The IR can be ad-hoc and unique to a binary rewriting tool, or it can be standardized. Valgrind [136] IR and LLVM [108] IR are two popular choices for representing analyzed binaries. The choice of IR can drastically effect the usability and development speed of implementing analysis and transformation within the binary rewriting framework. The use of a standardized IR may allow existing transformations and analysis to be paired with a novel binary analysis with little additional effort. For BinRec we have chosen to lift binaries to LLVM IR. It is not trivial, however, to represent a binary in this way. LLVM IR is a more abstract language than one can typically obtain from binary analysis, so some aspects have to be embedded as metadata within LLVM IR.

Control-Flow Canonicalization

Indirect Control Flow Resolution Our lifting front-end produces a collection of executed basic blocks, and a list of control-flow graph edges. We use this data to emit control-flow transfers with sound and precise lists of allowed targets. Direct control flow transfers have a one-to-one correspondence between nodes and edges in the observed control-flow graph of the client binary, and the recovered binary. We therefore represent them in a straightforward way in recovered code, using the original semantics.

Even the most precise static analysis allows more control flow targets than necessary due to analysis imprecision (see challenge L2). In contrast, we simply record the exact dynamic targets of each indirect control flow transfer in a client binary in the lifting engine. To execute the corresponding control flow in the recovered binary, we determine the address that original code would jump to, then use that address as a key to look up the recovered code target. This is represented as a switch table in LLVM IR. We emit the minimal set of dynamic targets, which can enable further optimization by limiting the lifetime of values. Static lifting can only receive these benefits to the extent that indirect branch targets can be statically determined. This has been extensively explored in the program analysis [12] and CFI literature [28, 33], and previous work has found even the most precise static analysis overapproximates the set of possible targets.

Library Calls BinRec supports calls to external (i.e., non-recovered) libraries. The principal step necessary to execute such a library call is to marshall the emulated program state into concrete state before the call. Marshalling is necessary to match the ABI of linked libraries. Upon return from

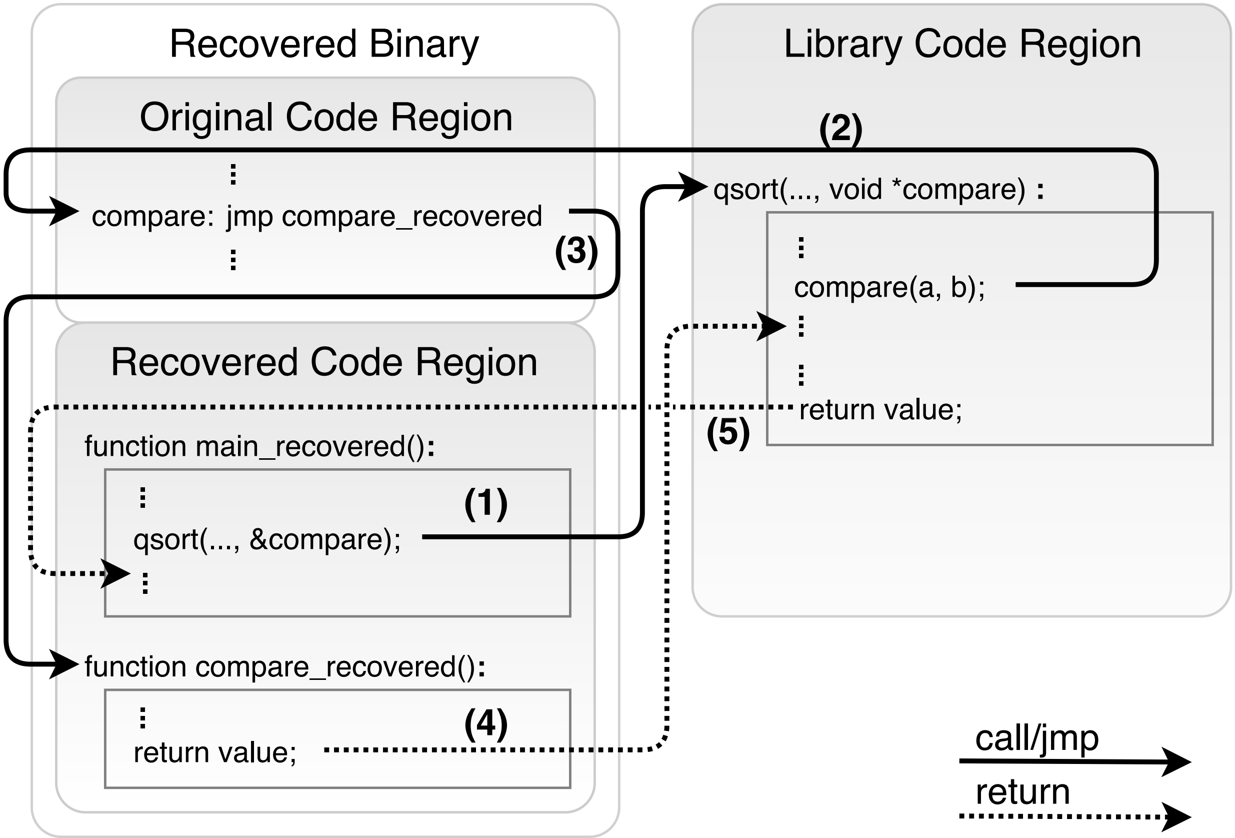


Figure 3.4: The address space of a recovered program that calls the `qsort` library function. Control flows as follows: (1) Call to library with original function pointer; (2) Callback via function pointer; (3) Original function was replaced with jump to recovered code; (4)(5) Returns.

the library, the concrete state is reloaded into the emulated state. The maximum amount of state that may need to be transferred is the full register set, including the stack pointer. When possible, we can use the function signatures of external library calls to optimize the state marshalling. With signature information, only caller saved registers which are actually read or written need to be marshalled from emulated state to concrete state. Our prototype implementation of BinRec uses signature information to optimize calls to the C library.

External Callbacks Our approach to solving the external callback challenge L3 is both sound and performant. Only a dynamic lifting approach can achieve both these properties at once. In the lifting front-end, we detect execution of the binary under analysis, and record call targets where the caller is outside the analysis region (i.e., callback functions). There is no need to track callback pointers at any other time because we detect when they are actually invoked. We also record the instruction pointer values when the called-back code exits to external library code via a `call` or `ret` instruction. We insert entry stubs for the external code to recovered code transitions, and exit stubs at recovered code to external code transitions. These stubs also perform the state marshalling mentioned in the previous paragraph. During the ELF stitching phase (Section 3.3.4), we insert code trampolines at the original virtual addresses of the called-back functions. Figure 3.4 visualizes the resulting control flow for a call to `qsort` which includes a simple callback.

If a static binary lifter attempted to use trampolines to handle callbacks as we have, they would lack the dynamic information about which functions are actually executed via callback. Without the dynamic information, the only sound approach would be to mark every function as a potential entry point. Creating many entry points to recovered code is deleterious to performance, as it increases code size and forces variables to be stored and reloaded.

Data Canonicalization

Accurately lifting data structures from binaries is a hard problem and the focus of orthogonal research [179]. Some architectures allow interleaving of code and data. This is true for ARM, but also for x86 where compilers often embed jump tables into code sections. We take a conservative approach by including data from the client binary as global variables in the IR, as well as copying any code sections in the binary that may contain data. We preserve their base mapping addresses in order not to invalidate existing references in the lifted code. We leave the task of applying existing analysis methods to split up the data into variables and creating typed references in the lifted code to future research. Thanks to our lifting engine, such analysis methods can benefit from strong data flow analysis at the level of compiler IR.

3.3.4 Lowering Lifted IR

After the client program IR has been transformed as desired, we produce a functional recovered binary. We use an unmodified LLVM compiler (`llc`) to generate a temporary ELF binary from the recovered IR. Then, our lowering toolchain stitches together ELF sections from the temporary binary and the original binary into one combined binary. We use the majority of sections from the temporary binary, and data sections from the original. Finally, we execute binary patching to insert the trampolines to support external callbacks (Section 3.3.3), and update dynamic linking structures (Section 3.3.4).

Dynamic Linking We lift all dynamic data and code references into canonical LLVM IR, and then lower this IR using LLVM's code generation infrastructure. This functionality requires us to redirect references to external functions and data used by the client binary. In addition to static references, we collect the dynamic addresses of every indirect load, which enables us to redirect those references to external symbols as well. We then ensure the dynamic linker operates on only lifted data structures, which is necessary given our atypical ELF layout. We utilize the ELF dynamic symbols section to determine the address of data symbols which will be filled by the dynamic linker. Even stripped binaries must retain this information. This approach could be extended to non-ELF binaries with minimal effort by implementing the API of the platform-specific dynamic linker. The real world benefit of dynamic linking support is that BinRec can support any off-the-shelf instrumentation scheme that acts via inserted calls to an external library. We use this functionality to enable the AddressSanitizer and SafeStack applications in Section 3.6.

3.3.5 Path Miss Handling

Binaries recovered with dynamic analysis may encounter unrecovered paths during testing or after deployment due to the coverage limitation of dynamic analysis (see Section 3.3.1). BinRec handles these control flow misses by forcing the recovered binary to invoke a *control flow miss handler* whenever it encounters an unrecovered path. Several control flow miss handlers are available.

The **log** hander logs the instruction pointer value that is missing from the recovered binary, and then aborts execution. This mode is useful when divergence between the recovered binary and the original is more dangerous than program termination.

The **fallback** handler diverts execution from the recovered code into the original code of the input binary. This involves marshalling of the emulated CPU state in the recovered code into the physical state of the original binary (see also Section 3.3.3), and then jumping to the original binary at the intended address. This miss handler is only available when the original binary and recovered binary target the same architecture. It is ideal for use cases that require program instrumentation without unexpected termination. Note that in a mitigation scenario, in which BinRec is used to augment lifted code with security instrumentation, this requires a binary-level mitigation for the remaining binary code. The binary mitigation may be heavyweight and hence inefficient. However, the fallback code is not expected to be on the hot path since it is not exercised by the lifting workload.

The **incremental lifting** handler feeds back the logged missing instruction pointers into the dynamic lifting engine, where we capture a trace covering the new control-flow edge, and merge it with the existing traces. Using this *incremental lifting* paradigm, the recovered binary can be continuously updated. Our current incremental lifting prototype lifts instructions until the next conditional control-flow transfer.

The recovered program can invoke the fallback miss handler, or the log handler. Meanwhile, the dynamic lifting engine can generate one or more new program traces via the logged instruction pointers in an asynchronous background process. We incorporate the new and existing traces to generate a new recovered binary.

An advantage of incremental lifting is it directly lifts new code without the need to reproduce the (explicit or implicit) input that triggers the miss during lifting. Consider a program feature that is only exercised due to unconstrained system randomness on the test system. There is no need to isolate and constrain the source of randomness to replicate it on the lifting system. Alternatively, there is no need to wait for non-deterministic fuzzing or concolic execution techniques to drive execution through the new paths.

Finally, when it is known that the tracing stage has already covered all paths that implement the features of interest, the miss handler can be optimized out completely. This is useful for aggressive optimization scenarios in which the lifting input is known to cover all necessary code, and eliminating a branch leads to new optimization opportunities.

3.4 Implementation

We implemented a prototype of BinRec, spanning 13,338 SLOC of which 9,709 are C++ code that implements lifting and canonicalization. The implementation targets single threaded 32-bit x86 binaries on Linux.

Our dynamic lifting engine is built on top of S^{2}E [43], a framework that facilitates symbolic execution of a single process running in the QEMU virtual machine [20]. Code is translated to LLVM IR in order to be symbolically executed by the KLEE symbolic executor [30]. S^{2}E automatically provides multi-architecture support and sandboxing of input binaries, since it is based on QEMU. This flexibility comes at the cost of a relatively long lifting time, which we discuss in Section 3.5.4.

3.4.1 Parallel Lifting

To address the scalability challenge (see Section 3.3.1), we architected BinRec with high parallelism. Dynamic tracing is expensive in time (due to dynamic binary translation) and disk usage (due to virtual machine images). We implemented a flexible run configuration scheme that allows operators to describe test cases to saturate a server's CPU and memory resources. Multiple traces through the same binary are lifted in parallel, and we can also lift different binaries in parallel.

The dynamic traces do not all have to be conducted at one time, so a lifted binary can be produced and used while more paths are being explored for the next version of the lifted binary. A dynamic trace is a stable artifact on disk that can be copied, shared, and reused. This allows the coverage of a binary to continuously be improved, and traces will not have to be regenerated.

3.4.2 Optimization

S^{2}E represents all instructions as modifications to a `struct` which stores the complete state of the original binary. This hinders existing LLVM passes from precisely analyzing and optimizing code. To address this issue, we optimize lifted code in several ways. First, our deinstrumentation described in Section 3.3.3 brings the code into a state where LLVM can perform existing optimizations including aggressive constant propagation and dead code elimination. Next, we guide the alias analysis with the fact that pointers to non-overlapping registers in the emulated register state cannot alias [64]. Third, we aggressively promote global variables representing the client binary state to equivalent local variables; even inlining functions that use them if it is favorable. Figure 3.5 shows the performance benefit obtained by applying our custom alias analysis and global variable promotion.

Stack unwinding optimization Client binaries often utilize error handling mechanisms such as setjmp and longjmp which save and restore the program state. Lifted binary programs have two contexts, the physical context of the recovered program, and the emulated context of the original program. Setjmp and longjmp calls in the original program should be translated to a save and restore of the emulated context in the recovered program. It would be possible to copy the emulated state to physical state, the same way we do for library calls, and thereby use the native setjmp/longjmp handlers. Instead, we implemented our own handlers to avoid the extra state copy by directly operating on emulated state.

3.4.3 Debugging

The current version of BinRec does not do anything with debug information. In principle, there are multiple useful vectors for its use as input or output. One vector would be to consume debug info in input binaries to aid analysis. This has been explored in prior work [72, 155, 168, 183]. Debug info is inherently static in nature, so using it for analysis would result in quite different capabilities, worthy of their own study. As static analysis is typically faster than dynamic analysis, it may accelerate certain analysis operations to use debug information.

Debug information could also be generated by a dynamic rewriting tool. There is potential to pass through initial debug information that was present in the input binary, or generate our own. If we wanted to pass through, the workflow could be as follows. Use a static process to parse debug strings and map them to addresses in memory. Then, attach the debug string for a particlar address to the lifted ir representation of the program, by look at the basic block address, or stores to the PC global variable. That would result in debug information in the liftd llvm ir assigned to the set of instructions that emulate the initial instruction the debug info was attached to. That debug info could be output by the regular llvm toolchain, and we could copy it into the final binary in elf stitching procedure. This construction would be extremely useful to debug both semantic bugs in the recompiled program, and bugs in the recompilation toolchain.

We implemented a prototype version of gdb integration and debugging info that was attched to recompiled programs, in order to debug our recompilation toolchain. The gdb integration was a script to print what values are in the emulated registers, when attached to a recompiled program. We are often interested in what functionality in the original binary is being emulated at a particular program point, and this script facilitated it. We also added a llvm pass for debug purposes to print the emulated stack contents.

3.5 Evaluation

In this section, we first compare our prototype against state-of-the-art static lifting approaches. We then assess the performance of programs lifted by BinRec in terms of run time and code coverage, as well as the lifting speed of our BinRec prototype. We use the SPEC CPU2006 benchmark suite, which is standard in the binary lifting literature [9, 18, 67], because it contains CPU-bound benchmarks, providing us with a pessimistic view of run-time overheads (as opposed to I/O-bound programs whose I/O performance is unaffected by lifting). We conducted our lifted binary run-time experiments on a system with 8GB RAM and an Intel i5-3210M running at 2.5GHz, with frequency scaling turned off to ensure stable performance. Lifting time experiments were conducted on an Intel Xeon E7-4870 @ 2.40GHz with 188 GB RAM. We used gcc 4.8.4 to compile all programs with optimization levels O0 and O3 (see Table 3.1). Our prototype is based on S^{2}E, which emulates floating-point instructions using integer instructions for portability. In this prototype implementation, we do not aim to optimize floating-point performance, so we limit our evaluation to the CINT subset of SPEC CPU2006.

3.5.1 Dynamic vs. Static Lifting

BinRec reliably lifts and recompiles a large number of real-world binaries. In addition to the qualitative benefits of our dynamic technique as discussed in Section 3.3, we investigated quantitative advantages of our approach. We compared BinRec to McSema [68] and Rev.ng [67], popular state-of-the-art binary lifting frameworks.[1] We limit our comparative study to active, open source binary

[1] Code snapshot on July 25th, 2019

```
1  void callback_func(j_common_ptr cinfo) {
2    printf(".");
3  }
4
5  int main (int argc, char **argv) {
6      struct jpeg_decompress_struct info; //jpeg info
7      struct jpeg_progress_mgr  progress;
8      ...
9      //After some initialization code
10     progress.progress_monitor = callback_func;
11     progress.pass_limit = 0x8048860;
12     progress.pass_counter = 0L;
13
14     info.progress = &progress;
15     jpeg_start_decompress( &info );
16
17     char *data = (char *) malloc(dataSize);
18     readData(info, data);
19     ...
20 }
```

Listing 3.2: Excerpt of *decompress.c*: libjpeg example in C.

lifters which, like BinRec, aim to be compiler-agnostic.

We found McSema [68] could only recover a limited number of binaries correctly in our tests. While trying to lift binaries compiled without optimization, we encountered errors with McSema's handling of double-precision floating point operations in 32-bit applications, unsupported xmm instructions (xmm xorpd, xmm andpd) on 64-bit, and segmentation faults in the C++ delete operator. In addition, some binaries lifted from compiler-optimized code caused segmentation faults upon launch or produced incorrect output.

We also identified cases where binaries generated by McSema interpreted data as code pointers, illustrating L1 in real-world code. McSema uses IDA Pro for control flow graph recovery and analysis. Hence, it is limited by IDA's inability to correctly identify function pointers in real-world code. This can lead to problems as illustrated by Listing 3.2: a structure type in libjpeg contains a member field that holds the address of a callback function (line 10), while another holds an integer that represents a loop bound (line 11) which happens to be in a similar value range. IDA is closed source, but we suspect it uses heuristics to identify integers with values in the executable segment as code pointers, which fails in this case. The recovered binary McSema generates from this program mistakenly changes the integer, thereby changing program semantics. Similarly, failure to identify code pointers correctly could cause mishandling of callbacks in this program. Unfortunately, the authors do not provide any performance numbers for correctly lifted binaries using McSema.

We were unable to recover most of the dynamically linked SPEC INT2006 binaries with Rev.ng [67]. While we managed to get some of the binaries running by reducing the optimization level to O0 (a classic example of L4—due to aggressive optimization), this still yielded mixed results. For instance, the tool was able to produce a lifted version of *libquantum* but its output differed from the output of the original program. The only test that was correctly recovered was *mcf*. Some tests failed completely (even at O0), e.g., *gcc*, *gobmk*, *perlbench*, and *xalancbmk*.[2] Table 3.1 compares the performance of BinRec to Rev.ng using the most recent published results [85]. The authors note that these were all statically linked. Although their client binaries' optimization level is not specified, BinRec's performance (0.98x for O0, 1.29x for O3) exceeds Rev.ng's (2.25x) in either case.

In summary, both state-of-the-art tools we looked at were unable to reproducibly recover even standard binaries, despite being actively developed and widely used open-source frameworks for binary lifting. We would like to stress that this does not reflect a lack of sophistication behind those tools (or the developers), but instead highlights the tremendous difficulty faced by static lifting approaches. Crucially, we found our dynamic tracing technique to aid the lifting process within BinRec significantly: we are able to recover all of the test binaries in question while the recovered binaries performed favorably by comparison and produced correct output.

3.5.2 Performance

Table 3.1 presents the performance of binaries lifted with BinRec. For every input program we compiled both optimized (O3) and unoptimized (O0) binaries which produce correct output in

[2] The error message indicated failed assertions in the IsolateFunctionsImpl class upon replacement of indirect branch targets, strongly hinting towards an instance of L2. We contacted the developers but did not get any detailed feedback in time for the submission.

Table 3.1: Measured execution time normalized to the original binaries. Rev.ng results are reported from publication [85].

	BinRec		**McSema**		**Rev.ng**
Benchmark	**O0**	**O3**	**O0**	**O3**	**reported**
400.perlbench	1.25	1.48	–	–	3.7
401.bzip2	0.76	1.05	2.84	–	2.2
403.gcc	1.26	1.37	–	–	2.1
429.mcf	0.83	1.00	2.31	1.41	1.5
445.gobmk	1.04	1.56	–	–	3.3
456.hmmer	0.77	0.74	–	–	2.2
458.sjeng	0.77	1.08	3.43	–	2.6
462.libquantum	0.95	1.30	2.07	1.04	1.1
464.h264ref	0.80	1.24	–	–	2.7
471.omnetpp	1.92	3.09	–	–	2.8
473.astar	0.80	0.94	–	–	1.5
483.xalancbmk	1.12	1.66	–	–	2.8
geomean	0.98	1.29	–	–	2.25

the test cases. Our results show that there is a potential for performance improvement by using BinRec as a post-release optimizer—particularly, if the original was not optimized at the source level. With BinRec, six benchmarks – *bzip2*, *mcf*, *hmmer*, *sjeng*, *h264ref*, and *astar* – run faster than the unoptimized client binaries. In some cases, BinRec can re-optimize release builds to be faster than even the optimized binaries (e.g., *hmmer* and *astar*). Compared to the *optimized* client binary, the *hmmer* "nph3.hmm swiss41" workload finished in 0.62x the time. Interestingly, *hmmer* is the only SPEC binary to be faster when re-optimized from an optimized (0.62x) rather than an unoptimized client binary (0.85x).

There are factors that accelerate and factors that slow down programs recovered by BinRec. We discussed several of the accelerating factors in Section 3.4.2 and show their benefit in Figure 3.5. Floating point instructions are emulated in the lifted binaries, which incurs a performance penalty (e.g., we found this to be one of the main factors for the slowdown of *omnetpp*). Further, the IR of recovered programs contains less accurate information about the size and lifetime of stack allocations compared to source code, which impedes optimization. The geometric mean run time factor of BinRec binaries compared to unoptimized and optimized input binaries is 0.98x and 1.29x, respectively.

In addition to the performance of rewritten but uninstrumented binaries, users of binary rewriting

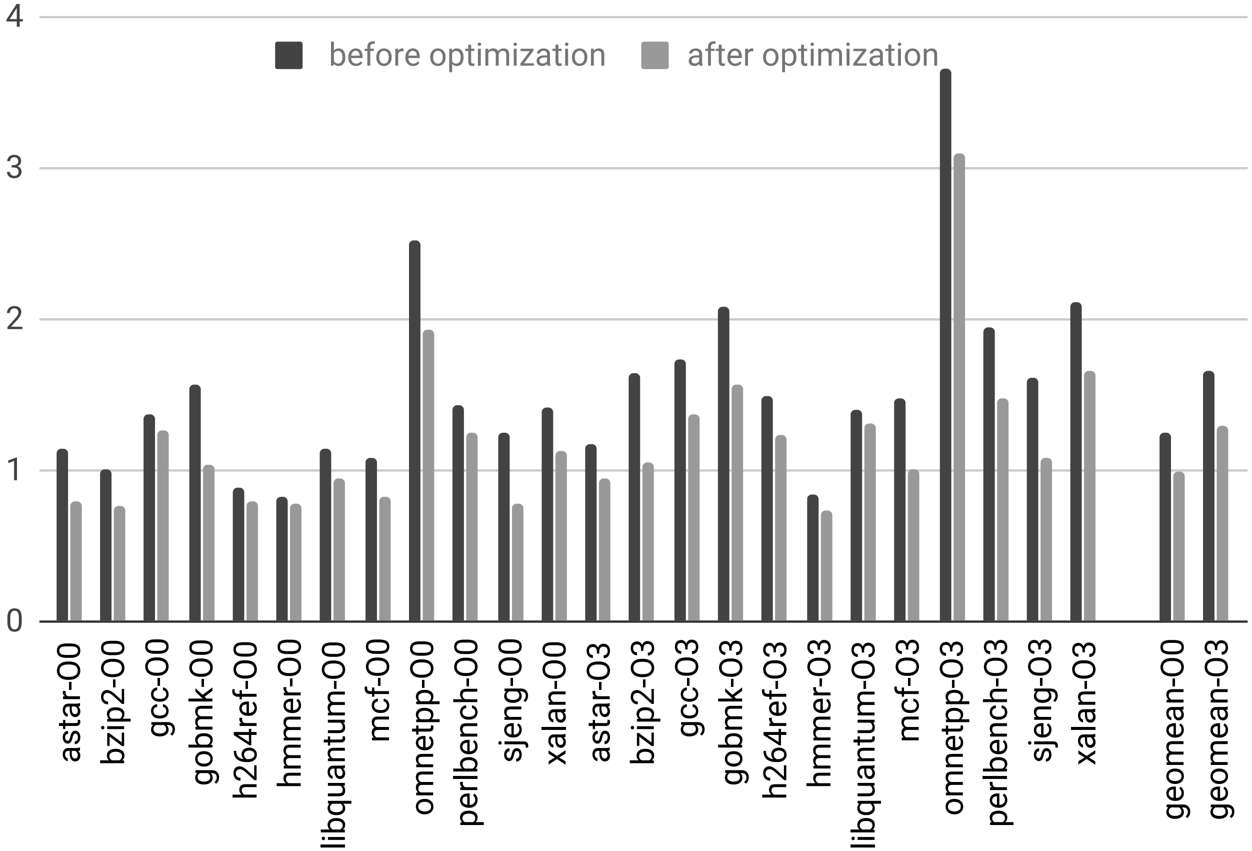


Figure 3.5: Execution time improvement from CPU state variable de-aliasing and global variable promotion.

frameworks care about the runtime performance with added hardening transformations and instrumentation. Multiverse, while able to support a wide variety of programs and transformations, suffers from particularly slow performance with instrumentation. This is because it must support a huge variety of possible program paths, due to its nature of deferring determination of the correct path till runtime, and the instrumentation must work on any of these paths. In contrast, BinRec does not have to support paths that will not be taken, since we record what paths will be dynamically taken. In addition, we can optimize the instrumentation using LLVM in BinRec. It would be interesting future work to develop standardized benchmarks for the overhead with a fixed set of instrumentation in a set of fixed programs.

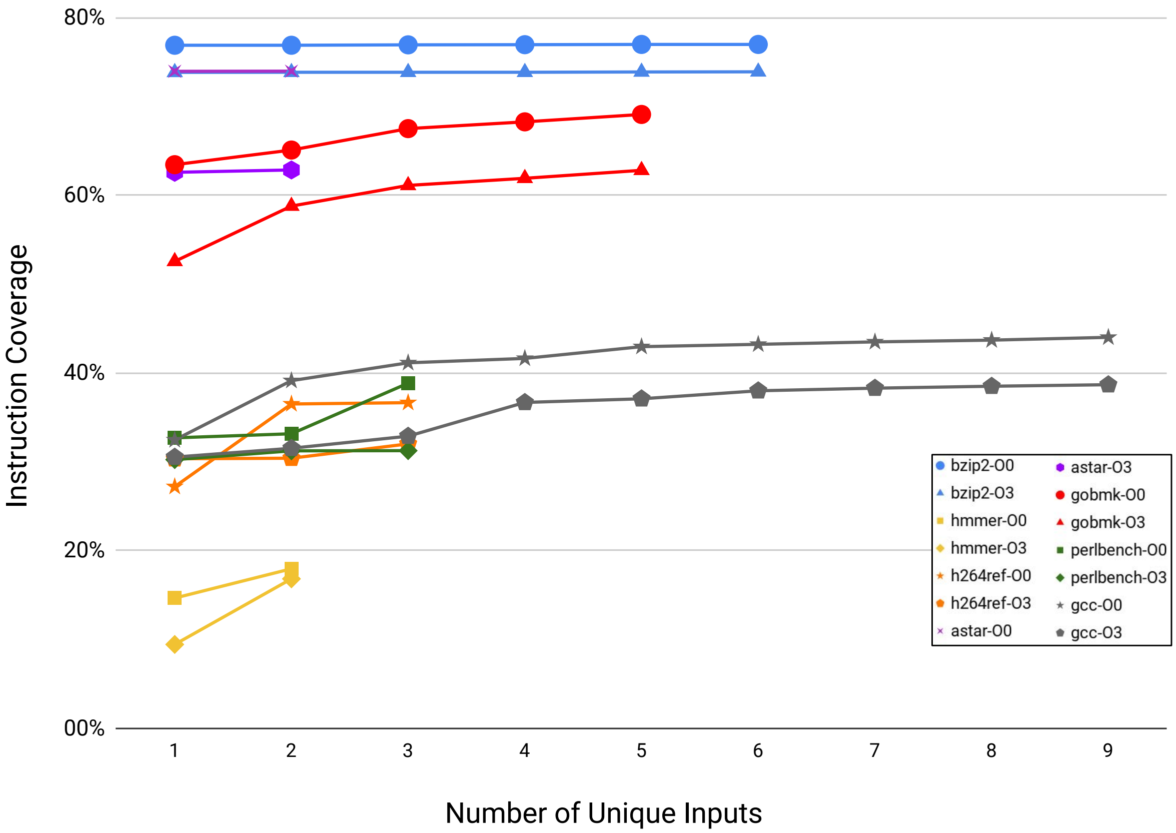


Figure 3.6: Coverage with respect to the original binaries. The input set is the *ref* workload of SPEC CPU2006.

3.5.3 Code Coverage

Figure 3.6 shows the instruction coverage of lifted binaries as we increase the number of supported input workloads. The rate of coverage change is substantially different between binaries, and reflects both the number of unrelated features in the binary and the similarity of the test cases. *bzip2*, for instance, exercises nearly the same code path for each input. In contrast, *gobmk* and *gcc* see a steady increase in code coverage for each added input. The level of instruction coverage should therefore be dependent on the application, lifted feature set, and use case. Users of our framework may aim to increase coverage or to keep it low, limiting the attack surface for attackers. In both cases, BinRec's ability to report code coverage provides the user with a practical metric to determine if incremental lifting is effective; either in maintaining low coverage or in increasing coverage.

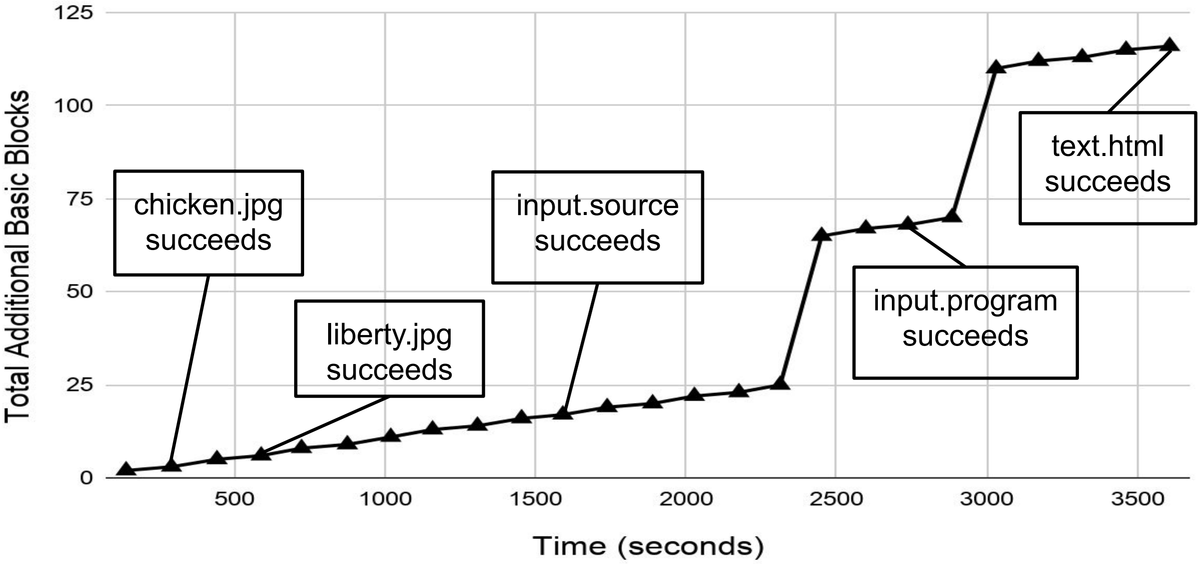


Figure 3.7: Incremental lifting progression of *bzip2*.

Incremental Lifting To show the effectiveness of incremental lifting, we conducted an experiment with *bzip2* as illustrated in Figure 3.7. We first lifted the binary with SPEC `training` inputs, which is the origin point of the graph. Then, we ran the lifted binary with `reference` inputs and incrementally lifted code to support each new input. The callouts on Figure 3.7 indicate when each additional input became functional in the recovered binary. Each triangle represents one cycle through the lifting frontend, and each cycle took approximately 140 seconds.

3.5.4 Analysis Time

BinRec's ability to robustly lift binaries without relying on heuristics comes at the cost of lifting time. As a dynamic lifting tool, BinRec's lifting time depends on the execution time of its input programs. Table 3.2 shows BinRec's lifting times for each input binary. In order to show the worst case lifting time, we used a SPEC reference input—which fully excercises loop iterations—for lifting. The lifting could be much faster with an optimized trace input which is designed to reach more paths and minimize loop counts, but such an optimization would not reflect real world workloads. Since the current prototype of BinRec uses S^{2}E [43] as its tracing frontend, we also present the time to

Table 3.2: Time in seconds to capture LLVM IR from input binaries with BinRec and McSema toolchains, alongside execution time for S^{2}E without BinRec instrumentation. For BinRec and S^{2}E we report the maximum time among the reference workloads from SPEC CPU2006.

	BinRec		**S^{2}E**		**McSema**	
Benchmark	**O0**	**O3**	**O0**	**O3**	**O0**	**O3**
400.perlbench	425,619	321,078	62,482	49,221	3,375	3,385
401.bzip2	86,181	69,389	27,614	18,311	117	122
403.gcc	37,276	28,468	6,156	4,929	6,996	7,378
429.mcf	283,413	227,999	209,914	197,910	11	8
445.gobmk	84,214	72,307	15,496	8,721	1,332	1,063
456.hmmer	179,127	144,529	87,911	28,159	204	189
458.sjeng	727,675	548,432	95,936	86,153	294	368
462.libquantum	421,269	176,536	–	49,334	21	16
464.h264ref	86,433	65,202	31,012	15,233	336	586
471.omnetpp	–	312,665	–	105,015	258	224
473.astar	211,782	119,436	80,613	66,201	22	18
483.xalancbmk	–	–	–	–	74,948	17,103
geomean	178,480	138,379	44,810	35,021	371	320

execute those workloads without instrumentation in S^{2}E. The lifting time of static lifting toolchains, such as McSema, does not depend on the input, and is in general faster than our dynamic approach. We present the lifting time which we collected with McSema here for comparision.

Binary lifting is a one-time, offline process and thus it does not affect performance of actual binary execution. If fast lifting times were in fact desired, it could be accomplished using a faster tracing frontend such as Pin, KVM-enabled QEMU, or native execution with a hardware control-flow tracing feature (e.g., Intel PT). In that case, however, we may miss the flexibility of disassembly in S^{2}E, and its ability to explore multiple code paths through concolic execution.

3.6 Applications

One of this project's main goals is to enable complex transformations on real-world program binaries. Since BinRec can lift binaries to a compiler-level IR and supports dynamic linking, this enables us to make use of a large ecosystem of off-the-shelf compiler-based transformations and analysis tools. In addition to compiler transformations, existing, black box binary utilities such as `readelf` or `LD_PRELOAD` remain usable on binaries recompiled with BinRec. In this section, we showcase some applications that demonstrate this ability: deobfuscation, AddressSanitizer and SafeStack through compiler transformations, and control-flow hijacking mitigation. Developers who are familiar with these transformations do not need any knowledge of binary analysis to use them within our framework. While we only provide a limited set of example applications in this section, we note that BinRec reliably enables—for the first time—a large number of interesting, feature-rich program analyses and transformations through extensive compiler-based tooling for binary programs.

3.6.1 Control-flow Hijacking Mitigation

Even without additional compiler-based transformations, BinRec has an endogenous ability to mitigate memory-corruption vulnerabilities in the original program. A recovered program emulates

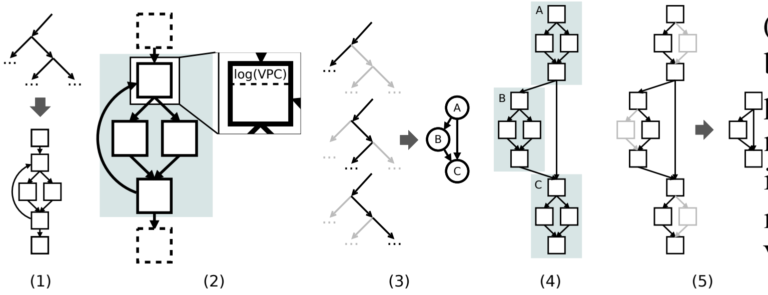


Figure 3.8: Our deobfuscation approach. (1) We lift the binary using symbolic execution or high-coverage inputs. (2) We identify the lifted interpreter loop and instrument it to log the virtual program counter (VPC) at the entry. (3) The instrumented binary is exercised for all uncovered code paths, yielding a control-flow graph of VPC nodes. (4) The interpreter loop is copied into each VPC node. (5) Standard optimizations eliminate non-taken paths in each VPC node.

the execution of the original program. Because of the emulation, what was control flow in the original program becomes data flow in the recovered program. BinRec does not natively mitigate data-only attacks [38], though they may be mitigated using additional transformation on the IR.

A control-flow hijacking attack typically subverts control flow by overwriting a code pointer. This pointer could be used by an indirect jump, indirect call, or return instruction. When tracing indirect control flow in the original program within BinRec, we observe actual control flow targets. The recovered program then contains switch statements where cases are jumps to these observed targets. The value of the instruction pointer (`%rip`) in the original program is emulated by the recovered program, and it is used as the index into the switch statement. The switch statements are lowered into assembly consisting of trees of compare and direct jump instructions, so no new attack surface is introduced by this dispatch mechanism. This is functionally equivalent to what is commonly

Table 3.3: Number of allowed targets for indirect branches/calls in SPEC CPU2006 binaries lifted by BinRec, compared to the number statically found by BinCFI [212].

	O0				**O3**			
	BinRec CFI			**BinCFI**	**BinRec CFI**			**BinCFI**
Benchmark	**Median**	**IQR**	**Max**		**Median**	**IQR**	**Max**	
400.perlbench	5	7.5	176	2,101	4	7	176	1,916
401.bzip2	3	0	22	151	3	0	22	117
403.gcc	4	3	212	6,593	3	3	212	5,407
429.mcf	3	1	7	68	3	0	7	66
445.gobmk	3	0	492	2,780	3	0	492	2,590
456.hmmer	3	0	8	671	3	0	7	620
458.sjeng	4.5	3	12	223	5.5	3.3	12	215
462.libquantum	3	0	2	177	3	0	5	161
464.h264ref	3	0	10	686	3	0	10	617
471.omnetpp	3	4	168	3,167	3	1.3	168	2,482
473.astar	3	0	3	213	3	0	4	139
483.xalancbmk	3	4	38	35,106	4	3	38	15,950

IQR: inter-quartile range

known as context-insensitive control-flow integrity on forward and backward edges [28].

Consider an original binary with a vulnerable stack buffer overflow using an unsanitized `strcpy` call, that can be used to overwrite a return address. In the recovered program, that buffer is located in a `@memory` array which emulates the memory of the original program. The `strcpy` call will proceed in the same way in the recovered program, allowing the attacker to overwrite the emulated return address of the vulnerable function. The original return instruction is emulated using a switch statement, loading an attacker-provided value via the emulated register `@RIP`. If the target is not one of the traced return sites of the vulnerable function, the error case of the switch statement will abort execution. Otherwise, execution will proceed in the style of a control-flow bending attack [33], since the target address represents a valid execution under context-sensitive (but not path-sensitive) analysis of the original program. If optimization is applied to the recovered binary and the error case is deleted from this switch statement, one of the observed return targets of the vulnerable function will be chosen in a compiler-specified manner. In this case, an attacker aware of BinRec could still perform control-flow bending. However, any attempt to hijack control flow via writing code pointers (*vtable* overwrite, indirect code pointer write via heap overflow, etc.) is mitigated.

To evaluate the security properties of the resulting solution, we measured the number of allowed targets across all the recovered edges. Though our approach also protects returns, we only present forward edges in Table 3.3 for easier comparison with other approaches. Our results show that BinRec can enforce a median number of around 3 indirect callees on a nontrivial fraction of the target programs. The table also shows these results for binCFI [212], a static binary-level CFI solution. Because it can not statically predict valid branch targets with precision, binCFI's policy must allow transfers to any address-taken function, increasing the number of allowed branch targets by orders of magnitude when compared to BinRec.

3.6.2 Virtualization-deobfuscation

We used BinRec to lift programs obfuscated by virtualization (cf., Section 3.2.5). Figure 3.8 illustrates our deobfuscation approach. For this use case, we detect the Virtual Program Counter (VPC) and virtual interpreter loop through known techniques [174]. We instrument the recovered IR to log the value of the VPC at the entry point of the interpreter loop, then produce a binary. We exercise the instrumented binary to obtain a graph of VPC nodes. We create a new program from this graph by copying the body of the virtual interperter loop into the VPC nodes. After applying standard compiler optimizations (most notably constant propagation and dead code elimination), only one virtual opcode handler remains for each duplicated interpreter. The result is a program with the semantics and static structure of the original program; the virtualiztion obfuscation has been removed.

To evaluate our deobfuscation approach, we implemented a virtualizer that supports a set of bytecode instructions. We then created a source-to-source virtualization-obfuscated version of two simple programs: `eq` checks if two arguments match, and `fib` computes the n-th Fibonacci number. Table 3.4 shows how deobfuscation affects the size of the recovered code. We attain a code size close to that of IR recovered from the unobfuscated binary. Figure 3.9 depicts the `fib` program, showing its control flow graph obfuscation and subsequent deobfuscation.

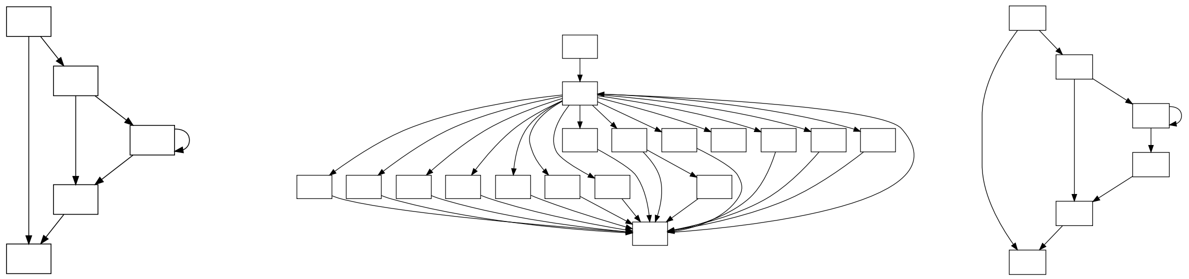


Figure 3.9: Deobfuscation of the `fib` program. The control flow graph structure of the deobfuscated binary matches that of the original bytecode, rather than that of the interpreter, which indicates the control flow obfuscation was successfully removed by the analysis implemented in BinRec.

Table 3.4: The number of LLVM instructions: after lifting, after optimization without deobfuscation, and after deobfuscation and optimization. The baseline is the number of LLVM instructions obtained by compiling the unobfuscated program with `clang`.

	Lifted	Optimized	Deobfuscated	Baseline
`eq`	2,362	152	35	38
`fib`	3,163	210	63	43

3.6.3 AddressSanitizer

AddressSanitizer (ASan) is a widely deployed bug finding tool that detects spatial and temporal memory errors [170]. It consists of an LLVM instrumentation pass and a run-time monitor. The ASan instrumentation pass identifies and registers memory allocations, and inserts checks for memory accesses. For binaries lifted by BinRec, all memory reads and writes are identified and instrumented automatically using the unmodified ASan instrumentation pass. Heap allocations (e.g., `malloc` or `new`) are recorded in the BinRec lifting frontend and rewritten in the recovered IR, making them visible to ASan. We leave the identification of stack and global allocations for future work as the problem is currently unsolved for binaries. While ASan has been applied to binaries recently [69], we note that this required a re-implementation of both the analysis and instrumentation passes—a substantial disadvantage in maintainability compared to BinRec. Our recovered IR enables the use of ASan to detect spatial and temporal heap access violations. We used two test programs containing (1) a heap use-after-free error and (2) an out-of-bounds write and lifted both test programs in BinRec before applying ASan, successfully discovering these errors.

3.6.4 SafeStack

SafeStack is a compiler-based transformation pass that separates sensitive data, such as return addresses, and potentially insecure data, such as large application buffers, into separate stacks [105]. If memory isolation features such as x86 segmentation or Intel Memory Protection Keys are available, they are used to isolate the two stacks. If hardware features are unavailable, SafeStack leverages ASLR to hide the safe stack, requiring attackers to bypass ASLR in order to corrupt sensitive data.

By default, BinRec generates programs which contain two stacks with SafeStack-like security properties. The native stack contains sensitive data such as register spills and return addresses, as well as any new instrumentation and library code frames. The emulated stack, which contains the stack data of the original binary, resides at an ASLR-randomized location.

We were additionally able to apply SafeStack's transformations to recovered programs without requiring any modifications to its analysis or transformation passes, since BinRec lifts programs to well-formed LLVM IR. After the SafeStack transformation, recovered programs therefore contain three stack-like memory regions. The native stack contains library frames and newly added safe variables. The emulated stack, at an ASLR-randomized offset, emulates the original binary stack. A third stack in a separate x86 memory segment contains new, potentially insecure buffers. We do not identify stack variables within the original binary, which impedes the transformation's ability to move unsafe buffers from the emulated stack to the third, segmented stack (see Section 3.7).

3.7 Limitations of the Prototype Implementation

Our prototype implementation of BinRec can only handle single threaded x86 ELF binaries. There are no theoretical limitations on threaded-ness or architecture. The architecture constraint comes from the engineering effort required to implement inline assembly snippets, mostly for library code interfacing. Additional snippets are required for the fallback path miss handler, and floating point arithmetics. The calling convention abi must be well understood and implemented to achieve these operations with efficiency. For example, in a library call, callee saved values do not need to be marshalled from emulated to concrete state, only parameters and return values need to be marshalled.

Supporting other binary formats such as PE is no fundamental problem, but requires reimplementing binary stitching in the new format. In particular, the decision of which sections to use from the original binary vs the rewritten binary would need to be revisited. The majority of extensible dynamic linking support in the compiler passes should be reusable, but the ELF manipulation would need to be modified.

It would require a modest amount of additional engineering effort to implement support for multi-threaded programs. Recording of control flow successors should be done on a per thread basis, to avoid control flow edges being erroneously inferred between unrelated executions in different thread contexts. S2E contains support for OS level threads, and these could be used to separately collect control flow successors. For user space thread, and cooperative multi-tasking, further study is needed. However, collecting traces through such a program should result in a lifted program that executes one valid serialized execution of the many possible schedules possible in the original program. To ensure accurate CFG recording in the presence of multi-processing and multi-threading, the data structures used to record CFG successors should be thread safe.

We did not implement handling of self-modifying code. To support it, we would need to add 'version labels' to each recovered code address. Our existing labels use the address of recovered code to uniquely identify them. In the presence of self-modifying code, we would additionally need a temporal label. One possible temporal label is trip count through the code address. Using temporal labels would incur some additional lifting time (because code cannot be cached). It would also add complexity while merging traces into one CFG. It is possible that each trip through a code address would have different code. The control flow graph would therefore be tree-shaped. On the other hand, the code at a certain address may never change, ie, that code is not self-modifying. We would need to do code clone detection to determine if CFG nodes which represent the same address at different time points should be merged. A simple diff based clone detection may be sufficient, since the candidates are generated programmatically.

BinRec does not recover a mapping between stack slots and variables. Such a mapping would improve optimization and allow more fine grained instrumentation by transformations such as

SafeStack and ASAN. SecondWrite [9] determined such a mapping for a limited set of input binaries using heuristics, but we leave the determination of a general procedure as future work. See Section 3.8.

Hardware backed trusted computing paradigms like SGX mean the dynamic analysis will not be able to observe or modify the interesting code in the enclave. Anti-debugging techniques make tracing of the program more difficult. Anti-fuzzing [84] techniques will limit the helpfulness of fuzzing and symbolic-execution for driving tracing, placing more emphasis on manual test case selection.

3.8 Future Work

3.8.1 Symbolization

The process of determining which memory locations in a binary represent semantic variables in the source program at different times has many facets and is known by many names. We will refer to it as symbolization. The difficulty of the problem is greatly determined by what auxilliary information is available (debug symbols) and what assumptions are made (frame layout). In BinRec, we tried to solve the problem with no heuristics or assumptions, and no auxialliary information. We attempted symbolization in one analysis domain, stack locations with both static and dynamic approaches. We did not yet achieve effective solutions, and greatly encourage further study of the area.

The benefit of symbolization is it results in more precise data-flow relations. These relations can be used to aid optimization, or to apply a precise security policy.

In the context of BinRec we approached the problem of stack symbolization in detail. Stack symbolization is the subset of the symbolization problem which deals with stack allocated variables. The main difficulties for stack symbolization are temporal aliasing via stack frame pushes and pops, and dynamic access to arrays on the stack. If a stack frame has been released, and another one has been reallocated in its place, then the same memory address now refers to a different symbol. This is a form of temporal aliasing. Without a stack symbolization analysis, each dynamic memory access may alias with every stack access. Therefore, all stores to the stack must be completed before any dynamic access, and no loads may be hoisted before any dynamic access.

In some architectures and programming paradigms, a function may access any memory location on the stack, even outside its own frame. A heuristic which simplifies the stack symbolization problem is to assume functions only access their own stack frame, and parameters passed to them in their parent's frame. This heuristic is based on the traditional way that compilers emit code. It was used in Secondwrite[9]. Secondwrite made an attempt and had good success at symbolizing stack variables within the subdomain they chose which was well formed compiler generated stack frames, no position independent code, and no inline assembly.

With or without the use of this heuristic, a stack symbolization routine should detect when a frame as has been deallocated, and therefore mark the end of the lifetimes of the variables contained within. The movement of the stack pointer in a way consistent with the way a compiler emits stack frames is a heuristic that can be used to detect the deallocation of a stack frame. However, in the face of arbitrary assembly, is is not easy to determine when the lifetime of a stack allocation ends. A conservative choice is to assume a value is live until it is overwritten, however, this choice will impede downstream optimization, and is such a low level abstraction it is not easy for programmer to reason about. A symbol under this assumption is not like a symbol in the source code.

Another program construct which impedes stack variable symbolization is dynamic access to stack allocated arrays. Such accesses are via a base address and an offset. The bounds of the array are typically not recoverable from the binary. The problem is therefore, to determine which locations on the stack may be accessed by an instruction which uses a base plus offset. Under the heuristic where functions can only access their own frames, dynamic accesses only alias variables in frame containing their base address. One way to tighten the bound on which locations will be accessed via a dynamic memory access is to use value set analysis to determine bounds on the offset. If the offset is bounded, any variables stored outside of that bound will not be accessed, and therefore they will be unchanged by the memory access.

Another way to view the optimization possibilities of symbolization is to remove loads and stores, hoist loads, and sink stores. If a variable is symbolized, we guarantee that there are no reads to it or writes of it within a region of code. The value can be kept in a register, instead of being stored back to memory after every operation. Yet, another way to look at the symbolization problem is as an extension of the single static assignment [55], scalar replacement of aggregates (SROA transformations [132]). Instead of the traditional `alloca` stack storage locations which are the usual input to the SSA transformation, each physical stack slot is an initial storage location. Each initial storage location may be representable by a symbol in SSA form.

We are in the early stages of a prototype symbolization scheme using dynamic analysis. Dynamic analysis has the potential to fully symbolize the stack without heuristics, however, it has a key caveat. The symbolization will only be correct as long as the data flows in the rewritten program follow the traced data flows. There can be a spectrum of correct program rewriting based on dynamic symbolization. At one extreme is to make no modifications to the data access pattern. At the other is to hoist loads as early as possible and sink stores as late as possible given the data flow observed in they dynamic traces. The longer data remains un-stored to memory, the more likely that data flows on unobserved traces would conflict with the rewritten data flow. This is related to the fact that a rewritten program will only be correct when the control flow in the rewritten program follows the control flow in the traces. The space of all possible data flows is distinct from all control flows, and one does not necessarily imply the other.

3.9 Related Work

Binary rewriting and analysis Many projects target the problem of low-level binary analysis and rewriting. PEBIL [111], UQBT [45] and Uroboros [196] all statically rewrite binary programs either at the machine code level or using a custom low-level IR. Their main aim is to support the insertion of simple instrumentation, where efficiency is more important than the ability to perform complex code transformations (such as altering the CFG). angr [178] supports static and dynamic analysis techniques, including symbolic execution, but does not target code rewriting (unlike BinRec). Earlier work such as ATOM [72], PLTO [168], Diablo [155], and Vulcan [183] are powerful tools, but to our knowledge they do not work well without debug symbols. Also, they typically do not support a generic compiler-level IR.

Bauman et al. [18] disassemble instructions from every offset of code sections, creating a superset of all possible disassemblies. They statically rewrite binaries without heuristics by preserving the superset of disassemblies, such that only the legal part of the rewritten binary will be executed at run time. However, deferring correct disassembly until runtime adversely affects rewritten binary performance. Yardimci and Franz [206] use a mostly static approach to automatically vectorize loops in stripped binaries. The approaches of both Yardimci and Bauman both use an indirect branch table which maps original program addresses to rewritten program addresses to support indirect control flow. BinRec uses a similar indirect branch table for external callback support, but it generates more optimal code because only those callbacks which are actually invoked need branch table entries and control flow graph entry points in rewritten code.

DBT and JIT compilation Dynamic instrumentation tools such as PIN [119], Dyninst [27], DynamoRIO [26] and Valgrind [136] are dynamic binary translation (DBT) tools, providing runtime APIs to analyze and instrument code at run time. These tools do not support saving the changes to an output binary with the intent of replacing the original binary. They can have substantial runtime overhead [144], and require specific assembly-level transformation passes for each application, whereas BinRec leverages existing techniques present in production compilers.

Just-in-time (JIT) compilers such as V8 [82] and SpiderMonkey [131] collect dynamic traces to determine which code to optimize and to speculate dynamic data types. Sulong [160] is a frontend for the Graal compiler that effectively creates an LLVM bitcode execution engine. Similar to BinRec, Sulong optimizes LLVM bitcode using dynamic traces and applies instrumentation such as bounds checks to detect safety violations. However, such source-level JIT compilation approaches leverage language semantics and thus do not address the problem of binary lifting or analysis. Instead, they focus on solving a different set of problems such as how to optimize dynamic type checks or when to trigger different tiers of execution.

Binary code lifting LLBT [176, 177] statically retargets binaries to different ISAs after lifting them to LLVM IR. McSema [68], Dagger [24], Rev.ng [67] and RevNIC [42] (based on S^2E) and SecondWrite [9] lift machine code for the purpose of high-level static binary translation on LLVM IR.

HQEMU [92] extends QEMU's back-end to lift code to LLVM IR similarly to S^2E, for the purpose of optimization. It does not decouple lifted code from the QEMU runtime to produce a standalone executable binary, like BinRec.

Our prior work, a short workshop paper [103] presents a high-level idea of dynamic binary lifting. This prior work constructs a rewritten binary from a single dynamic trace, which in turn fails to produce a binary that covers a whole targeted input corpus. This version addresses this issue i) by stitching multiple, parallel traces into a single executable binary; ii) by incrementally recovering missing basic blocks and control flow edges from the original binary. Compared to our previous work, BinRec shows evaluation results with the complete set of SPEC CINT2006 benchmarks, with significant performance improvement due to our new optimized alias analysis. Furthermore, the prior work was solely targeted for attack surface reduction and it does not present a mechanism to modify the dynamic linkage of their input binaries, limiting code instrumentation. This extended work, on the other hand, shows effectiveness with a rich set of applications including virtualization-deobfuscation, AddressSanitizer, SafeStack, and a control-flow hijacking defense. Moreover, it outlines unsolved challenges of static disassembly in the context of binary lifting.

3.10 Conclusion

We presented a new solution for binary lifting based on dynamic analysis. Our prototype, BinRec, lifts a program to compiler-level intermediate code for ease of analysis, while ensuring that it can still compile the result to executable code. Compared to existing static analysis-based techniques, dynamic approaches can seamlessly handle indirect control flow transfers, handwritten assembly and obfuscations. We designed BinRec to overcome the coverage issue of dynamic analysis by using trace merging and incremental recovery. We demonstrated the powerful applications made possible by BinRec: recovering program semantics of virtualization-obfuscated binaries, and applying compiler-level optimizations and hardening transformations to stripped binaries.

Chapter 4

Control-Flow Integrity: Precision, Security, and Performance

4.1 Abstract

Memory corruption errors in C/C++ programs remain the most common source of security vulnerabilities in today's systems. Control-flow hijacking attacks exploit memory corruption vulnerabilities to divert program execution away from the intended control flow. Researchers have spent more than a decade studying and refining defenses based on Control-Flow Integrity (CFI), and this technique is now integrated into several production compilers. However, so far no study has systematically compared the various proposed CFI mechanisms, nor is there any protocol on how to compare such mechanisms.

We compare a broad range of CFI mechanisms using a unified nomenclature based on (i) a qualitative discussion of the conceptual security guarantees, (ii) a quantitative security evaluation, and (iii) an empirical evaluation of their performance in the same test environment. For each mechanism, we evaluate (i) protected types of control-flow transfers, (ii) the precision of the protection for forward

and backward edges. For open-source compiler-based implementations, we additionally evaluate (iii) the generated equivalence classes and target sets, and (iv) the runtime performance.

4.2 Introduction

Systems programming languages such as C and C++ give programmers a high degree of freedom to optimize and control how their code uses available resources. While this facilitates the construction of highly efficient programs, requiring the programmer to manually manage memory and observe typing rules leads to security vulnerabilities in practice. Memory corruptions, such as buffer overflows, are routinely exploited by attackers. Despite significant research into exploit mitigations, very few of these mitigations have entered practice [185]. The combination of three such defenses, (i) Address Space Layout Randomization (ASLR) [150], (ii) stack canaries [188], and (iii) Data Execution Prevention (DEP) [11] protects against *code-injection attacks,* but are unable to fully prevent *code-reuse attacks*. Modern exploits use Return-Oriented Programming (ROP) or variants thereof to bypass currently deployed defenses and divert the control flow to a malicious payload. Common objectives of such payloads include arbitrary code execution, privilege escalation, and exfiltration of sensitive information.

The goal of Control-Flow Integrity (CFI) [5] is to restrict the set of possible control-flow transfers to those that are strictly required for correct program execution. This prevents code-reuse techniques such as ROP from working because they would cause the program to execute control-flow transfers which are illegal under CFI. Conceptually, most CFI mechanisms follow a two-phase process. An *analysis* phase constructs the Control-Flow Graph (CFG) which approximates the set of legitimate control-flow transfers. This CFG is then used at runtime by an *enforcement* component to ensure that all executed branches correspond to edges in the CFG.

During the analysis phase, the CFG is computed by analyzing either the source code or binary of a given program. In either case, the limitations of static program analysis lead to an over-approximation of the control-flow transfers that can actually take place at runtime. This over-approximation limits the security of the enforced CFI policy because some non-essential edges are included in the CFG.

The enforcement phase ensures that control-flow transfers which are potentially controlled by an attacker, i.e., those whose targets are computed at runtime, such as indirect branches and return instructions, correspond to edges in the CFG produced by the analysis phase. These targets are commonly divided into forward edges such as indirect calls, and backward edges like return instructions (so called because they return control back to the calling function). All CFI mechanisms protect forward edges, but some do not handle backward edges. Assuming code is static and immutable [1], CFI can be enforced by instrumenting existing indirect control-flow transfers at compile time through a modified compiler, ahead of time through static binary rewriting, or during execution through dynamic binary translation. The types of indirect transfers that are subject to such validation and the number of valid targets per branch varies greatly between different CFI defenses. These differences have a major impact on the security and performance of the CFI mechanism.

CFI does not seek to prevent memory corruption, which is the root cause of most vulnerabilities in C and C++ code. While mechanisms that enforce spatial [134] and temporal [135] memory safety eliminate memory corruption (and thereby control-flow hijacking attacks), existing mechanisms are considered prohibitively expensive. In contrast, CFI defenses offer reasonably low overheads while making it substantially harder for attackers to gain arbitrary code execution in vulnerable programs. Moreover, CFI requires few changes to existing source code which allows complex software to be protected in a mostly automatic fashion. While the idea of restricting branch instructions based on target sets predates CFI [100, 101, 149], Abadi et al.'s seminal paper [5] was the first formal description of CFI with an accompanying implementation. Since this paper was published over a

[1] DEP marks code pages as executable and readable by default. Programs may subsequently change permissions to make code pages writable using platform-specific APIs such as `mprotect`. Mitigations such as PaX MPROTECT, SELinux [122], and the `ProcessDynamicCodePolicy` Windows API restrict how page permissions can be changed to prevent code injection and modification.

decade ago, the research community has proposed a large number of variations of the original idea. More recently, CFI implementations have been integrated into production-quality compilers, tools, and operating systems.

Current CFI mechanisms can be compared along two major axes: performance and security. In the scientific literature, performance overhead is usually measured through the SPEC CPU2006 benchmarks. Unfortunately, sometimes only a subset of the benchmarks is used for evaluation. To evaluate security, many authors have used the Average Indirect target Reduction (AIR) [212] metric that counts the overall reduction of targets for any indirect control-flow transfer.

Current evaluation techniques do not adequately distinguish among CFI mechanisms along these axes. Performance measurements are all in the same range, between 0% and 20% across different benchmarks with only slight variations for the same benchmark. Since the benchmarks are evaluated on different machines with different compilers and software versions, these numbers are close to the margin of measurement error. On the security axis, AIR is not a desirable metric for two reasons. First, all CFI mechanisms report similar AIR numbers (a $> 99\%$ reduction of branch targets), which makes AIR unfit to compare individual CFI mechanisms against each other. Second, even a large reduction of targets often leaves enough targets for an attacker to achieve the desired goals [34, 60, 81], making AIR unable to evaluate security of CFI mechanisms on an absolute scale.

We systematize the different CFI mechanisms (where “mechanism” captures both the analysis and enforcement aspects of an implementation) and compare them against metrics for security and performance. By introducing metrics for these areas, our analysis allows the objective comparison of different CFI mechanisms both on an absolute level and relatively against other mechanisms. This in turn allows potential users to assess the trade-offs of individual CFI mechanisms and choose the one that is best suited to their use case. Further, our systematization provides a more meaningful way to classify CFI mechanism than the ill-defined and inconsistently used “coarse” and “fine” grained classification.

To evaluate the security of CFI mechanisms we follow a *comprehensive* approach, classifying them according to a *qualitative* and a *quantitative* analysis. In the qualitative security discussion we compare the strengths of individual solutions on a conceptual level by evaluating the CFI policy of each mechanism along several axes: (i) precision in the forward direction, (ii) precision in the backward direction, (iii) supported control-flow transfer types according to the source programming language, and (iv) reported performance. In the quantitative evaluation, we measure the target sets generated by each CFI mechanism for the SPEC CPU2006 benchmarks. The precision and security guarantees of a CFI mechanism depend on the *precision* of target sets used at runtime, i.e., across all control-flow transfers, how many superfluous targets are reachable through an individual control-flow transfer. We compute these target sets for all available CFI mechanisms and compare the ranked sizes of the sets against each other. This methodology lets us compare the actual sets used for the integrity checks of one mechanism against other mechanisms. In addition, we collect all indirect control-flow targets used for the individual SPEC CPU2006 benchmarks and use these sets as a lower bound on the set of required targets. We use this lower bound to compute how close a mechanism is to an *ideal* CFI mechanism. An ideal CFI mechanism is one where the enforced CFG's edges exactly correspond to the executed branches.

As a second metric, we evaluate the performance impact of open-sourced, compiler-based CFI mechanisms. In their corresponding publications, each mechanism was evaluated on different hardware, different libraries, and different operating systems, using either the full or a partial set of SPEC CPU2006 benchmarks. We cannot port all evaluated CFI mechanisms to the same baseline compiler. Therefore, we measure the overhead of each mechanism relative to the compiler it was integrated into. This apples-to-apples comparison highlights which SPEC CPU2006 benchmarks are most useful when evaluating CFI.

We first give a detailed background of the theory underlying the analysis phase of CFI mechanisms. This allows us to then qualitatively compare the different mechanisms on the precision of their analysis. We then quantify this comparison with a novel metric. This is followed by our performance results for the different implementation. Finally, we highlight best practices and future research directions for the CFI community identified during our evaluation of the different mechanisms, and conclude.

4.3 Foundational Concepts

We first introduce CFI and discuss the two components of most CFI mechanisms: (i) the *analysis* that defines the CFG (which inherently limits the precision that can be achieved) and (ii) the runtime instrumentation that *enforces* the generated CFG. Secondly, we classify and systematize different types of control-flow transfers and how they are used in programming languages. Finally, we briefly discuss the CFG precision achievable with different types of static analysis. For those interested, a more comprehensive overview of static analysis techniques is available in **??**.

4.3.1 Control-Flow Integrity

CFI is a policy that restricts the execution flow of a program at runtime to a predetermined CFG by validating indirect control-flow transfers. On the machine level, indirect control-flow transfers may target any executable address of mapped memory, but in the source language (C, C++, or Objective-C) the targets are restricted to valid language constructs such as functions, methods and switch statement cases. Since the aforementioned languages rely on manual memory management, it is left to the programmer to ensure that non-control data accesses do not interfere with accesses to control data such that programs execute legitimate control flows. Absent any security policy, an attacker can therefore exploit memory corruption to redirect the control-flow to an arbitrary memory location, which is called control-flow hijacking. CFI closes the gap between machine and source code semantics by restricting the allowed control-flow transfers to a smaller set of target locations. This smaller set is determined per indirect control-flow location. Note that languages providing

```
 1  void foo(int a){
 2      return;
 3  }
 4  void bar(int a){
 5      return;
 6  }
 7  void baz(void){
 8      int a = input();
 9      void (*fptr)(int);
10      if(a){
11        fptr = foo;
12        fptr();
13      } else {
14        fptr = bar;
15        fptr();
16      }
17  }
```

Figure 4.1: Simplified example of over approximation in static analysis.

complete memory and type safety generally do not need to be protected by CFI. However, many of these "safe" languages rely on virtual machines and libraries written in C or C++ that will benefit from CFI protection.

Most CFI mechanisms determine the set of valid targets for each indirect control-flow transfer by computing the CFG of the program. The security guarantees of a CFI mechanism depend on the precision of the CFG it constructs. The CFG cannot be perfectly precise for non-trivial programs. Because the CFG is statically determined, there is always some over-approximation due to imprecision of the static analysis. An equivalence class is the set of valid targets for a given indirect control-flow transfer. Throughout the following, we reference Figure 4.1. Assuming an analysis based on function types or a flow-insensitive analysis, both `foo()` and `bar()` end up in the same equivalence class. Thus, at line 12 and line 15 either function can be called. However, from the source code we can tell that at line 12 only `foo()` should be called, and at line 15 only `bar()` should be called. While this specific problem can be addressed with a flow-sensitive analysis, all known static program analysis techniques are subject to some over-approximation (see 2.3).

Once the CFI mechanism has computed an approximate CFG, it has to enforce its security policy. We first note that CFI does not have to enforce constraints for control-flows due to direct branches because their targets are immune to memory corruption thanks to DEP. Instead, it focuses on attacker-corruptible branches such as indirect calls, jumps, and returns. In particular, it must protect control-flow transfers that allow runtime-dependent, targets such as `void (*fptr)(int)` in Figure 4.1. These targets are stored in either a register or a memory location depending on the compiler and the exact source code. The indirection such targets provide allows flexibility as, e.g., the target of a function may depend on a call-back that is passed from another module. Another example of indirect control-flow transfers is return instructions that read the return address from the stack. Without such an indirection, a function would have to explicitly enumerate all possible callers and check to which location to return to based on an explicit comparison.

For indirect call sites, the CFI enforcement component validates target addresses before they are used in an indirect control-flow transfer. This approach detects code pointers (including return addresses) that were modified by an attacker – if the attacker's chosen target is not a member of the statically determined set.

4.3.2 Classification of Control-Flow Transfers

Control-flow transfers can broadly be separated into two categories: (i) *forward* and (ii) *backward*. Forward control-flow transfers are those that move control to a new location inside a program. When a program returns control to a prior location, we call this a backward control-flow[2].

A CPU's instruction-set architecture (ISA) usually offers two forward control-flow transfer instructions: call and jump. Both of these are either direct or indirect, resulting in four different types of forward control-flow:

[2] Note the ambiguity of a backward edge in machine code (i.e., a backward jump to an earlier memory location) which is different from a backward control-flow transfer as used in CFI.

- *direct jump*: is a jump to a constant, statically determined target address. Most local control-flow, such as loops or if-then-else cascaded statements, use direct jumps to manage control.

- *direct call*: is a call to a constant, statically determined target address. Static function calls, for example, use direct call instructions.

- *indirect jump*: is a jump to a computed, i.e., dynamically determined target address. Examples for indirect jumps are switch-case statements using a dispatch table, Procedure Linkage Tables (PLT), as well as the threaded code interpreter dispatch optimization [19, 63, 102].

- *indirect call*: is a call to a computed, i.e., dynamically determined target address. The following three examples are relevant in practice:

  Function pointers are often used to emulate object-oriented method dispatch in classical record data structures, such as C `structs`, or for passing callbacks to other functions.

  vtable dispatch is the preferred way to implement dynamic dispatch to C++ methods. A C++ object keeps a pointer to its *vtable*, a table containing pointers to all virtual methods of its dynamic type. A method call, therefore, requires (i) dereferencing the vtable pointer, (ii) computing table index using the method offset determined by the object's static type, and (iii) an indirect call instruction to the table entry referenced in the previous step. In the presence of multiple inheritance, or multiple dispatch, dynamic dispatch is slightly more complicated.

  Smalltalk-style `send`-method dispatch that requires a dynamic type look-up. Such a dynamic dispatch using a `send`-method in Smalltak, Objective-C, or JavaScript requires walking the class hierarchy (or the prototype chain in JavaScript) and selecting the first method with a matching identifier. This procedure is required for all method calls and therefore impacts performance negatively. Note that, e.g., Objective-C uses a lookup cache to reduce the overhead.

We note that jump instructions can also be either conditional or unconditional.

All common ISAs support backward and forward indirect control-flow transfers. For example, the x86 ISA supports backward control-flow transfers using just one instruction: return, or just `ret`. A return instruction is the symmetric counterpart of a call instruction, and a compiler emits function prologues and epilogues to form such pairs. A call instruction pushes the address of the immediately following instruction onto the native machine stack. A return instruction pops the address off the native machine stack and updates the CPU's instruction pointer to point to this address. Notice that a return instruction is conceptually similar to an indirect jump instruction, since the return address is unknown a priori. Furthermore, compilers are emitting call-return pairs by *convention* that hardware usually does not enforce. By modifying return addresses on the stack, an attacker can "return" to all addresses in a program, the foundation of return-oriented programming [36, 161, 172].

Control-flow transfers can become more complicated in the presence of exceptions. Exception handling complicates control-flows locally, i.e., within a function, for example by moving control from a try-block into a catch-block. Global exception-triggered control-flow manipulation, i.e., interprocedural control-flows, require unwinding stack frames on the current stack until a matching exception handler is found.

Other control-flow related issues that CFI mechanisms should (but not always do) address are: (i) separate compilation, (ii) dynamic linking, and (iii) compiling libraries. These present challenges because the entire CFG may not be known at compile time. This problem can be solved by relying on LTO, or dynamically constructing the combined CFG. Finally, keep in mind that, in general, not all control-flow transfers can be recovered from a binary.

Summing up, our classification scheme of control-flow transfers is as follows:

- **CF.1**: backward control-flow,
- **CF.2**: forward control-flow using direct jumps,
- **CF.3**: forward control-flow using direct calls,

- **CF.4**: forward control-flow using indirect jumps,
- **CF.5**: forward control-flow using indirect calls supporting function pointers,
- **CF.6**: forward control-flow using indirect calls supporting vtables,
- **CF.7**: forward control-flow using indirect calls supporting Smalltalk-style method dispatch,
- **CF.8**: complex control-flow to support exception handling,
- **CF.9**: control-flow supporting language features such as dynamic linking, separate compilation, etc.

According to this classification, the C programming language uses control-flow transfers 1–5, 8 (for setjmp/longjmp) and 9, whereas the C++ programming language allows all control-flow transfers except no. 7.

4.3.3 Classification of Static Analysis Precision

As we saw in Section 4.3.1, the security guarantees of a CFI mechanism ultimately depend on the precision of the CFG that it computes. This precision is in turn determined by the type of static analysis used. The following classification summarizes prior work to determine forward control-flow transfer analysis precision . In order of increasing static analysis precision (SAP), our classifications are:

- **SAP.F.0**: No forward branch validation
- **SAP.F.1a**: ad-hoc algorithms and heuristics
- **SAP.F.1b**: context- and flow-insensitive analysis
- **SAP.F.1c**: labeling equivalence classes
- **SAP.F.2**: class-hierarchy analysis

- **SAP.F.3**: rapid-type analysis
- **SAP.F.4a**: flow-sensitive analysis
- **SAP.F.4b**: context-sensitive analysis
- **SAP.F.5**: context- and flow-sensitive analysis
- **SAP.F.6**: dynamic analysis (optimistic)

The following classification summarizes prior work to determine backward control-flow transfer analysis precision:

- **SAP.B.0**: No backward branch validation
- **SAP.B.1**: Labeling equivalence classes
- **SAP.B.2**: Shadow stack

Note that there is well established and vast prior work in static analysis that goes well beyond the scope of this discussion [138]. The goal of our systematization is merely to summarize the most relevant aspects and use them to shed more light on the precision aspects of CFI.

4.3.4 Nomenclature and Taxonomy

Prior work on CFI usually classifies mechanisms into fine-grained and coarse-grained. Over time, however, these terms have been used to describe different systems with varying granularity and have, therefore, become overloaded and imprecise. In addition, prior work only uses a rough separation into forward and backward control-flow transfers without considering sub types or precision. We hope that the classifications here will allow a more precise and consistent definition of the precision of CFI mechanisms underlying analysis, and will encourage the CFI community to use the most precise techniques available from the static analysis literature.

4.4 Static Analysis for CFI

Object-Oriented Programming Languages A C-like language requires call-string or type context-sensitivity to compute precise results for function pointers. Due to dynamic dispatch, however, a C++-like language should consider more context provided by object sensitivity [116, 125]. Alternatively, prior work describes several algorithms to "devirtualize" call-sites. If a static analysis identifies that only one receiver is possible for a given call-site (i.e., if the points-to set is a singleton) a compiler can sidestep expensive dynamic dispatch via the vtable and generate a direct call to the referenced method. Class-hierarchy analysis (CHA) [62] and rapid-type analysis (RTA) [17] are prominent examples that use domain-specific information about the class hierarchy to optimize virtual method calls. RTA differs from CHA by pruning entries from the class hierarchy from objects that have not been instantiated. As a result, the RTA precision is higher than CHA precision [83]. Grove and Chambers [83] study the topic of call-graph construction and present a partial order of various approaches' precision (Figure 19, pg. 735). With regards to CFI, higher precision in the call-graph of virtual method invocations translates to either (i) more de-virtualized call-sites, which replace an indirect call by a direct call, or (ii) shrinking the points-to sets, which reduce an adversary's attack surface. Note that the former, de-virtualization of a call-site also has the added benefit of removing the call-site from a points-to set and transforming an indirect control-flow transfer to a direct control-flow transfer that need not be validated by the CFI enforcement component.

4.4.1 A Practical Perspective

Points-to analysis over-approximation reduces precision and therefore restricts the optimization potential of programs. The reduced precision also lowers precision for CFI, opening the door for attackers. If, for instance, the over-approximated set of computed targets contains many more "reachable" targets, then an attacker can use those control-flow transfers without violating the CFI policy. Consequently, prior results from studying the precision of static points-to analysis are of key importance to understanding CFI policies' security properties.

Mock et al. have studied dynamic points-to sets and compared them to statically determined points-to sets [128]. More precisely, the study used an instrumentation framework to compute dynamic points-to sets and compared them with three flow- and context-insensitive points-to algorithms. The authors report that static analyses identified 14% of all points-to sets as singletons, whereas dynamic points-to sets were singletons in 97% of all cases. In addition, the study reports that one out of two statically computed singleton points-to sets were optimal in the sense that the dynamic points-to sets were also singletons. The authors describe some caveats and state that flow and context sensitive points-to analyses were not practical in evaluation since they did not scale to practical programs. Subsequent work has, however, established the scalability of such points-to analyses [86–88], and a similar experiment evaluating the precision of computed results is warranted.

Concerning the analysis of devirtualized method calls, prior work reports the following results. By way of manual inspection, Rountev et al. [164] report that 26% of call chains computed by RTA were actually infeasible. Lhotak and Hendren [116] studied the effect of context-sensitivity to improve precision on object-oriented programs. They find that context sensitivity has only a modest effect on call-graph precision, but also report substantial benefits of context sensitivity to resolve virtual calls. In particular, Lhotak and Hendren highlight the utility of object-sensitive analyses for this task. Tip and Palsberg [187] present advanced algorithms, XTA among others, and report that it improves precision over RTA, on average, by 88%.

4.4.2 Backward Control Flows

Figure 4.2 shows two functions, f and g, which call another function h. The return instruction in function h can, therefore, return to either function f or g, depending on which function actually called h at run-time. To select the proper caller, the compiler maintains and uses a stack of

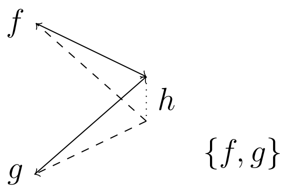


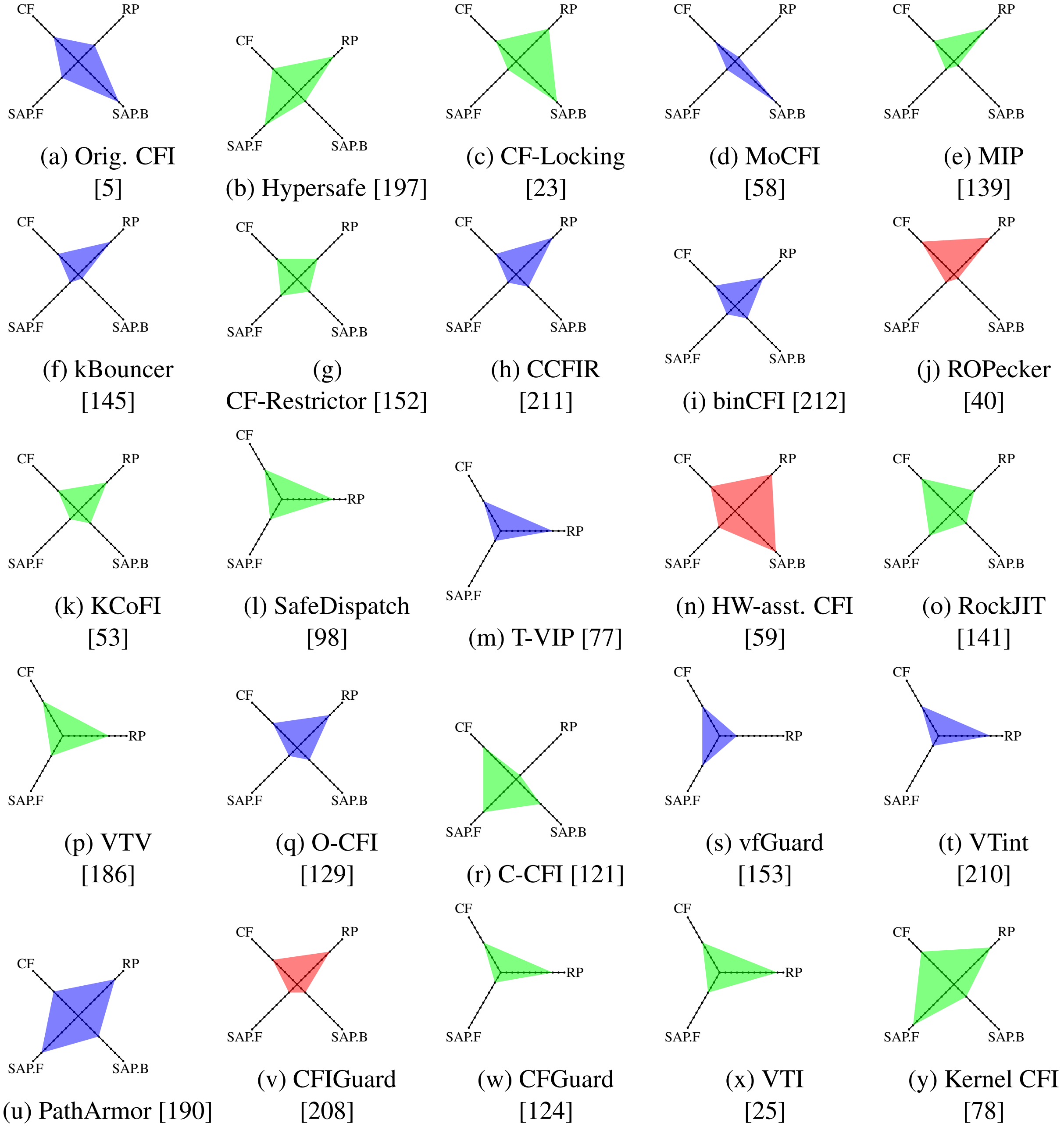
Figure 4.2: Backward control-flow precision. Solid lines correspond to function calls and dashed lines to returns from functions to their call sites. Call-sites are singletons whereas h's return can return to two callers.

activation records, also known as stack frames. Each stack frame contains information about the CPU instruction pointer of the caller as well as bookkeeping information for local variables.

Since there is only one return instruction at the end of a function, even the most precise static analysis can only infer the set of callers for all calls. Computing this set, inevitably, leads to imprecision and all call-sites of a given function must therefore share the same label/ID such that the CFI check succeeds. Presently, the only known alternative to this loss of precision is to maintain a shadow stack and check whether the current return address equals the return address of the most recent call instruction.

4.5 Security

In this section we present a security analysis of existing CFI implementations. Drawing on the foundational knowledge in Section 4.3, we present a qualitative analysis of the theoretical security of different CFI mechanisms based on the policies that they implement. We then give a quantitative evaluation of a selection of CFI implementations. Finally, we survey previous security evaluations and known attacks against CFI.



(a) Orig. CFI [5]

(b) Hypersafe [197]

(c) CF-Locking [23]

(d) MoCFI [58]

(e) MIP [139]

(f) kBouncer [145]

(g) CF-Restrictor [152]

(h) CCFIR [211]

(i) binCFI [212]

(j) ROPecker [40]

(k) KCoFI [53]

(l) SafeDispatch [98]

(m) T-VIP [77]

(n) HW-asst. CFI [59]

(o) RockJIT [141]

(p) VTV [186]

(q) O-CFI [129]

(r) C-CFI [121]

(s) vfGuard [153]

(t) VTint [210]

(u) PathArmor [190]

(v) CFIGuard [208]

(w) CFGuard [124]

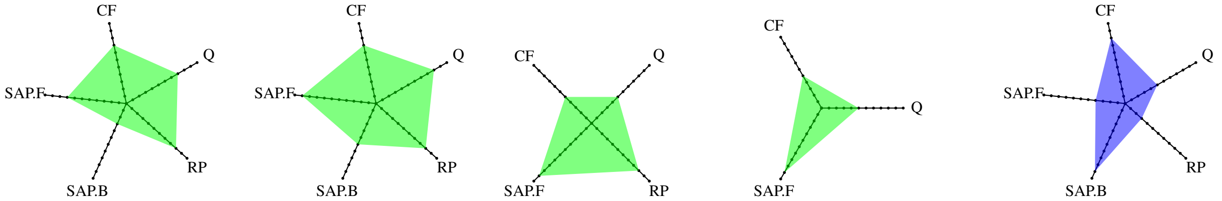
(x) VTI [25]

(y) Kernel CFI [78]

Figure 4.3: CFI implementation comparison: supported control-flows (CF), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B). Backward (SAP.B) is omitted for mechanisms that do not support back edges. Color coding of CFI implementations: binary are blue, source-based are green, others red.

4.5.1 Qualitative Security Guarantees

Our qualitative analysis of prior work and proposed CFI implementations relies on the classifications of the previous section (cf. Section 4.3) to provide a higher resolution view of precision and security.



(a) MCFI [140]

(b) πCFI [143]

(c) IFCC [186]

(d) LLVM-CFI-3.7 (2015)

(e) Lockdown [151]

Figure 4.4: Quantitative comparison: control-flows (CF), quantitative security (Q), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B).

Figure 4.3 summarizes our findings among four dimensions based on the author's reported results and analysis techniques. Figure 4.4 presents our verified results for open source LLVM-based implementations that we have selected. Further, it adds a quantitative argument based on our work in Section 4.5.2.

In Figure 4.3 the axes and values were calculated as follows. Note that (i) the scale of each axis varies based on the number of data points required and (ii) weaker/slower always scores lower and stronger/faster higher. Therefore, the area of the spider plot roughly estimates the security/precision of a given mechanism:

- CF: supported control-flow transfers, assigned based on our classification scheme in Section 4.3.2;
- RP: reported performance numbers. Performance is quantified on a scale of 1-10 by taking the arctangent of reported runtime overhead and normalizing for high granularity near the median overhead. An implementation with no overhead receives a full score of 10, and one with about 35% or greater overhead receives a minimum score of 1.
- SAP.F: static-analysis precision of forward control-flows, assigned based on our classification in Section 4.3.3; and
- SAP.B: static-analysis precision of backward control-flows, assigned based on our classification in Section 4.3.3.

The shown CFI implementations are ordered chronologically by publication year, and the colors indicate whether a CFI implementation works on the binary-level (blue), relies on source-code (green), or uses other mechanisms (red), such as hardware implementations.

Our classification and categorization efforts for reported performance were hindered by methodological variances in benchmarking. Experiments were conducted on different machines, different operating systems, and also different or incomplete benchmark suites. Classifying and categorizing static analysis precision was impeded by the high level, imprecise descriptions of the implemented static analysis by various authors. Both of these impediments, naturally, are sources of imprecision in our evaluation.

Comprehensive protection through CFI requires the validation of both forward and backward branches. This requirement means that the reported performance impact for forward-only approaches (i.e., SafeDispatch, T-VIP, VTV, IFCC, vfGuard, and VTint) is restricted to partial protection. The performance impact for backward control-flows must be considered as well, when comparing these mechanisms to others with full protection.

CFI mechanisms satisfying SAP.B.2, i.e., using a shadow stack to obtain high precision for backward control-flows are: original CFI [5], MoCFI [58], HAFIX [13, 59], and Lockdown [151]. PathArmor emulates a shadow stack through validating the last-branch register (LBR).

Increasing the precision of static analysis techniques that validate whether any given control-flow transfer corresponds to an edge in the CFG decreases the performance of the CFI mechanism. Most implementations choose to combine precise results of static analysis into an equivalence class. Each such equivalence class receives a unique identifier, often referred to as a label, which the CFI enforcement component validates at runtime. By not using a shadow stack, or any other comparable high-precision backward control-flow transfer validation mechanism, even high precision forward control-flow transfer static analysis becomes imprecise due to labeling. The explanation for this loss in precision is straightforward: to validate a control-flow transfer, all callers of a function need to carry the same label. Labeling, consequently, is a substantial source of imprecision (see Section 4.5.2 for more details). The notable exception in this case is πCFI, which uses dynamic information, to activate pre-determined edges, dynamically enabling high-resolution, precise control-flow graph (somewhat analogous to dynamic points-to sets [128]. Borrowing a term from information-flow control [165], πCFI can, however, suffer from *label creep* by accumulating too many labels from the static CFG.

CFI implementations introducing imprecision via labeling are: the original CFI paper [5], control-flow locking [23], CF-restrictor [152], CCFIR [211], MCFI [140], KCoFI [54], and RockJIT [141].

According to the criteria established in analyzing points-to precision, we find that at the time of this writing, πCFI [143] offers the highest precision due to leveraging dynamic points-to information. πCFI's predecessors, RockJIT [141] and MCFI [140], already offered a high precision due to the use of context-sensitivity in the form of types. Ideal PathArmor also scores well when subject to our evaluation: high-precision in both directions, forward and backward, but is hampered by limited hardware resources (LBR size) and restricting protection to the main executable (i.e., trusting libraries). Lockdown [151] offers high precision on the backward edges but derives its equivalence classes from the number of libraries used in an application and is therefore inherently limited in the precision of the forward edges. IFCC [186] offers variable static analysis granularity. On the one hand, IFCC describes a Full mode that uses type information, similar to πCFI and its predecessors. On the other hand, IFCC mentions less precise modes, such as using a single set for all destinations, and separating by function arity. With the exception of Hypersafe [197], all other evaluated CFI implementations with supporting academic publications offer lower precision of varying degrees, at most as precise as SAP.F.3.

4.5.2 Quantitative Security Guarantees

Quantitatively assessing how much security a CFI mechanism provides is challenging as attacks are often program dependent and different implementations might allow different attacks to succeed. So far, the only existing quantitative measure of the security of a CFI implementation is Average Indirect Target Reduction (AIR). Unfortunately, AIR is known to be a weak proxy for security [186]. A more meaningful metric must focus on the number of targets (i.e., number of equivalence classes) available to an attacker. Furthermore, it should recognize that smaller classes are more secure, because they provide less attack surface. Thus, an implementation with a small number of large equivalence classes is more vulnerable than an implementation with a large number of small equivalence classes.

One possible metric is the product of the number of equivalence classes (EC) and the inverse of the size of the largest class (LC), see Equation 4.1. Larger products indicate a more secure mechanism as the product increases with the number of equivalence classes and decreases with the size of the largest class. More equivalence classes means that each class is smaller, and thus provides less attack surface to an adversary. Controlling for the size of the largest class attempts to control for outliers, e.g., one very large and thus vulnerable class and many smaller ones. A more sophisticated version would also consider the usability and functionality of the sets. Usability considers whether or not they are located on an attacker accessible “hot” path, and if so how many times they are used. Functionality evaluates the quality of the sets, whether or not they include “dangerous” functions like mprotect. A large equivalence class that is pointed to by many indirect calls on the hot path poses a higher risk because it is more accessible to the attacker.

$$EC * \frac{1}{LC} = QuantitativeSecurity \tag{4.1}$$

This metric is not perfect, but it allows a meaningful direct comparison of the security and precision of different CFI mechanisms, which AIR does not. The gold standard would be adversarial analysis. However, this currently requires a human to perform the analysis on a per-program basis. This leads to a large number of methodological issues: how many analysts, which programs and inputs, how to combine the results, etc. Such a study is beyond the scope of this work, which instead uses our proposed metric which can be measured programatically.

This section measures the number and sizes of sets to allow a meaningful, direct comparison of the security provided by different implementations. Moreover, we report the dynamically observed number of sets and their sizes. This quantifies the maximum achievable precision from the implementations' CFG analysis, and shows how over-approximate they were for a given execution of the program.

Implementations

We evaluate four compiler-based, open-source CFI mechanisms IFCC, LLVM-CFI, MCFI, and πCFI. For IFCC and MCFI we also evaluated the different analysis techniques available in the implementation. Note that we evaluate two different versions of LLVM-CFI, the first release in LLVM 3.7 and the second, highly modified version in LLVM 3.9. In addition to the compiler-based solutions, we also evaluate Lockdown, which is a binary-based CFI implementation.

MCFI and πCFI already have a built-in reporting mechanism. For the other mechanisms we extend the instrumentation pass and report the number and size of the produced target sets. We then used the implementations to compile, and for πCFI run, the SPEC CPU2006 benchmarks to produce the data we report here. πCFI must be run because it does dynamic target activation. This does tie our results to the ref data set for SPEC CPU2006, because as with any dynamic analysis the results will depend on the input.

IFCC[3] comes with four different CFG analysis techniques: *single*, *arity*, *simplified*, and *full*. *Single* creates only one equivalence class for the entire program, resulting in the weakest possible CFI policy. *Arity* groups functions into equivalence classes based on their number of arguments. *Simplified* improves on this by recognizing three types of arguments: composite, integer, or function pointer. *Full* considers the precise return type and types of each argument. We expect full to yield the largest number of equivalence classes with the smallest sizes, as it performs the most exact distribution of targets.

Both MCFI and πCFI rely on the same underlying static analysis. The authors claim that disabling tail calls is the single most important precision enhancement for their CFG analysis [142]. We measure the impact of this option on our metric. MCFI and πCFI are also unique in that their policy and enforcement mechanisms consider backward edges as well as forward edges. When comparing to other implementations, we only consider forward edges. This ensures direct comparability for the number and size of sets. The results for backward edges are presented as separate entries in the figures.

As of LLVM 3.7, LLVM-CFI could not be directly compared to the other CFI implementations because its policy was strictly more limited. Instead of considering all forward, or all forward and backward edges, LLVM-CFI 3.7 focused on virtual calls and ensures that virtual, and non-virtual calls are performed on objects of the correct dynamic type. As of LLVM 3.9, LLVM-CFI has added support for all indirect calls. Despite these differences, we show the full results for both LLVM-CFI implementations in all tables and graphs.

Lockdown is a CFI implementation that operates on compiled binaries and supports the instrumentation of dynamically loaded code. To protect backward edges, Lockdown enforces a shadow stack. For the forward edge, it instruments libraries at runtime, creating one equivalence class per library. Consequently, the set size numbers are of the greatest interest for Lockdown. Lockdown's precision depends on symbol information, allowing indirect calls anywhere in a particular library if

[3] Note that the IFCC patch was pulled by the authors and will be replaced by LLVM-CFI.

it is stripped. Therefore, we only report the set sizes for non-stripped libraries where Lockdown is more precise.

To collect the data for our lower bound, we wrote an LLVM pass. This pass instruments the program to collect and report the source line for each indirect call, the number of different targets for each indirect call, and the number of times each of those targets was used. This data is collected at runtime. Consequently, it represents only a subset of all possible indirect calls and targets that are required for the sample input to run. As such, we use it to present a lower bound on the number of equivalence sets (i.e. unique indirect call sites) and size of those sets (i.e. the number of different locations called by that site).

4.6 Quantitative security results

We conducted three different quantitative evaluations in line with our proposed metric for evaluating the overall security of a CFI mechanism and our lower bound. For IFCC, LLVM-CFI (3.7 and 3.9), and MCFI it is sufficient to compile the SPEC CPU2006 benchmarks as they do not dynamically change their equivalence classes. πCFI uses dynamic information, so we had to run the SPEC CPU2006 benchmarks. Similarly, Lockdown is a binary CFI implementation that only operates at run time. We highlight the most interesting results in Figure 4.4, see Table 4.1 in Section 4.6 for the full data set.

Figure 4.5 shows the number of equivalence classes for the five CFI implementations that we evaluated, as well as their sub-configurations. As advertised, IFCC *Single* only creates one equivalence class. This IFCC mode offers the least precision of any implementation measured. The other IFCC analysis modes only had a noticeable impact for perlbench and soplex. Indeed, on the sjeng benchmark all four analysis modes produced only one equivalence class.

On forward edges, MCFI and πCFI are more precise than IFCC in all cases except for perlbench

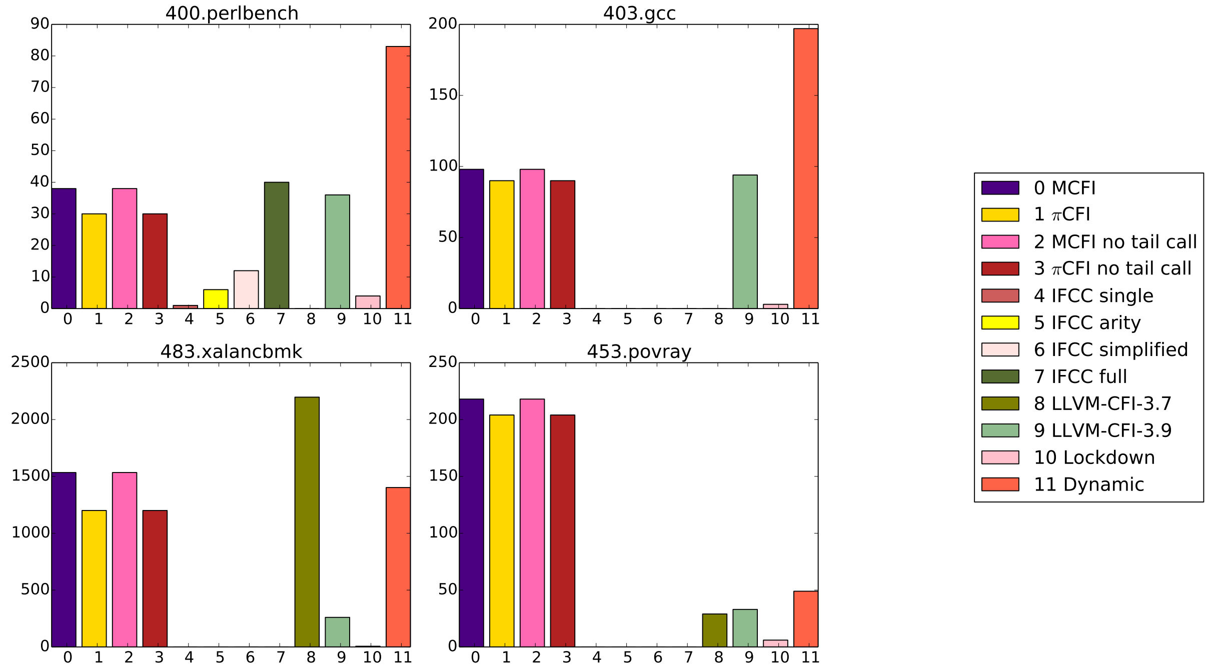


Figure 4.5: Total number of forward-edge equivalence classes when running SPEC CPU2006 (higher is better).

where they are equivalent. LLVM-CFI 3.9 is more precise than IFCC while being less precise than MCFI. MCFI and πCFI are the only implementations to consider backward edges, so no comparison with other mechanisms is possible on backward edge precision. Relative to each other, πCFI's dynamic information decreases the number of equivalence classes available to the attacker by 21.6%. The authors of MCFI and πCFI recommend disabling tail calls to improve CFG precision. This only impacts the number of sets that they create for backward edges, not forward edges, see Section 4.6. As such this compiler flag does not impact most CFI implementations, which rely on a shadow stack for backward edge security.

LLVM-CFI 3.7 creates a number of equivalence classes equal to the number of classes used in the C++ benchmarks. Recall that it only provides support for a subset of indirect control-flow transfer types. However, we present the results in Figure 4.5 and Figure 4.6 to show the relative cost of protecting vtables in C++ relative to protecting all indirect call sites.

We quantify the set sizes for each of the four implementations in Figure 4.6. We show box and

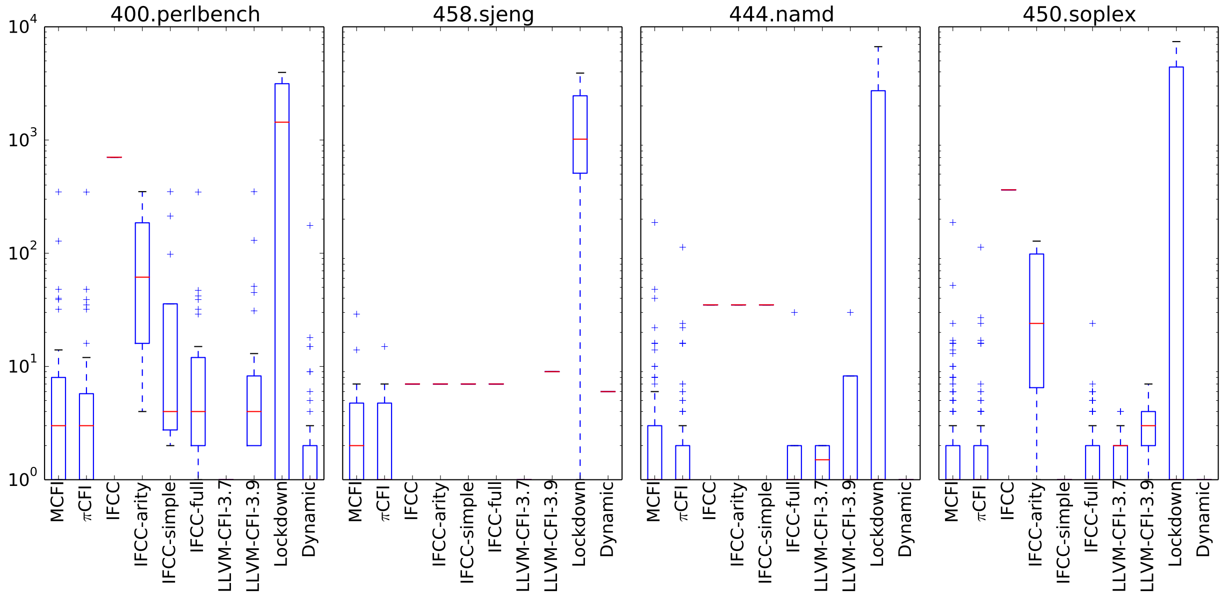


Figure 4.6: Whisker plot of equivalence class sizes for different mechanisms when running SPEC CPU2006. (Smaller is Better)

whisker graphs of the set sizes for each implementation. The red line is the median set size and a smaller median set size indicates more secure mechanisms. The blue box extends from the 25th percentile to the 75th, smaller boxes indicate a tight grouping around the median. An implementation might have a low median, but large boxes indicate that there are still some large equivalence classes for an attacker to target. The top whisker extends from the top of the box for 150% of the size of the box. Data points beyond the whiskers are considered outliers and indicate large sets. This plot format allows an intuitive understanding of the security of the distribution of equivalence class sizes. Lower medians and smaller boxes are better. Any data points above the top of the whisker show very large, outlier equivalence classes that provide a large attack surface for an adversary.

Note that IFCC only creates a single equivalence class for xalancbmk and namd (except for the Full configuration on namd which is more precise). Entries with just a single equivalence class are reported as only a median. IFCC data points allow us to rank the different analysis methods, based on the results for benchmarks where they actually impacted set size: perlbench and soplex. In increasing order of precision (least precise to most precise) they are: *single*, *arity*, *simplified*, and *full*. This does not necessarily mean that the more precise analysis methods are more secure, however. For perlbench the more precise methods have outliers at the same level as the median for the least precise (i.e., *single*) analysis. For soplex the outliers are not as bad, but the *full* outlier is the same size as the median for *arity*. While increasing the precision of the underlying CFG analysis increases the overall security, edge cases can cause the incremental gains to be much smaller than anticipated.

The MCFI forward-edge data points highlight this. The MCFI median is always smaller than the IFCC median. However, for all the benchmarks where both ran, the MCFI outliers are greater than or equal to the largest IFCC set. From a quantitative perspective, we can only confirm that MCFI is at least as secure as IFCC. The effect of the outlying large sets on relative security remains an open question, though it seems likely that they provide opportunities for an attacker.

LLVM-CFI 3.9 presents an interesting compromise. As the full set of whisker plots in Section 4.6 shows, it has fewer outliers. However, it also has, on average, a greater median set size. Given the open question of the importance of the outliers, LLVM-CFI 3.9 could well be more secure in practice.

LLVM-CFI 3.7's sets do not have extreme outliers as only virtual calls are protected. Additionally, Figure 4.6 shows that the equivalence classes that are created have a low variance, as seen by the more compact whisker plots that lack the large number of outliers present for other techniques. As such, LLVM-CFI 3.7 does not suffer from the edge cases that effect more general analyzes.

Lockdown consistently has the largest set sizes, as expected because it only creates one equivalence class per library and the SPEC CPU2006 benchmarks are optimized to reduce the amount of external library calls. These sets are up to an order of magnitude larger than compiler techniques. However, Lockdown isolates faults into libraries as each library has its independent set of targets compared to a single set of targets for other binary-only approaches like CCFIR and binCFI.

The lower bound numbers were measured dynamically, and as such encapsulate a subset of the actual equivalence sets in the static program. Further, each such set is at most the size of the static set. Our lower bound thus provides a proxy for an ideal CFI implementation in that it is perfectly precise for each run. However, all of the IFCC variations report fewer equivalence classes than our dynamic bound.

The whisker plots for our dynamic lower bound in Figure 4.6 show that some of the SPEC CPU2006 benchmarks inherently have outliers in their set sizes. For perlbench, gcc, gobmk, h264ref, omnetpp, and xalancbmk our dynamic lower bound and the static set sizes from the compiler-based implementations all have a significant number of outliers. This provides quantitative backing to the intuition that some code is more amenable to protection by CFI. Evaluating what coding styles and practices make code more or less amenable to CFI is out of scope here, but would make for interesting future work.

Note that for namd and soplex in Figure 4.6 there is no visible data for our dynamic lower bound because all the sets had a single element. This means the median size is one which is too low to be visible. For all other mechanisms no visible data means the mechanism was incompatible with the benchmark.

Table 4.1 contains the number of equivalence sets for each benchmark and every CFI mechanism that we evaluated. Figure 4.7 contains the full set of box and whisker plots. As this data is fundamentally three dimensional, these plots are the best way to display it. As a final note, the holes in this data reflect the fact that the CFI mechanisms that we evaluated cannot run the full set of SPEC CPU2006 benchmarks. This greatly complicates the task of comparatively evaluating them, as there is only a narrow base of programs that all the CFI mechanisms run.

4.6.1 Previous Security Evaluations and Attacks

Evaluating the security of a CFI implementation is challenging because exploits are program dependent and simple metrics do not cover the security of a mechanism. The Average Indirect target

Benchmark	CFI Implementation															
	MCFI	πCFI	MCFI	πCFI	MCFI	πCFI	MCFI	πCFI		IFCC			LLVM-CFI		Lock-	Dynamic
		back edge				forward edge			single	arity	simpl.	full	3.7	3.9	down	
			no tail call				no tail call									
400.perlbench	978	310	1192	429	38	30	38	30	1	6	12	40	0	36	4	83
401.bzip2	484	82	489	86	14	10	14	10	1	2	2	2	0	2	3	12
403.gcc	2219	1260	3282	1836	98	90	98	90	0	0	0	0	0	94	3	197
429.mcf	475	96	475	96	12	8	12	8	0	0	0	0	0	0	3	0
445.gobmk	922	283	1075	230	21	17	21	17	0	0	0	0	0	11	4	0
456.hmmer	663	134	720	147	14	9	14	9	0	0	0	0	0	3	4	9
458.sjeng	540	119	557	125	13	9	13	9	1	1	1	1	0	1	3	1
462.libquantum	495	88	519	102	12	8	12	8	1	1	1	1	0	0	4	0
464.h264ref	773	285	847	327	21	15	21	15	0	0	0	0	0	9	4	59
471.omnetpp	1693	581	1784	624	357	321	357	321	0	0	0	0	114	35	0	224
473.astar	1096	226	1108	237	166	150	166	150	0	0	0	0	1	1	6	1
483.xalancbmk	6161	2381	7162	2869	1534	1200	1534	1200	0	0	0	0	2197	260	6	1402
433.milc	602	169	628	180	13	9	13	9	0	0	0	0	0	1	4	3
444.namd	1080	217	1087	224	166	150	166	150	1	1	1	5	4	4	6	12
447.dealII	2952	817	3468	896	293	258	293	258	0	0	0	0	43	15	0	95
450.soplex	1444	432	1569	479	321	291	321	291	1	7	0	186	41	9	6	157
453.povray	1748	650	1934	743	218	204	218	204	0	0	0	0	29	33	6	49
470.lbm	465	70	470	74	12	8	12	8	0	0	0	0	0	0	4	0
482.sphinx3	633	239	677	257	13	9	13	9	0	0	0	0	0	1	4	2

Table 4.1: Full quantitative security results for number of equivalence classes.

Reduction (AIR) metric [212] captures the average reduction of allowed targets, following the idea that an attack is less likely if fewer targets are available. This metric and variants were then used to measure new CFI implementations, generally reporting high numbers of more than 99%. Such high

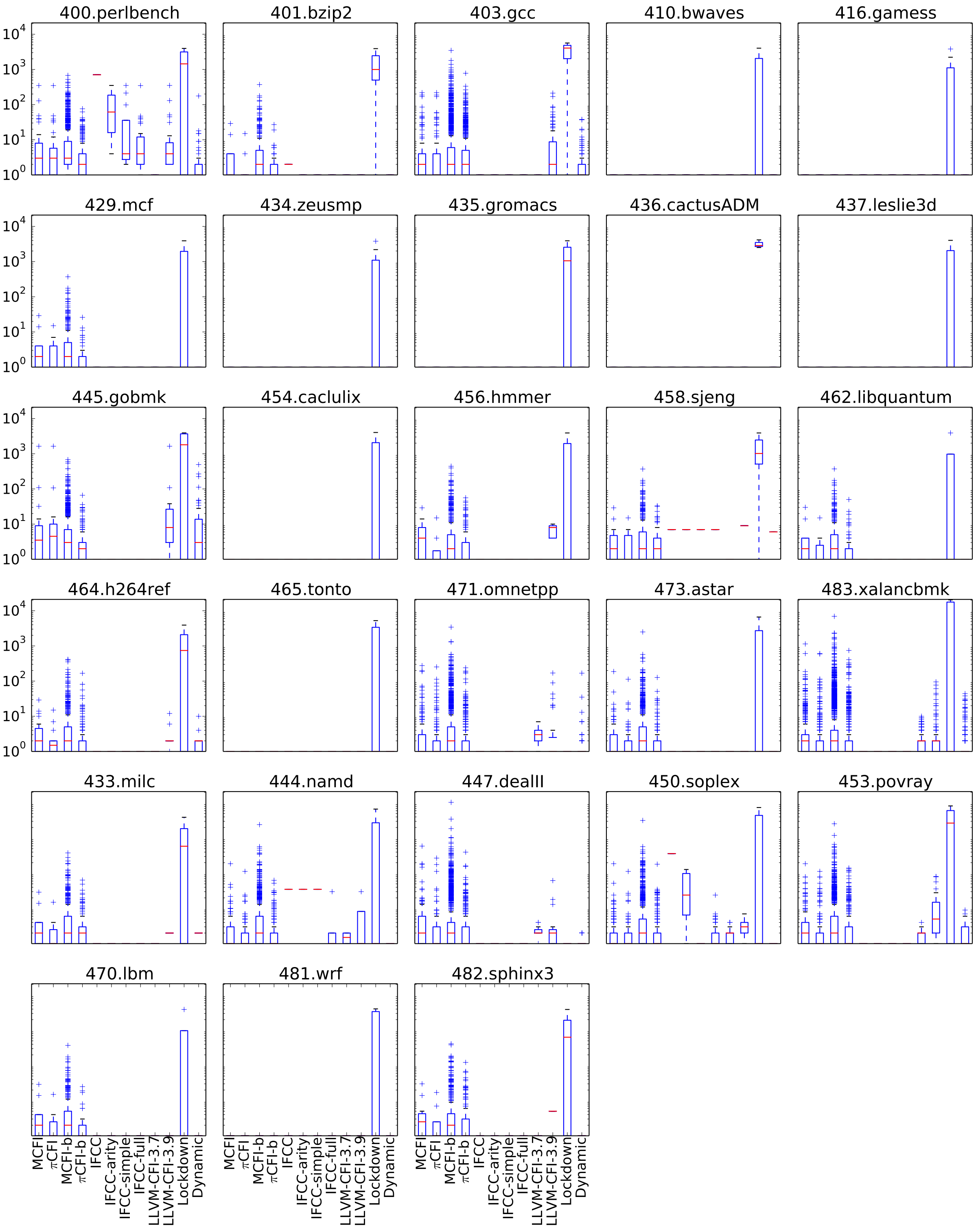


Figure 4.7: Whisker plot of equivalence classes size for all SPEC CPU2006 benchmarks across all implementations (smaller is better).

numbers give the illusion of relatively high security but, e.g., if a binary has 1.8 MB of executable code (the size of the glibc on Ubuntu 14.04), then an AIR value of 99.9% still allows 1,841 targets, likely enough for an arbitrary attack. A similar alternative metric to evaluate CFI effectiveness is the gadget reduction metric [140]. Unfortunately, these simple relative metrics give, at best, an intuition for security and we argue that a more rigorous metric is needed.

A first set of attacks against CFI implementations targeted *coarse-grained* CFI that only had 1-3 equivalence classes [34, 60, 81]. These attacks show that equivalence classes with a large number of targets allow an attacker to execute code and system calls, especially if return instructions are allowed to return to any call site.

Counterfeit Object Oriented Programming (COOP) [167] introduced the idea that whole C++ methods can be used as gadgets to implement Turing-complete computation. Virtual calls in C++ are a specific type of indirect function calls that are dispatched via vtables, which are arrays of function pointers. COOP shows that an attacker can construct counterfeit objects and, by reusing existing vtables, perform arbitrary computations. This attack shows that indirect calls requiring another level-of-indirection (e.g., through a vtable) must have additional checks that consider the types at the language level for the security check as well.

Control Jujutsu [74] extends the existing attacks to so-called fine-grained CFI by leveraging the imprecision of points-to analysis. This work shows that common software engineering practices like modularity (e.g., supporting plugins and refactoring) force points-to analysis to merge several equivalence classes. This imprecision results in target sets that are large enough for arbitrary computation.

Control-Flow Bending [32] goes one step further and shows that attacks against ideal CFI are possible. Ideal CFI assumes that a precise CFG is available that is not achievable in practice, i.e., if any edge would be removed then the program would fail. Even in this configuration attacks are likely possible if no shadow stack is used, and sometimes possible even if a shadow stack is used.

Several attacks target data structures used by CFI mechanisms. StackDefiler [49] leverages the fact that many CFI mechanisms implement the enforcement as a compiler transformation. Due to this high-level implementation and the fact that the optimization infrastructure of the compiler is unaware of the security aspects, an optimization might choose to spill registers that hold sensitive CFI data to the stack where it can be modified by an attack [6]. Any CFI mechanism will rely on some runtime data structures that are sometimes writeable (e.g., when MCFI loads new libraries and merges existing sets). Missing the Point [73] shows that ASLR might not be enough to hide this secret data from an adversary.

4.7 Performance

While the security properties of CFI (or the lack thereof for some mechanisms) have received most scrutiny in the academic literature, performance characteristics play a large part in determining which CFI mechanisms are likely to see adoption and which are not. Szekeres et al. [185] surveyed mitigations against memory corruption and found that mitigations with more than 10% overhead do not tend to see widespread adoption in production environments and that overheads below 5% are desired by industry practitioners.

Comparing the performance characteristics of CFI mechanisms is a non-trivial undertaking. Differences in the underlying hardware, operating system, as well as implementation and benchmarking choices prevents apples-to-apples comparison between the performance overheads reported in the literature. For this reason, we take a two-pronged approach in our performance survey: for a number of publicly available CFI mechanisms, we measure performance directly on the same hardware platform and, whenever possible, on the same operating system, and benchmark suite. Additionally, we tabulate and compare the performance results reported in the literature.

We focus on the aggregate cost of CFI enforcement. For a detailed survey of the performance cost of protecting backward edges from callees to callers we refer to the recent, comprehensive survey by Dang [56].

4.7.1 Measured CFI Performance

Selection Criteria It is infeasible to replicate the reported performance overheads for all major CFI mechanisms. Many implementations are not publicly available or require substantial modification to run on modern versions of Linux or Windows. We therefore focus on recent, publicly available, compiler-based CFI mechanisms.

Several compiler-based CFI mechanisms share a common lineage. LLVM-CFI, for instance, improves upon IFCC, πCFI improves upon MCFI, and VTI is an improved version of SafeDispatch. In those cases, we opted to measure the latest available version and rely on reported performance numbers for older versions.

Method Most authors use the SPEC CPU2006 benchmarks to report the overhead of their CFI mechanism. We follow this trend in our own replication study. All benchmarks were compiled using the `-O2` optimization level. The benchmarking system was a Dell PowerEdge T620 dual processor server having 64GiB of main memory and two Intel Xeon E5-2660 CPUs running at 2.20 GHz. To reduce benchmarking noise, we ran the tests on an otherwise idle system and disabled all dynamic frequency and voltage scaling features. Whenever possible, we benchmark the implementations under 64-bit Ubuntu Linux 14.04.2 LTS. The CFI mechanisms were baselined against the compiler they were implemented on top of: VTV on GCC 4.9, LLVM-CFI on LLVM 3.7 and 3.9, VTI on LLVM 3.7, MCFI on LLVM 3.5, πCFI on LLVM 3.5. Since CFGuard is part of Microsoft Visual C++ Compiler, MSVC, we used MSVC 19 to compile and run SPEC CPU2006 on a pristine 64-bit Windows 10 installation. We report the geometric mean overhead averaged over three benchmark runs using the reference inputs in Table 4.2.

Some of the CFI mechanisms we benchmark required link-time optimization, LTO, which allows the compiler to analyze and optimize across compilation units. LLVM-CFI and VTI both require LTO, so for these mechanisms, we report overheads relative to a baseline SPEC CPU2006 run that also had LTO enabled. The increased optimization scope enabled by LTO can allow the compiler to perform additional optimizations such as de-virtualization to lower the cost of CFI enforcement. On the other hand, LLVM's LTO is less practical than traditional, separate compilation, e.g., when compiling large, complex code bases. To measure the πCFI mechanism, we applied the author's patches[4] for 7 of the SPEC CPU2006 benchmarks to remove coding constructs that are not handled by πCFI's control-flow graph analysis [140]. Likewise, the authors of VTI provided a patch for the xalancbmk benchmark. It updates code that casts an object instance to its sibling class, which can cause a CFI violation. We found these patches for hmmer, povray, and xalancbmk to also be necessary for LLVM-CFI 3.9, which otherwise reports a CFI violation on these benchmarks. VTI was run in interleaved vtable mode which provides the best performance according to its authors [25].

Results Our performance experiments show that recent, compiler-based CFI mechanisms have mean overheads in the low single digit range. Such low overhead is well within the threshold for adoption specified by [185] of 5%. This dispenses with the concern that CFI enforcement is too costly in practice compared to alternative mitigations including those based on randomization [107]. Indeed, mechanisms such as CFGuard, LLVM-CFI, and VTV are implemented in widely-used compilers, offering some level of CFI enforcement to practitioners.

We expect CFI mechanisms that are limited to virtual method calls—VTV, VTI, LLVM-CFI 3.7—to have lower mean overheads than those that also protect indirect function calls such as IFCC. The return protection mechanism used by MCFI should introduce additional overhead, and πCFI's runtime policy ought to result in a further marginal increase in overhead. In practice, our results show that LLVM-CFI 3.7 and VTI are the fastest, followed by CFGuard, πCFI, and VTV. The

[4] The patches are available at: `https://github.com/mcfi/MCFI/tree/master/spec2006`.

Table 4.2: Measured and reported CFI performance overhead (%) on the SPEC CPU2006 benchmarks. The programming language of each benchmark is indicated in parenthesis: C(C), C++(+), Fortran(F). CF in a cell indicates we were unable to build and run the benchmark with CFI enabled. Blank cells mean that no results were reported by the original authors or that we did not attempt to run the benchmark. Cells with bold fonts indicate 10% or more overhead, ntc stands for no tail calls.

Benchmark	Measured Performance							Reported Performance									
	VTV	LLVM-CFI		VTI	CFGuard	πCFI	πCFI	VTV	VTI	πCFI	IFCC	MCFI	PathArmor	Lockdown	C-CFI	ROPecker	binCFI
Version		3.7	3.9														
Options		LTO	LTO	LTO			ntc		LTO		LTO						
400.perlbench(C)			2.4			8.2	5.3			5.0	1.9	5.0	**15.0**	**150.0**		5.0	**12.0**
401.bzip2(C)			-0.7		-0.3	1.2	0.8			1.0		1.0	0.0	8.0	5.0	0.0	-9.0
403.gcc(C)			CF			6.1	**10.5**			4.5		4.5	9.0	**50.0**		3.0	4.5
429.mcf(C)			3.6		0.5	4.0	1.8			4.0		4.0	1.0	2.0	**10.0**	1.0	0.0
445.gobmk(C)			0.2		-0.2	**11.4**	**11.8**			7.5		7.0	0.0	**43.0**	**50.0**	1.0	**15.0**
456.hmmer(C)			0.1		0.7	0.1	-0.1			0.0		0.0	1.0	3.0	**10.0**	0.0	-0.5
458.sjeng(C)			1.6		3.4	8.4	**11.9**			5.0		5.0	0.0	**80.0**	**40.0**	0.0	-2.5
464.h264ref(C)			5.3		5.4	7.9	8.3			6.0		6.0	1.0	**43.0**	**45.0**	1.0	**28.0**
462.libquantum(C)			-6.9			-3.0	-1.0			-0.3		0.0	3.0	5.0	**10.0**	0.0	-0.5
471.omnetpp(+)	5.8	-1.9	CF	CF	3.8	6.7	**18.8**	8.0	1.2	5.0	-1.2	5.0				2.0	**45.0**
473.astar(+)	3.6	-0.3	0.9	1.6	0.1	2.0	2.9	2.4	0.1	4.0	-0.2	3.5		**17.0**	**75.0**	0.0	**14.0**
483.xalancbmk(+)	**24.0**	7.1	7.2	3.7	5.5	**10.3**	**17.6**	**19.2**	1.4	7.0	3.1	7.0		**118.0**	**170.0**	**15.0**	
410.bwaves(F)														1.0			
416.gamess(F)														**11.0**			
433.milc(C)			0.2		2.0	-1.7	1.4			2.0		2.0	4.0	8.0			2.5
434.zeusmp(F)														0.0			
435.gromacs(C,F)														1.0			
436.cactusADM(C,F)														0.0			
437.leslie3d(F)														1.0			
444.namd(+)	-0.1	-0.2	0.1	-0.3	0.1	-0.3	-0.5			-0.5	-0.2	-0.5		3.0			-2.0
447.dealII(+)	0.7	CF	7.9	CF	-0.1	5.3	4.4			4.5	-2.2	4.5					
450.soplex(+)	0.5	0.5	-0.3	-0.6	2.3	-0.7	0.9		-0.7	-4.0	-1.7	-4.0		**12.0**			3.5
453.povray(+)	-0.6	1.5	8.9	2.0	**10.8**	**11.3**	**17.4**			**10.5**	0.2	**10.0**		**90.0**			**37.0**
454.calculix(C,F)														3.0			
459.gemsFDTD(F)														7.0			
465.tonto(F)														**19.0**			
470.lbm(C)			-0.2		4.2	-0.2	-0.5			1.0		1.0	0.0	2.0			-2.5
482.sphinx3(C)			-0.8		-0.1	0.7	2.4			1.5		1.5	3.0	8.0			0.5
Geo Mean	4.6	1.1	4.4	1.3	2.3	4.0	5.8	9.6	0.5	3.2	-0.3	2.9	3.0	**20.0**	**45.0**	2.6	8.5

reported numbers for IFCC when run in *single* mode show that it achieves -0.3%, likely due to cache effects. Although our measured overheads are not directly comparable with those reported by the authors of the seminal CFI paper, we find that researchers have managed to improve the precision while lowering the cost[5] of enforcement as the result of a decade worth of research into CFI enforcement.

The geometric mean overheads do not tell the whole story, however. It is important to look closer at the performance impact on benchmarks that execute a high number of indirect branches. Protecting the xalancbmk, omnetpp, and povray C++ benchmarks with CFI generally incurs substantial overheads. All benchmarked CFI mechanisms had above-average overheads on xalancbmk. LLVM-CFI and VTV, which take virtual call semantics into account, were particularly affected. On the other hand, xalancbmk highlights the merits of the recent virtual table interleaving mechanism of VTI which has a relatively low 3.7% overhead (vs. 1.4% reported) on this challenging benchmark.

Although povray is written in C++, it makes few virtual method calls [210]. However, it performs a large number of indirect calls. The CFI mechanisms which protect indirect calls—πCFI, and CFGuard—all incur high performance overheads on povray. Sjeng and h264ref also include a high number of indirect calls which again result in non-negligible overheads particularly when using πCFI with tail calls disabled to improve CFG precision. The hmmer, namd, and bzip2 benchmarks on the other hand show very little overhead as they do not execute a high number of forward indirect branches of any kind. Therefore these benchmarks are of little value when comparing the performance of various CFI mechanisms.

Overall, our measurements generally match those reported in the literature. The authors of VTV [186] only report overheads for the three SPEC CPU2006 benchmarks that were impacted the most. Our measurements confirm the authors' claim that the runtimes of the other C++ benchmarks are virtually unaffected. The leftmost πCFI column should be compared to the reported column for

[5] Non-CFI related hardware improvements, such as better branch prediction [162], also help to reduce performance overhead.

πCFI. We measured overheads higher than those reported by Niu and Tan [143]. Both gobmk and xalancbmk show markedly higher performance overheads in our experiments; we believe this is in part explained by the fact that Niu and Tan used a newer Intel Xeon processor having an improved branch predictor [162] and higher clock speeds (3.4 vs 2.2 GHz).

We ran πCFI in both normal mode and with tail calls disabled. The geometric mean overhead increased by 1.9% with tail calls disabled. Disabling tail calls in turn increases the number of equivalence classes on each benchmark Figure 4.5. This is a classic example of the performance/security precision trade-off when designing CFI mechanisms. Implementers can choose the most precise policy within their performance target. CFGuard offers the most efficient protection of forward indirect branches whereas πCFI offers higher security at slightly higher cost.

4.7.2 Reported CFI Performance

The right-hand side of Table 4.2 lists reported overheads on SPEC CPU2006 for CFI mechanisms that we do not measure. IFCC is the first CFI mechanism implemented in LLVM which was later replaced by LLVM-CFI. MCFI is the precursor to πCFI. PathArmor is a recent CFI mechanism that uses dynamic binary rewriting and a hardware feature, the Last Branch Record (LBR) [96] register, that traces the 16 most recently executed indirect control-flow transfers. Lockdown is a pure dynamic binary translation approach to CFI that includes precise enforcement of returns using a shadow stack. C-CFI is a compiler-based approach which stores a cryptographically-secure hash-based message authentication code, HMAC, next to each pointer. Checking the HMAC of a pointer before indirect branches avoids a static points-to analysis to generate a CFG. ROPecker is a CFI mechanism that uses a combination of offline analysis, traces recorded by the LBR register, and emulation in an attempt to detect ROP attacks. Finally, the binCFI approach uses static binary rewriting like the original CFI mechanism; binCFI is notable for its ability to protect stripped, position-independent ELF binaries that do not contain relocation information.

The reported overheads match our measurements: xalancbmk and povray impose the highest overheads—up to 15% for ROPecker, which otherwise exhibits low overheads, and 1.7x for C-CFI. The interpreter benchmark, perlbench, executes a high number of indirect branches, which leads to high overheads, particularly for Lockdown, PathArmor, and binCFI.

Looking at CFI mechanisms that do not require re-compilation—PathArmor, Lockdown, ROPecker, and binCFI we see that the mechanisms that only check the contents of the LBR before system calls (PathArmor and ROPecker) report lower mean overheads than approaches that comprehensively instrument indirect branches (Lockdown and binCFI) in existing binaries. More broadly, comparing compiler-based mechanisms with binary-level mechanisms, we see that compiler-based approaches are typically as efficient as the binary-level mechanisms that trace control flows using the LBR although compiler-based mechanisms do not limit protection to a short window of recently executed branches. More comprehensive binary-level mechanisms, Lockdown and binCFI generally have higher overheads than compiler-based equivalents. On the other hand, Lockdown shows the advantage of binary translation: almost any program can be analyzed and protected, independent from the compiler and source code. Also note that Lockdown incurs additional overhead for its shadow stack, while none of the other mechanisms in Table 4.2 have a shadow stack.

Although we cannot directly compare the reported overheads of binCFI with our measured overheads for CFGuard, the mechanisms enforce CFI policies of roughly similar precision (compare Figure 4.3i and Figure 4.3w). CFGuard, however, has a substantially lower performance overhead. This is not surprising given that compilers operate on a high-level program representation that is more amenable to static program analysis and optimization of the CFI instrumentation. On the other hand, compiler-based CFI mechanisms are not strictly faster than binary-level mechanisms, C-CFI has the highest reported overheads by far although it is implemented in the LLVM compiler.

Table 4.3 surveys CFI approaches that do not report overheads using the SPEC CPU2006 benchmarks like the majority of recent CFI mechanisms do. Some authors, use an older version of the SPEC benchmarks [5, 129] whereas others evaluate performance using, e.g., web browsers [98, 211], or web servers [151, 201]. Although it is valuable to quantify overheads of CFI enforcement on more modern and realistic programs, it remains helpful to include the overheads for SPEC CPU2006 benchmarks.

Table 4.3: CFI performance overhead (%) reported from previous publications. A label of ^{C} indicates we computed the geometric mean overhead over the listed benchmarks, otherwise it is the published average.

	Benchmarks	Overhead
ROPGuard [76]	PCMark Vantage, NovaBench, 3DMark06, Peacekeeper, Sunspider, SuperPI 16M	0.5%
SafeDispatch [98]	Octane, Kraken, Sunspider, Balls, linelayout, HTML5	2.0%
CCFIR [211]	SPEC2kINT, SPEC2kFP, SPEC2k6INT	^{C} 2.1%
kBouncer [145]	wmplayer, Internet Explorer, Adobe Reader	^{C} 4.0%
OCFI [129]	SPEC2k	4.7%
CFIMon [201]	httpd, Exim, Wu-ftpd, Memcached	6.1%
Original CFI [5]	SPEC2k	16.0%

4.7.3 Discussion

As Table 4.2 shows, authors working in the area of CFI seem to agree to evaluate their mechanisms using the SPEC CPU2006 benchmarks. There is, however, less agreement on whether to include both the integer and floating point subsets. The authors of Lockdown report the most complete set of benchmark results covering both integer and floating point benchmarks and the authors of binCFI, πCFI, and MCFI include most of the integer benchmarks and a subset of the floating point ones. The authors of VTV and IFCC only report subsets of integer and floating point benchmarks where their solutions introduce non-negligible overheads. Except for CFI mechanisms focused on a particular type of control flows such as virtual method calls, authors should strive to report overheads on the full suite of SPEC CPU2006 benchmarks. In case there is insufficient time to evaluate a CFI mechanism on all benchmarks, we strongly encourage authors to focus on the ones that are challenging to protect with low overheads. These include perlbench, gcc, gobmk, sjeng, omnetpp, povray, and xalancbmk. Additionally, it is desirable to supplement SPEC CPU2006 measurements with measurements for large, frequently targeted applications such as web browsers and web servers.

Although “traditional” CFI mechanisms (e.g., those that check indirect branch targets using a pre-computed CFG) can be implemented most efficiently in a compiler, this does not automatically make such solutions superior to binary-level CFI mechanisms. The advantages of the latter type of approaches include, most prominently, the ability to work directly on stripped binaries when the corresponding source is unavailable. This allows CFI enforcement to be applied independently of the code producer and therefore puts the performance/security trade off in the hands of the end-users or system administrators. Moreover, binary-level solutions naturally operate on the level of entire program modules irrespective of the source language, compiler, and compilation mode that was used to generate the code. Implementers of compiler-based CFI solutions on the other hand must spend additional effort to support separate compilation or require LTO operation which, in some instances, lowers the usability of the CFI mechanism [185].

4.8 Cross-cutting Concerns

This section discusses CFI enforcement mechanisms, presents calls to action identified by our study for the CFI community, and identifies current frontiers in CFI research.

4.8.1 Enforcement Mechanisms

The CFI precursor Program Shepherding [101] was built on top of a dynamic optimization engine, RIO. For CFI like security policies, Program Shepherding effects the way RIO links basic blocks together on indirect calls. They improve the performance overhead of this approach by maintaining traces, or sequences of basic blocks, in which they only have to check that the indirect branch target is the same.

Many CFI papers follow the ID-based scheme presented by Abadi et. al [5]. This scheme assigns a label to each indirect control flow transfer, and to each potential target in the program. Before the transfer, they insert instrumentation to insure that the label of the control flow transfer matches the label of the destination.

Recent work from Google [48, 186] and Microsoft [124] has moved beyond the ID-based schemes to optimized set checks. These rely on aligning metadata such that pointer transformations can be performed quickly before indirect jumps. These transformations guarantee that the indirect jump target is valid.

Hardware-Supported Enforcement Modern processors offer several hardware security-oriented features. Data Execution Prevention is a classical example of how a simple hardware feature can eliminate an entire class of attacks. Many processors also support AES encryption, random number generation, secure enclaves, and array bounds checking via instruction set extensions.

Researchers have explored architectural support for CFI enforcement [13, 44, 59, 184] with the goal of lowering performance overheads. A particular advantage of these solutions is that backward edges can be protected by a fully-isolated shadow stack with an average overhead of just 2% for protection of forward and backward edges. This stands in contrast to the average overheads for software-based shadow stacks which range from 3 to 14% according to Dang [56].

There have also been efforts to repurpose existing hardware mechanisms to implement CFI [40, 145, 190, 208]. kBouncer [145] was first to demonstrate a CFI mechanism using the 16-entry LBR branch trace facility of Intel x86 processors. The key idea in their kBouncer solution is to check the control flow path that led up to a potentially dangerous system call by inspecting the LBR; a heuristic was used to distinguish execution traces induced by ROP chains from legitimate execution traces. ROPecker [40] subsequently extended LBR-based CFI enforcement to also emulate what code would execute past the system call. While these approaches offer negligible overheads and do not require recompilation of existing code, subsequent research showed that carefully crafted ROP attacks can bypass both of these mechanisms [34, 60, 81]. The CFIGuard mechanism [208] uses the LBR feature in conjunction with hardware performance counters to heuristically detect ROP attacks. [201] used the branch trace store, which records control-flow transfers to a buffer in memory, rather than the LBR for CFI enforcement. C-CFI [121] uses the Intel AES-NI instruction set to compute cryptographically-enforced hash-based message authentication codes, HMACs, for pointers stored in attacker-observable memory. By verifying HMACs before pointers are used, C-CFI prevents control-flow hijacking. Mohan et. al. [129] leverage Intel's MPX instruction set extension by re-casting the problem of CFI enforcement as a bounds checking problem over a randomized CFG.

Most recently, Intel announced hardware support for CFI in future x86 processors [146]. Intel Control-flow Enforcement Technology (CET) adds two new instructions, ENDBR32 and ENDBR64, for forward edge protection. Under CET, the target of any indirect jump or indirect call must be a ENDBR instruction. This provides coarse-grained protection where any of the possible indirect targets are allowed at every indirect control-flow transfer. There is only one equivalence class which contains every ENDBR instruction in the program. For backward edges, CET provides a new Shadow Stack Pointer (SSP) register which is exclusively manipulated by new shadow stack instructions. Memory used by the shadow stack resides in virtual memory and is protected with page permissions. In summary, CET provides precise backward edge protection using a shadow stack, but forward edge protection is imprecise because there is only one possible label for destinations.

4.8.2 Open Problems

As seen in Section 4.5.1 most existing CFI implementations use ad hoc, imprecise analysis techniques when constructing their CFG. This unnecessarily weakens these mechanisms, as seen in Section 4.5.2. All future work in CFI should use flow-sensitive and context-sensitive analysis for forward edges, SAP.F.5 from Section 4.3.3. On backward edges, we recommend shadow stacks as they have negligible overhead and are more precise than any possible static analysis. In this same vein, a study of real world applications that identifies coding practices that lead to large equivalence classes would be immensely helpful. This could lead to coding best practices that dramatically increase the security provided by CFI.

Quantifying the incremental security provided by CFI, or any other security mechanism, is an open problem. However, a large adversarial analysis study would provide additional insight into the security provided by CFI. Further, it is likely that CFI could be adapted as a result of such a study to make attacks more difficult.

4.8.3 Research Frontiers

Recent trends in CFI research target improving CFI in directions beyond new analysis or enforcement algorithms. Some approaches have sought to increase CFI protection coverage to include just-in-time code and operating system kernels. Others leverage advances in hardware to improve performance or enable new enforcement strategies. We discuss these research directions in the CFI landscape which cross-cut the traditional categories of performance and security.

Protecting Operating System Kernels. In monolithic kernels, all kernel software is running at the same privilege levels and any memory corruption can be fatal for security. A kernel is vastly different from a user-space application as it is directly exposed to the underlying hardware and an attacker in that space has access to privileged instructions that may change interrupts, page table structures, page table permissions, or privileged data structures. KCoFI [54] introduces a first CFI policy for commodity operating systems and considers these specific problems. The CFI mechanism is fairly coarse-grained: any indirect function call may target any valid functions and returns may target any call site (instead of executable bytes). Xinyang Ge et al. [78] introduce a precise CFI policy inference mechanism by leveraging common function pointer usage patterns in kernel code (SAP.F.4b on the forward edge and SAP.B.1 on the backward edge).

Protecting Just-in-time Compiled Code. Like other defenses, it is important that CFI is deployed comprehensively since adversaries only have to find a single unprotected indirect branch to compromise the entire process. Some applications contain just-in-time, JIT, compilers that dynamically emit machine code for managed languages such as Java and JavaScript. Niu and Tan [141] presented RockJIT, a CFI mechanism that specifically targets the additional attack surface exposed by JIT compilers. RockJIT faces two challenges unique to dynamically-generated code: (i) the code heap used by JIT compilers is usually simultaneously writable and executable to allow important optimizations such as inline caching [91] and on-stack replacement, (ii) computing the control-flow graphs for dynamic languages during execution without imposing substantial performance overheads. RockJIT solves the first challenge by replacing the original heap with a shadow code heap which is readable and writable but not executable and by introducing a sandboxed code heap which is readable and executable, but not writable. To avoid increased memory consumption, RockJIT maps the sandboxed code heap and the shadow heap to the same physical memory pages with different permissions. RockJIT addresses the second challenge by both (i) modifying the JIT compiler to emit meta-data about indirect branches in the generated code and (ii) enforcing a coarse-grained CFI policy on JITed code which avoids the need for static analysis. The authors argue that a less precise CFI policy for JITed code is acceptable as long as both (i) the host application is protected by a more precise policy and (ii) JIT-compiled code prevents adversaries from making system calls. In the Edge browser, Microsoft has updated the JIT compilers for JavaScript and Flash to instrument generated calls and to inform CFGuard of new control-flow targets through calls to `SetProcessValidCallTargets` [75, 127, 199].

Protecting Interpreters. Control-flow integrity for interpreters faces similar challenges as just-in-time compilers. Interpreters are widely deployed, e.g., two major web browsers, Internet Explorer and Safari, rely on mixed-mode execution models that interpret code until it becomes “hot” enough for just-in-time compilation [16], and some Desktop software, too, is interpreted, e.g., Dropbox’s client is implemented in Python. We have already described the “worst-case” interpreters pose to CFI from a security perspective: even if the interpreter’s code is protected by CFI, its actual functionality is determined by a program in data memory. This separation has two important implications: (i) static analysis for an interpreter dispatch routine will result in an over-approximation, and (ii) it enables non-control data attacks through manipulating program source code in writeable data memory prior to JIT compilation.

Interpreters are inherently dynamic, which on the one hand means, CFI for interpreters could rely on precise dynamic points-to information, but on the other hand also indicates problems to build a complete control-flow graph for such programs. Dynamically executing strings as code (`eval`) further complicates this. Any CFI mechanism for interpreters needs to address this challenge.

Protecting Method Dispatch in Object-Oriented Languages. In C/C++ method calls use vtables, which contain addresses to methods, to dynamically bind methods according to the dynamic type of an object. This mechanism is, however, not the only possible way to implement dynamic binding. Predating C++, for example, is Smalltalk-style method dispatch, which influenced the method dispatch mechanisms in other languages, such as Objective-C and JavaScript. In Smalltalk, all method calls are resolved using a dedicated function called `send`. This `send` function takes two parameters: (i) the object (also called the receiver of the method call), and (ii) the method name. Using these parameters, the `send` method determines, at call-time, which method to actually invoke. In general, the determination of which methods are eligible call targets, and which methods cannot be invoked for certain objects and classes cannot be computed statically. Moreover, since objects and classes are both data, manipulation of data to hijack control-flow suffices to influence the method dispatch for malicious intent. While Pewny and Holz [152] propose a mechanism for Objective-C send-like dispatch, the generalisation to Smalltalk-style dispatch remains unsolved.

4.9 Conclusions

Control-flow integrity substantially raises the bar against attacks that exploit memory corruption vulnerabilities to execute arbitrary code. In the decade since its inception, researchers have made major advances and explored a great number of materially different mechanisms and implementation choices. Comparing and evaluating these mechanisms is non-trivial and most authors only provide ad-hoc security and performance evaluations. A prerequisite to any systematic evaluation is a set of well-defined metrics. We have proposed metrics to qualitatively (based on the underlying analysis) and quantitatively (based on a practical evaluation) assess the security benefits of a representative sample of CFI mechanisms. Additionally, we have evaluated the performance trade-offs and have surveyed cross-cutting concerns and their impacts on the applicability of CFI.

Our systematization serves as an entry point and guide to the now voluminous and diverse literature on control-flow integrity. Most importantly, we capture the current state of the art in terms of precision and performance. We report large variations in the forward and backward edge precision for the evaluated mechanisms with corresponding performance overhead: higher precision results in (slightly) higher performance overhead.

We hope that our unified nomenclature will gradually displace the ill-defined qualitative distinction between “fine-grained” and “coarse-grained” labels that authors apply inconsistently across publications. Our metrics provide the necessary guidance and data to compare CFI implementations in a more nuanced way. This helps software developers and compiler writers gain appreciation for the performance/security trade-off between different CFI mechanisms. For the security community, this work provides a map of what has been done, and highlights fertile grounds for future research. Beyond metrics, our unified nomenclature allows clear distinctions of mechanisms. These metrics, if adopted, are useful to evaluate and describe future improvements to CFI.

Chapter 5

Hardware Assisted Randomization of Data

5.1 Abstract

Data-oriented attacks are gaining traction thanks to advances in code-centric mitigation techniques for memory corruption vulnerabilities. Previous work on mitigating data-oriented attacks includes Data Space Randomization (DSR). DSR classifies program variables into a set of equivalence classes, and encrypts variables with a key randomly chosen for each equivalence class. This thwarts memory corruption attacks that introduce illegitimate data flows. However, existing implementations of DSR trade precision for better run-time performance, which leaves attackers sufficient leeway to mount attacks. We show that high precision and good run-time performance are not mutually exclusive. We present HARD, a precise and efficient hardware-assisted implementation of DSR. HARD distinguishes a larger number of equivalence classes, and incurs lower run-time overhead than software-only DSR. Our implementation achieves run-time overheads of just 6.61% on average, while the software version with the same protection costs 40.96%.

5.2 Introduction

Memory corruption exploits remain an important attack vector in practice. Attempts to eliminate this class of vulnerabilities are being undertaken from many angles including i) migration to type safe languages, ii) static and dynamic program analysis, and iii) retrofitting unsafe code with memory safety mechanisms. Automatic exploit mitigations have also been highly effective at driving up the cost of exploitation, and are transparent to developers and end-users. They also avoid the substantial overheads associated with full memory safety enforcement [133–135]. Mitigation techniques such as Address Space Layout Randomization (ASLR), Data Execution Prevention, and Control Flow Integrity (CFI) are widely deployed in modern systems. These techniques increase the difficulty of performing arbitrary code execution attacks, which has encouraged attackers to explore alternatives such as data-oriented attacks [38, 94, 95]. These attacks corrupt program's data flow without diverting its control flow.

Data Space Randomization (DSR) is a promising defense that mitigates data-oriented attacks [22, 29]. DSR thwarts unintended data flows while leaving all legitimate data flows unaffected. To do so, DSR encrypts variables that are stored in the program's memory, and it uses different random keys to encrypt unrelated variables. Generating these keys with sufficient entropy makes the results of load and store operations that violate the program's intended data flow unpredictable, and thus hinders reliable construction of data-oriented attacks.

Prior work on DSR makes several trade-offs that favor run-time performance over security. First, existing versions of DSR do not encrypt variables that cannot be used as the base of an overflow attack. This leaves programs unprotected against temporal memory exploits such as use-after-free or uninitialized reads. Second, prior versions often use weak encryption keys to avoid the cost of handling unaligned memory accesses. Lastly, existing implementations rely on imprecise program analyses, which leads them to incorrectly classify many variables as related. As a result, these unrelated variables are encrypted with the same keys. Many unintended data flows are therefore still possible, which gives attackers some leeway to construct exploits.

This motivated our work on Hardware-Assisted Randomization of Data (HARD), a hardware-assisted implementation of more precise DSR. HARD offers greater security than prior approaches by distinguishing more unrelated variables. To do this, HARD uses a context-sensitive points-to analysis and generates encryption operations that use calling context-specific keys. HARD also encrypts *all* of the program data, and consistently uses strong 64-bit encryption keys. Thus, unlike existing schemes, HARD does not compromise its security guarantees for better run-time performance. Furthermore, HARD incurs less overhead than prior work thanks to its hardware extensions: specialized instructions to access encrypted data and efficient caches to manage encryption keys. These extensions also shield our solution against information leakage attacks because the keys are managed by the hardware and cannot be accessed from user-space.

5.3 Background

Our goal is to thwart attacks that violate the intended data flow of a program. Example 5.1 illustrates two such violations: a *use-after-free* and an *uninitialized read*. Both types of unintended data flows are highly relevant in practice. Use-after-free is commonly exploited to attack high-profile targets such as web browsers and operating system kernels [157], and well-known Heartbleed bug was, at its core, an uninitialized read vulnerability [71].

At lines `(a-1)` and `(a-2)` in the example, the program allocates and initializes a list, `X`, as depicted in Figure 5.1-(a). At line `(b-1)`, the program frees the second element of list `X`, so the `Next` member of the first element becomes a dangling pointer. The program then allocates a new list, `Y`, at line `(b-2)`. The program now reads the contents of list `Y` without initialization at line `(b-3)`. Due to the deterministic nature of common memory allocators such as *dlmalloc* [112], the two lists will likely be laid out in the memory as shown in Figure 5.1-(b). Thus, the data read at line

```
struct list { struct list *Next; int Data; };
list *makeList(int Num) {
  list *New = new list;
  New->Next = Num ? makeList(Num-1) : 0;
  return New;
}
void fillList(list* L, int base) {
  if(L->Next) fillList(L->Next, base+1);
  L->Data = base;
}
void dumpList(list* L) {
  for (list* T = L; T->Next; T = T->Next)
    printf("%d\n", T->Data);
}
int main(int argc, char** argv) {
  list *X = makeList(3);    // (a-1)
  fillList(X,10);           // (a-2)
  free(X->Next);            // (b-1)
  list *Y = makeList(2);    // (b-2)
  dumpList(Y);              // (b-3)
  fillList(Y,20);           // (c-1)
  dumpList(X);              // (c-2)
  return 0;
}
```

Example 5.1: A synthesized program illustrating use-after-free and uninitialized read vulnerabilities.

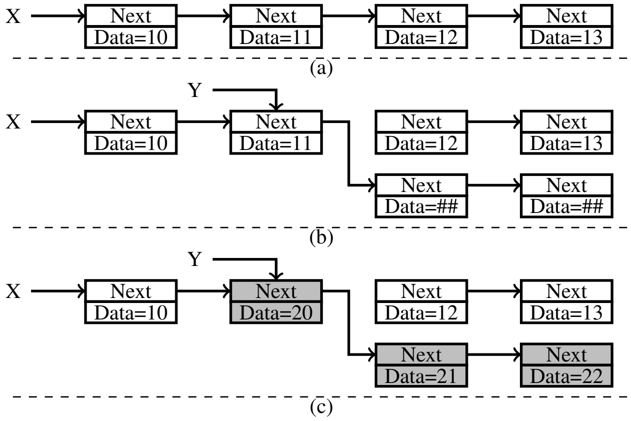


Figure 5.1: The diagram shows the lists generated in Example 5.1. (a) shows list X after initialization at line (a-2). (b) shows the most likely layouts of lists X and Y at line (b-3). (c) shows the most likely layouts of the lists at line (c-2).

(b-3) will likely include the recently free'd element of list X.

The rest of the example demonstrates the use-after-free vulnerability. The program attempts to print the contents of list `X`, whose second element was freed at line `(b-1)`. A deterministic memory allocator may allocate the list `X` as shown in Figure 5.1-(c), and the dumped list includes elements of list `Y`.

5.3.1 Mitigation with DSR

DSR mitigates such unintended data flows by randomizing the representation of program data in memory. DSR relies on alias analysis to compute the points-to relations between pointers and the storage locations they can reference. Two pointers are considered aliases if they can reference the same storage location. Similarly, a pointer p may alias named object o, if p can point to o. Based on the alias analysis, DSR partitions storage locations into *equivalence classes* so that all storage locations belong to an equivalence class. Any two storage locations that may alias each other belong to the same equivalence class.

DSR encrypts storage locations belonging to different equivalence classes with distinct encryption keys. Locations belonging to the same equivalence class, however, must be encrypted with the same key. In the previous example, an ideal implementation of DSR would see that lists `X` and `Y` are disjoint, and would encrypt them with different keys. An attacker that does not know the keys cannot extract the true contents of the illegally read list element.

Unfortunately, existing implementations of DSR cannot prevent the exploits in this example [22, 29]. They *do* consider lists `X` and `Y` related because of the imprecise *(context-insensitive)* alias analysis which does not consider the functions' calling contexts. In the example, both `X` (at line `(a-2)`) and `Y` (at line `(c-1)`) are passed as an argument to `fillList`, and the context-insensitive alias analysis will report that the formal argument `L` of `fillList` may alias both `X` and `Y`. Variables `X` and `Y` will therefore be assigned to the same equivalence class.

We avoid this loss of precision by using a *context-sensitive* alias analysis. If we analyze our example program with a context-sensitive alias analysis, we obtain two sets of aliasing relations: one for the calling context at line `(a-2)` where `fillList`'s formal argument `L` aliases `X`, and one for the calling context at line `(c-1)` where `L` aliases `Y`. By taking the calling context into account, we avoid having to treat `X` and `Y` as aliases and can therefore place them in different equivalence classes.

Leveraging the greater precision of context-sensitive alias analyses is challenging since the DSR instrumentation code must then take the calling context into account to determine which encryption key should be used. We discuss this challenge at length in Section 5.5, and present a novel DSR scheme that supports different contexts via dynamic key binding.

5.4 Threat Model

We assume the following threat model, which is realistic and consistent with related work in this area [95, 182]:

- The victim program contains a memory corruption vulnerability that lets adversaries read and write arbitrary locations as long as such accesses are permitted by the MMU.
- The victim program is protected against code injection by enforcing the $W \oplus X$ policy such that no executable code is writable.
- The victim program runs in user mode and that the host system's software running in supervisor mode has not been compromised.
- We do not consider side-channel attacks, flaws in the hardware, or adversaries that have physical access to the system hosting the victim program.

Note that we do not require that ASLR is enabled, and we do not include any assumptions about it in our threat model. The approach is fully compatible with ASLR, and additional randomness will increase security, for example by making known plaintext attacks harder (see Section 5.10).

5.5 DSR Design

We begin this section by providing a conceptual overview of our design, and then discuss several key components in detail. Our design can either be realized in pure software, similar to prior implementations of DSR, or supported by the hardware extensions we present in Section 5.6.

Our scheme transforms input programs at the compiler intermediate representation (IR) level. The first step is a context-sensitive alias analysis that categorizes the program's memory locations into equivalence classes based on the points-to sets computed by this analysis. We then assign two types of keys to the memory access instructions in the program, according to the equivalence classes they access. We assign a *static key* to instructions that always accesses the same equivalence class, regardless of its calling context, and a *dynamic key* to the others, which may access multiple equivalent classes depending on the calling context. The static keys are directly embedded to the data section of the program so that each instruction can fetch its key, while dynamic keys are passed to a callee through the *context frames*, which the caller should construct. Our scheme transforms 1) function call sites to construct context frames, 2) instructions that use static keys to fetch their keys from the data section, 3) instructions that use dynamic keys to fetch their keys from the context frame, 4) all store instructions to encrypt the data, and 5) all load instructions to decrypt the data.

5.5.1 Enabling Context Sensitivity

One of our primary goals is to support dynamic key assignment for memory instructions that may access multiple equivalence classes depending on their calling contexts. We determine the set of equivalence classes that can be accessed through dynamic keys as follows. For each function in the program, we identify the set of equivalence classes reachable from the function's pointer arguments or pointer return value. From that set, we remove any equivalence classes which contain global variables. If an instruction accesses an equivalence class that contains global variables, then that instruction always accesses that same class, regardless of which context the function is called from. Thus, such an equivalence class can safely be removed from the set. The remaining set of equivalence classes are the *dynamic classes* in that function. Other classes that are used in the function, but that are not in the set (i.e., the classes that were removed because they contain globals, and the classes that are not reachable from the pointer arguments or pointer return value), are considered *static classes*. During instrumentation, we assign dynamic keys to memory access instructions that target dynamic classes, and static keys to those that target static classes.

Managing Context Frames

We store dynamic keys in *context frames*. For each function that contains instructions with dynamic keys, we first instrument all of the function's callers to create the necessary context frame and to populate the frame with the keys for the actual callee arguments. We then instrument the callee so that instructions accessing dynamic classes read the keys from the context frame.

Handling Indirect Calls

Instrumenting indirect call sites complicates this process because if care is not taken, different target functions could require different sets of dynamic keys, even if the target functions have the same signature. To correctly instrument indirect call sites we constrain all functions that may be called from the same call site to have the same dynamic classes.

Static Equivalence Classes

Every instruction that accesses a static class will always access *that* static class, regardless of the calling context. Thus, we can safely assign static keys to instructions that access static classes.

Equivalence classes that contain global variables are always *static classes*. To understand why this is

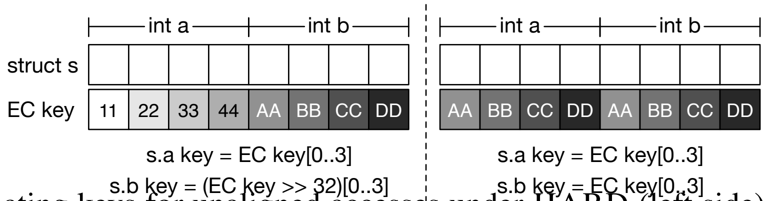


Figure 5.2: Calculating keys for unaligned accesses under HARD (left side) and prior work by Cadar et al. (right side). “EC key" is the key for the equivalence class.

always true, consider how a flow-insensitive alias analysis constructs equivalence classes. An alias analysis evaluates all of the instructions in the program and incorporates any aliasing relationship introduced by an instruction into the points-to sets. When a flow-insensitive alias analysis such as ours evaluates a statement such as:

```
void* a = condition ? &global : &function_argument;
```

it will consider pointer `a` an alias for both `global` and `function_argument`, which will therefore be placed into the same equivalence class. This equivalence class will now be a static class, because, no matter which context this function is called from, any instruction that accesses this static class can now potentially access the memory storage location occupied by `global`.

5.5.2 Memory Encryption

We instrument memory access operations so that the values are `xor`-encrypted before they are stored to and after they are loaded from memory. The encryption/decryption instructions we add use the unique randomly-generated 8-byte key we assign to their respective target equivalence classes.

To use 8-byte keys consistently for all equivalence classes, we must carefully handle memory accesses which are not 8-byte aligned. For example, consider an equivalence class containing a structure with two fields, as shown in Figure 5.2. When accessing field `s.b`, we should shift the key to mask the field’s data with the correct part of the key (left side of the figure). Cadar et al.’s DSR implementation assigns weaker, repeating keys (right side of Figure 5.2) to avoid costly shift operations [29]. We use the hardware support to efficiently handle shift operations.

Our design encrypts all possible equivalence classes. To reduce the run-time overhead, prior DSR systems did not protect equivalence classes that are “safe”. An equivalence class is considered safe if a static analysis can show that none of the accesses to that equivalence class can read or write outside the bounds of the target object. This weakens their protection against temporal memory errors such as use-after-free and uninitialized read. Our hardware extension enables higher level of protection with reasonable overhead.

5.5.3 Support for External Code and Data

We designed our scheme to allow interaction with external code. To do so, we must ensure that any encrypted data is decrypted before it is accessed by the external code, and re-encrypted afterwards. Otherwise, we cannot encrypt any equivalence classes that include the data that is passed to the external functions. To maximize the amount of memory we encrypt, we use wrapper functions around calls to known external functions, as was done in prior DSR implementations. However, we cannot encrypt accesses to external global variables because they could be accessed by external code at any point.

We must also handle function pointers that may escape to external code. An escaping function pointer could be called by the external code without the proper keys in the context frame. Therefore, calls through this pointer must not require dynamic keys. However, it is still possible to pass dynamic keys to direct calls to the same function. To handle this, we maintain two copies of the affected functions—one that accepts dynamic keys and one that does not encrypt accesses to the equivalence classes of the arguments. Note that an attacker may seek to use the version that does not expect encrypted arguments. However, in order to redirect control flow to such a function, she will need to overwrite a code pointer. This memory access will be encrypted, so the attacker will already have to bypass DSR to perform such an overwrite.

5.6 Hardware Design

We designed an extension of the RISC-V architecture to accelerate our DSR scheme's encryption operations and to protect the encryption keys from information leakage attacks. The primary goal of our hardware design is to achieve low overhead. To accomplish this we use a sophisticated hardware design to accelerate the encryption operations used by DSR. If both the encryption key and the data are in the cache, our hardware implementation is able to perform a load or store, key fetch, and XOR within a single instruction without any additional latency or pipeline stalls compared to a normal load or store instruction.

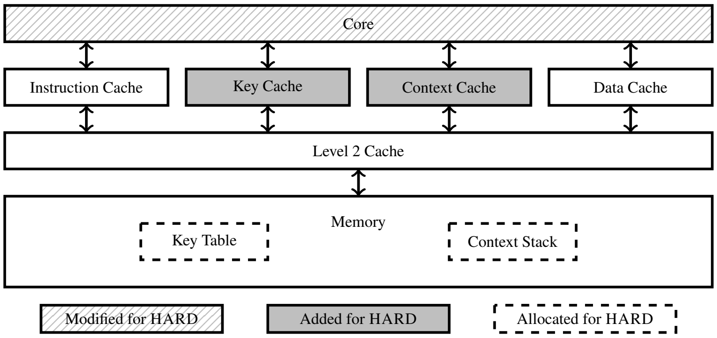


Figure 5.3: Hardware overview for a HARD-enabled system.

Overview. HARD adds or modifies several hardware components, as shown in Figure 5.3. When executing a HARD'ened program, the processor uses two reserved memory regions, the *Context Stack* and the *Key Table*, to store and manage the encryption keys used by the program. The processor accesses these regions directly using their physical addresses and the regions are not mapped into the virtual address space of the protected program. This design ensures that the encryption keys cannot be leaked, as the MMU forbids accesses to unmapped memory. The Key Table stores all encryption keys used in the program. To support dynamically assigned keys, programs must create context frames on the context stack and copy keys from the Key Table to the context frames.

Both the Context Stack and the Key Table have a corresponding cache: the *Context Cache* and the *Key Cache* respectively. These caches are internal to the processor and cannot be read by an attacker. Keys are always loaded through the corresponding caches, and if a key is not present in the cache, the processor will transparently fetch it from the corresponding memory region. To pair the caches with their corresponding memory regions, each cache has a base register containing the physical memory address of the associated memory region.

5.6.1 Hardware Initialization

The OS kernel is responsible for the initialization of the aforementioned memory regions and caches. When the OS loads a HARD'ened program, the kernel allocates the Key Table and initializes it with randomly generated encryption keys. The kernel then sets the base address register of the Key Cache and activates the cache using a control register. Finally, the kernel allocates the Context Stack and sets the base address register of the Context Cache.

5.6.2 New Instructions

HARD adds two sets of instructions. One set of instructions is used to load data from or store data to encrypted memory. The other set of instructions is used to manage the Context Cache and Context Stack.

Memory Access Instructions. The RISC-V instruction set architecture, which we extend, contains nine load instructions and six store instructions. For each of these, HARD adds a specialized version that decrypts data when loading or encrypts data when storing. The specialized instructions use the same mnemonic as the original instructions, but have a `um` suffix (for loads) or `m` suffix (for stores). The double-word load/store instructions, for example, look as follows:

- `ldum rd, id(rb)`: load a double word from the virtual address stored in register `rb`, decrypt the data with the key at index `id` in the Key Table/Context Stack, and write the decrypted data to register `rd`.
- `sdm rd, id(rb)`: encrypt the data in register `rd` with the key at index `id` in the Key

Table/Context Stack and store the encrypted data to the virtual address in register `rb` as a double word.

The type of encryption key is encoded in the Most Significant Bit (MSB) of the index id. If the MSB is set to 0, the remainder of the index id is interpreted as an index into the Key Table, and the instruction therefore has a statically assigned key. We refer to these index IDs as *static IDs*. If the MSB is set to 1, the remainder of the index id is interpreted as an index into the current context frame on the Context Stack, and the instruction therefore has a dynamically assigned key. We refer to these index IDs as *dynamic IDs*.

Context Stack Management Instructions. The second group of instructions are used to manage the Context Stack. Before a new context is entered, the program should prepare a new context frame on the Context Stack and copy the keys used within the corresponding context into that frame. This newly prepared context frame must then be activated before entering the corresponding context. HARD offers four instructions to prepare, activate, and deactivate context frames.

- `mksc dest_id, src_id`: move the key at index `src_id` in the Key Table to slot `dest_id` in the context frame under preparation.
- `mkcc dest_id, src_id`: move a key from slot `src_id` of the currently activated context frame to slot `dest_id` of the context frame under preparation.
- `drpush cur_len`: deactivate the active context frame and activate the context frame under preparation. `cur_len` is the number of slots in the current frame.
- `drpop`: deactivate the active context frame and activate the previous context frame.

The `mksc`, `mkcc`, and `drpush` instructions should be used just before calling a function to provide the matching context frame. Similarly, the `drpop` instruction should be used just before a return to restore the matching frame for the caller.

5.7 DSR Implementation

We implemented our DSR scheme as a link-time optimization pass in LLVM/Clang 3.8 for RISC-V, and use alias analysis algorithms from the PoolAlloc module [109].

5.7.1 Computing Equivalence Classes

We use Bottom-up Data Structure Analysis (Bottom-up DSA) [110] to categorize memory objects into equivalence classes. Bottom-up DSA is a context- and field-sensitive points-to analysis that scales well to large programs. It is context sensitive to arbitrary length acyclic call paths, and it is speculatively field-sensitive. It is field-sensitive for type-safe code, and falls back to field-insensitive for type-unsafe code. The algorithm is unification based and is not flow sensitive. The output of Bottom-up DSA is a *points-to graph* for each function, which incorporates the aliasing effects of all callees of that function (thus "Bottom-up"). A node in the points-to graph represents a set of memory objects joined through aliasing relationships, and nodes represent disjoint sets of objects. Each node therefore identifies a distinct equivalence class within that function. For each function and its associated points-to graph, we assign equivalence classes based on Bhaktar and Sekar's mask assignment algorithm [22], with a slight modification to differentiate the static and dynamic equivalence classes.

The first step in class assignment is identifying the dynamic equivalence classes. To handle indirect calls, we constrain all possible targets of an indirect call site to have the same dynamic classes. Bottom-up DSA can create classes of functions that are all callable from the same call site. The analysis result for these functions is a single points-to graph shared by all functions in the class. Within this graph all arguments and return values for these functions will share the same set of nodes. We use this functionality to compute the set of dynamic classes for all functions in the class simultaneously. We mark all nodes that are reachable from the pointer arguments and the pointer return values of each function in the class, and then remove all nodes that contain global variables or are marked un-encryptable. The resulting nodes become the set of dynamic classes for every function in the class. We use the same procedure for functions that are only called directly, but apply the procedure individually to each function.

For each node and its associated equivalence class, we assign a dynamic ID if a node is marked as dynamic and a static ID otherwise. If a node contains a global variable, we ensure that every such class in all functions uses the same static ID. If a node is marked un-encryptable, we assign it a null static ID which indicates that memory accesses to this class should not be instrumented.

5.7.2 Handling External Code and Data

To minimize the number of nodes that need to be marked as un-encryptable, we implemented wrapper functions for the library functions that our benchmark programs call (cf. subsection 5.5.3). A wrapper functions decrypts the variables in equivalence classes that may be accessed by an external function, and re-encrypt them when that external function returns. The wrappers must access keys from the Context Stack and use the new instructions for memory accesses. To ensure that the correct instructions are used, the wrappers are written in `C` using inline assembly code. We manually implemented the wrapper functions for all 71 C library functions used in SPEC CINT 2000. Implementing new wrappers is straightforward; writing a wrapper generally takes just a few minutes after consulting documentation such as man pages. Most wrappers have a predictable structure, and generating wrappers for many common cases could be automated by adding annotations to augment the type signature of the function with additional information. For example, an annotation would distinguish between different uses of `char*` arguments, indicating if the pointer refers to a single `char` variable, an array, or a null-terminated string.

5.7.3 Program Transformation

Our transformation pass runs after all analysis steps have completed. It starts by creating a constructor function that runs before the `main` function. The constructor encrypts the initial values of all global variables. After we create the constructor, we annotate load and store operations with the class ID assigned to the memory location accessed by the operation. Next, we insert the instructions to manage the Context Stack. For each call site, we construct a mapping from the class IDs of the actual arguments in the caller context to the dynamic IDs of the formal arguments of the callee function. We use this mapping to insert the `mksc` and `mkcc` instructions, which initialize the callee context. We insert `drpush` instructions directly before call instructions to switch to the callee's context, and we insert `drpop` before return instructions to restore the caller's context. During code generation, we emit the annotated loads and stores as specialized instructions with the dynamic or static equivalence class ID encoded into the immediate operand.

5.8 Hardware Implementation

We implemented the proposed hardware architecture by extending one of the instances generated by the Rocket Chip Generator [14]. This instance is composed of a Rocket Core [114] with a 16KiB L1 instruction cache, a 16KiB L1 data cache, and a 256KiB unified L2 cache. We extended this system with the two hardware components described in Section 5.6, the Key Cache and the Context Cache. We also modified the core pipeline to interact with these caches.

5.8.1 Instruction Encoding

To avoid intrusive changes to the existing instruction decoder, we designed our specialized instructions to resemble the instructions they are based on. Our specialized instructions differ from their

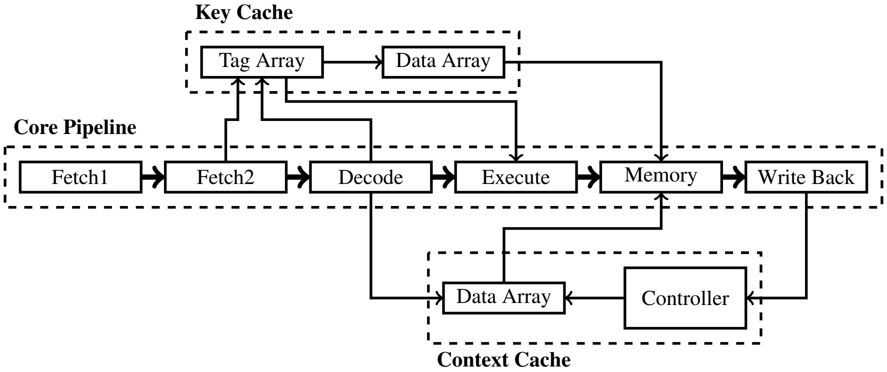


Figure 5.4: Overview of the modified Rocket core, showing the interaction between the original core pipeline, the Key Cache and Context Cache added by HARD.

base instructions in only one respect: the specialized ones interpret their immediate fields as index IDs, rather than memory offsets. This means that, like these memory offsets, the size of the index IDs is limited to twelve bits. We use the most significant bit of the index ID to indicate whether the index should be interpreted as an index into the Key Table, or an index into the current frame on the Context Stack. This leaves us with eleven bits to encode the ID itself.

The `mksc` and `mkcc` instructions each require two index ID operands, a source ID and a destination ID. For this reason, we based these instructions on the RISC-V instruction that can encode the longest immediate field, which is 20 bits long. The semantics of the instructions defines the type of index IDs they operate on, so we do not have to encode it in the MSB. The `mksc` instruction has a static ID (index into the Key Table) and a dynamic ID (index into the current context frame) as its operands, and the `mkcc` instruction has two dynamic IDs as operands. The size of these pairs of index IDs cannot exceed the available 20 bits. We therefore limit the size of dynamic IDs to nine bits, and the size of static IDs to eleven bits. This limits the size of the Key Table to 2048 entries and the size of the context frames to 512 slots. We analyzed a large number of programs and found that 512 is a realistic limit to the number of dynamic keys in a single context frame. We discuss the security impact of the Key Table size and how to handle programs that could use a larger number of keys in Section 5.10. The other context management instructions, `drpush` and `drpop`, are pseudo-instructions using the Control and Status Registers (CSR) interface.

5.8.2 Processor Pipeline

We modified the core pipeline to enable interaction with the Key Cache and Context Cache, as shown in Figure 5.4. The modified pipeline sends static and dynamic IDs to the Key Cache or Context Cache, which respond with the corresponding statically or dynamically assigned encryption keys respectively.

Key Cache

The Key Cache is a fully-associative cache that services requests for static IDs by loading the corresponding keys from its data array or from the Key Table in the memory. The Key Cache has two components: the pipeline depicted in Figure 5.4 and a miss handler. The cache has a tag array, containing a set of $(valid, id, \mathit{offset})$ tuples. For a static ID whose key is currently present in the Key Cache's data array, this tuple gives us the offset of that static ID's corresponding encryption key in the data array. The data array has a size of 2KiB, which means that it can contain 256 keys. If the core pipeline requests a key that is not present in the data array, the miss handler loads that key directly from the Key Table. Keys are never written back to memory upon eviction from the data array because the Key Table cannot be updated at run time.

Due to the tag array access and tag matching, our Key Cache takes two cycles to respond to a request, even if the requested key is present in the data array. To avoid stalling the Execute stage, we forward the raw instruction bytes from the Fetch2 stage to the Key Cache. The Key Cache uses a minimal decoder to determine if the forwarded instruction contains a static ID. If the instruction does indeed contain a static ID, the Key Cache will look up the corresponding key immediately. This allows the Key Cache to provide the Execute stage with the appropriate key without stalling if the key was present in the cache. Otherwise, it will stall the pipeline to fetch the key from the memory.

Context Cache

The Context Cache consists of two major components, the stack controller and the data array, following the design of a hardware stack with on-chip memory presented in earlier work [166]. The cache has dedicated registers to keep track of the locations of three context frames: the previous frame, the current (activated) frame, and the next frame. The Context Cache's data array is 1KiB, which is sufficient to store the top 128 slots on the Context Stack.

When the program copies an encryption key to the Context Stack using the `mkcc` or `mksc` instructions, the key will be stored directly in the corresponding slot of the next frame in the Cache's data array. This allows the Context Stack to minimize costly memory accesses. Whenever the program executes a `drpush` or `drpop` instruction to activate a different frame, the stack controller updates the dedicated registers accordingly. After executing one of these instructions, the cache may evict the oldest entries or fetch entries from memory depending on the available space in the data array. Eviction, fetching, and changes to the context frame registers happen at the last stage of the pipeline. This creates a possible hazard for other instructions accessing the Context Cache. We therefore modified the pipeline so that whenever a `drpush` or `drpop` instruction is decoded, or an eviction or fetch is in progress, any instructions that access the context cache are stalled until the `drpush`/`drpop` has finished executing, or the eviction/fetch has completed.

5.9 Evaluation

We implemented and tested several configurations of HARD's analysis and instrumentation passes and compared them to prior DSR implementations:

- The **Prior DSR** configuration mimics prior DSR implementations. For this configuration, we implemented a context-insensitive points-to analysis to calculate the equivalence classes, but we did not instrument accesses to safe objects and used weak encryption keys for unaligned

Benchmark	**Prior DSR**	**Full Key Size**		**Full Ctx Insensitive**		**Ctx Sensitive**	
	SW Only	SW Only	HW Supp.	SW Only	HW Supp.	SW Only	HARD
164.gzip	11.42%	40.17%	3.19%	70.63%	4.43%	70.94%	7.68%
175.vpr	20.14%	40.29%	8.67%	51.24%	9.64%	51.57%	9.81%
176.gcc	12.35%	22.43%	3.23%	29.00%	3.93%	34.68%	6.37%
181.mcf	7.91%	7.88%	3.70%	7.80%	3.74%	7.85%	3.69%
186.crafty	35.61%	58.81%	6.77%	68.20%	7.03%	70.83%	8.04%
197.parser	3.59%	7.21%	0.43%	17.97%	0.87%	25.17%	4.70%
252.eon	10.85%	17.51%	6.18%	18.21%	5.55%	22.59%	8.88%
253.perlbmk	1.65%	1.58%	0.46%	22.22%	1.35%	23.19%	1.11%
254.gap	14.69%	14.20%	5.75%	21.48%	6.32%	24.26%	6.64%
255.vortex	11.95%	28.32%	2.58%	28.75%	4.33%	43.68%	12.33%
256.bzip2	8.52%	76.04%	5.17%	83.92%	6.81%	83.98%	5.78%
300.twolf	16.11%	29.11%	3.51%	48.43%	4.47%	54.13%	4.70%
geomean	12.60%	26.99%	**4.11%**	36.96%	**4.85%**	40.96%	**6.61%**

Table 5.1: Run-time overhead of HARD and software-only DSR on SPEC CINT 2000. HARD's run-time overhead is lower than prior DSR implementations, which provide weaker security guarantee due to their less precise analyses and performance-oriented optimizations.

accesses (cf. subsection 5.5.2).

- The **Full Key Size** configuration uses the same analysis, but uses full 8-byte keys for all memory accesses (including unaligned accesses).

- The **Full Context Insensitive** configuration also uses the context-insensitive analysis, but encrypts accesses to *all* equivalence classes rather than just the unsafe ones.

- The **Context Sensitive** configuration uses HARD's context-sensitive analysis to calculate equivalence classes.

5.9.1 Performance

We measured the run-time overhead of all four of HARD's configurations and evaluated them with and without our architectural support. We instantiated HARD on a Xilinx Zynq ZC702 evaluation board using the *Rocket Chip Generator* [14]. The board has an FPGA running at 25MHz and has 256MiB of DDR3 memory. We ran the RISC-V port of the Linux kernel 4.1.17, and modified the kernel to initialize the Key Table and Context Stack on program startup. As our prototyping platform is severely resource constrained, we evaluated the run-time performance using the SPEC CINT 2000 instead of the more recent SPEC CPU 2006. For the same reason, we also ran the

Bench- mark	Prior DSR	Context Insensitive	Context Sensitive	Bench- mark	Prior DSR	Context Insensitive	Context Sensitive
164.gzip	77	127 (64.9%)	145 (88.3%)	nginx	250	348 (39.2%)	1396 (458.4%)
175.vpr	630	717 (13.8%)	801 (27.1%)	ProFTPD	424	646 (52.4%)	818 (92.9%)
176.gcc	1221	2115 (73.2%)	2957 (142.2%)	sshd	266	352 (32.3%)	412 (54.9%)
181.mcf	37	41 (10.8%)	41 (10.8%)	WU-FTPD	579	719 (24.2%)	745 (28.7%)
186.crafty	943	1133 (20.2%)	1161 (23.1%)	sudo	125	145 (16.0%)	178 (42.4%)
197.parser	289	343 (18.7%)	443 (53.3%)	mcrypt	177	223 (26.0%)	257 (45.2%)
252.eon	1160	1556 (34.1%)	1722 (48.5%)				
253.perlbmk	268	491 (83.2%)	528 (97.0%)				
254.gap	196	394 (101.0%)	499 (154.6%)				
255.vortex	763	911 (19.4%)	1598 (109.4%)				
256.bzip2	71	96 (35.2%)	106 (49.3%)				
300.twolf	442	692 (56.6%)	797 (80.3%)				
precision increase		(41.4%)	(68.1%)	precision increase		(31.2%)	(88.2%)

Table 5.2: The number of static equivalence classes that each analysis finds. For real world programs, HARD identifies and distinguishes 88.2% more equivalence classes. The impact was notable in case of `nginx` (458.4%), a widely used web server.

benchmark programs on the *train* inputs, as the board does not have enough memory to use the *ref* inputs.

Table 5.1 shows our evaluation results for the four configurations. For the *Prior DSR* configuration, we only have the results for software-only DSR as our hardware does not support variable-length keys. Each increasingly secure configuration incurs additional overhead, which is substantially reduced by HARD's hardware component. The overhead of the most precise configuration is 6.61% with hardware support, while the overhead of the software-only implementation is 40.96%.

5.9.2 Area Overhead

We used the `yosys` open synthesis suite to measure the hardware cost of HARD, and found that HARD adds 21% die area to an unmodified core. Since HARD makes modifications to the processor core and L1 caches only and `yosys` is unable to model L2 caches, the area cost is relative to the unmodified processor core + L1 cache only. This number would, in other words, be much lower if we also took L2 into consideration, or if we added HARD to a larger core such as those found in a mainstream x86 CPU.

5.9.3 Precision

HARD can only stop data-oriented attacks if it can place the legitimate targets of attacker-controlled instructions in different equivalence classes than memory locations the attacker wishes to access. If an attacker-controlled instruction accesses a memory location in the same equivalence class as its legitimate targets, an attack will likely succeed. This property also applies to other defenses that rely on static analysis to restrict data flow, including Data-Flow Integrity [35] and WIT [7]. Thus, it is important that the analysis distinguishes memory accesses into as many distinct equivalence classes as possible.

To demonstrate the added security of our context-sensitive analysis, we compiled several programs using three of the four different configurations of HARD and we counted the number of encrypted equivalence classes under each configuration. We excluded *Full Key Size* from this comparison, as it uses the exact same equivalence classes as *Prior DSR*. Table 5.2 shows the number of encrypted equivalence classes for each configuration, as well as the percentage increase from the first configuration.

We observe that HARD yields an increased number of equivalence classes compared to prior work and context-insensitive DSR. The greatest increase in the number of equivalence classes was for `nginx`. One of the reasons for the large improvement is that `nginx` uses a single logging function called from many different program locations. When using a context-insensitive analysis, all arguments to this function must be placed into a single equivalence class. With the context-sensitive analysis, the arguments to the logging function are in independent equivalence classes for different calling contexts. The loss of precision from a context-insensitive analysis increases the chances that an attacker will manage to find vulnerable code that encrypts data with the desired encryption key. It is important to note that the additional equivalence classes identified and protected by HARD include memory that is considered safe and thus left unencrypted by prior work (cf. subsection 5.5.2). This gives additional resistance against temporal memory vulnerabilities such as use-after-free or uninitialized-read.

Benchmark	Context Insensitive		Context Sensitive		Benchmark	Context Insensitive		Context Sensitive	
	Average	Max	Average	Max		Average	Max	Average	Max
164.gzip	1.21	8	1.09	8	nginx	3.56	2059	1.29	1318
175.vpr	1.21	71	1.13	48	ProFTPD	1.42	1586	0.88	1063
176.gcc	2.48	2824	1.87	2187	sshd	2.01	331	1.11	220
181.mcf	1.07	3	1.07	3	WU-FTPD	1.44	512	1.15	326
186.crafty	1.11	57	1.08	42	sudo	1.23	105	0.78	90
197.parser	1.73	379	1.41	290	mcrypt	1.01	151	0.90	140
252.eon	1.47	519	1.23	273					
253.perlbmk	4.13	1875	4.24	1872					
254.gap	4.18	1355	3.73	1270					
255.vortex	2.99	1521	3.73	1071					
256.bzip2	1.11	11	1.01	3					
300.twolf	1.05	18	0.91	9					

Table 5.3: Number of allocations per equivalence class.

Another important security property is the size of the equivalence classes, since the larger an equivalence class gets, the easier it generally becomes to illegitimately access variables within that class. To quantify equivalence class sizes, we modified our analyses to track the number of allocation sites (global, stack, and heap) contained within an equivalence class. For global and stack allocations, these correspond to variable declarations, for heap allocations they are calls to heap allocator functions like `malloc`. We counted both the average and maximum number of allocation sites per equivalence class, as shown in Table 5.3. The results show that, in general, the context-sensitive analysis used gives lower number of allocation sites across the benchmarks. Note that some benchmarks actually show an increase in average number of allocation sites. This is because an allocation site can be counted multiple times in different contexts with context sensitive analysis.

5.9.4 Real World Exploit

We evaluated our *Context Sensitive* configuration against a recent data-oriented attack presented by Hu et al. [94]. Instead of porting the attack to the RISC-V platform, we tested the software-only variant of HARD on x86 platform. The attack exploits a format string vulnerability in the *wu-ftpd* server to perform privilege escalation. Specifically, the attack overwrites a global pointer to a `struct passwd`. The overwritten pointer is later read and then dereferenced by the server, and the dereferenced value is interpreted as a user ID. This user ID is subsequently used as an argument for a `setuid` call. By overwriting the global pointer with the address of a memory location that contains a value of 0, which is the user ID of the root user, the attacker escalates the privileges of the vulnerable application.

We built two versions of the *wu-ftpd* binary: a base and a HARD'ened version. We then tested the exploit against both versions. The exploit was successful against the base version, but did not work against the HARD'ened version. While the attacker is still able to overwrite the pointer in the HARD'ened version, the subsequent read used a different encryption key than the instruction that overwrote the pointer, making it impossible for the attacker to reliably control the outcome of the overwrite. This causes the argument to the `setuid` call to be an unpredictable value. HARD identifies three equivalence classes involved in this exploit: the class accessed by the vulnerable instruction during valid executions, the class of the pointer variable, and the class used for dereferences of the pointer. These classes are accessed using distinct keys, k_v, k_p, and k_d respectively. To reliably control the result of this exploit an attacker would have to guess two 64-bit secret values, $k_v \oplus k_p$ and k_d, and therefore has a low chance of succeeding.

5.10 Limitations

Hardware Limitations. HARD limits the size of static IDs to eleven bits, which limits the number of equivalence classes with distinct keys to 2048. To run programs with over 2048 equivalence classes, we are forced to assign some static IDs to multiple equivalence classes. Other techniques that have a space constraint imposed on the protection mechanism are also limited in the protection they can provide. For example, the entries in the color table used by WIT [7] are 1-byte long, which limits WIT to use 256 distinct colors at most. HARD's limit of 2048 IDs allows it to protect much more complex programs than WIT. The security impact of static ID reuse could be reduced by carefully choosing which equivalence classes may share IDs.

Known Plaintext Attacks. Like the other DSR schemes, HARD is vulnerable against known plaintext attacks because it uses xor operations with fixed keys to encrypt data. If an attacker discloses encrypted data and knows the plaintext data, then she can recover the key which can be used to craft a successful payload. However, to reliably disclose data, she must know the data layout of the target program. Randomizing this data layout using ASLR or more fine-grained layout randomization can therefore mitigate this attack vector [51].

Lack of Integrity. Randomizing data using xor operations does not provide any integrity checking. This gives the attacker leeway to exchange encrypted data within the same equivalence class without knowing the key. In order to craft an exploit using this technique, the attacker will still need to know the meaning of the encrypted data, although they do not need to know the exact plaintext value. This is analogous to the limitation of many CFI approaches where an adversary can swap a pointer with another pointer as long as both pointers are allowed targets for a given indirect branch. The lack of integrity checking is an example of a performance-security trade off, and like CFI, DSR makes attacks substantially harder to construct.

Attacks on Skewed Values. Another attack vector against DSR is to target variables for which the range of valid values is a small subset of the possible values for the data type. An example is Boolean variables in C programs. A memory byte representing a Boolean value can have 2^8 different values, but only one of them will be interpreted as false. If an attacker wishes to change a false value to true, the attack will have a high probability of succeeding. In practice, many C programs are written such that Boolean variables will only have a limited number of values, often just 0 or 1. Attacks targeting these values could be mitigated by using a range analysis to identify the valid ranges and inserting checks to ensure the plaintext data is always within the allowed range.

Deployment Challenges. Hardware components have a longer time-to-market than a software based solution. However, hardware vendors have shown that they are willing to develop hardware components designed to prevent memory corruption exploits. Intel now provides Memory Protection Extension for bounds checking [158], and Control-Flow Enforcement Technology for control flow integrity [97]. These commercial offerings are driven by consumer demand for effective defenses with low overhead. HARD provides protection against a wide range of exploits with low performance overhead, using moderate amounts of hardware resources, so we feel it justifies the additional deployment challenges associated with a hardware-based solution.

5.11 Related Work

DSR was first proposed by Bhatkar et al. [22] and Cadar et al. [29]. Compared with those works, HARD provides greater security by using context-sensitive analysis and by randomizing all data using strong keys, which can be done efficiently thanks to our hardware support.

Data-Flow Integrity (DFI) [35] and Write Integrity Testing (WIT) [7] also perform alias analysis to build a set of equivalence classes and define a data-flow policy. However, they instrument the code to enforce the data flow and do not randomize the data representation. Both DFI and WIT used a context-insensitive analysis, so HARD can stop attacks that are not detected by either DFI or WIT due to the imprecise analysis. Thanks to its architectural support, HARD also incurs less performance overhead.

HDFI [182] introduced the notion of *Data-flow Isolation*, which allows a program to place sensitive data in isolated memory regions effectively and efficiently. The HDFI hardware is used to classify instructions and prevent those in one group from accessing the memory accessed by the instructions in the other group. However, HDFI only supports two groups because it uses one bit to distinguish each group. HARD can classify the memory regions into 2^{11} groups as it uses 11-bit IDs to identify which class an instruction should access. HDFI is not accompanied with an automated way to classify the instructions, while HARD relieves developers from this burden.

Enforcement of memory safety also mitigates data-only attacks because most attacks violate memory safety. Memory safety enforcement usually does not rely on the precision of static analysis and it provides a deterministic protection. A number of either software-only or hardware-based memory safety mechanisms have been proposed. However, some of these mechanisms cannot handle memory reallocation correctly [37, 65, 79, 193]. Others are incompatible with unprotected external code [147, 202]. More recently Softbound [134] used *fat pointers* with disjoint metadata to prevent violation of spatial memory safety and maintain compatibility with unprotected external binaries, and *low fat pointer* mechanisms [104, 106] have also been proposed to reduce the performance cost. Subsequently, CETS [135] was proposed to prevent the violation of temporal safety by using identifier to track the allocation states and disjoint metadata. Later DangSan, DangNull, FreeSentry and Oscar [57, 113, 189, 207] addressed this by nullifying, invalidating or not reusing the pointers to the freed objects.

Yang and Shin propose using a hypervisor to encrypt memory pages to provide memory secrecy from the operating system and other processes [205]. Similar to our work, this technique uses hardware (hypervisor mode) to support data encryption. However, an attempt to extend their technique to provide intra-process data isolation would change the page lifetime assumptions of their paper substantially, and incur substantial performance and memory overhead.

Works such as SeCage [118] or Intel's MPK [96] are designed to restrict memory access to protect secrets. These techniques could be used to control access to the encryption keys in HARD. However, these systems are primarily intended for infrequently used secrets, while HARD does consider any data "secret" and encrypts all program data. Our proposed hardware cache therefore provides a performant solution to protect many keys.

5.12 Conclusion

We presented a hardware-assisted defense against memory corruption attacks. HARD provides stronger protection than prior data space randomization implementations, with lower overhead. Our protection is stronger than prior work because (i) we use a context-sensitive analysis to distinguish more illegitimate data flows, (ii) we encrypt all possible equivalence classes to protect against all types of memory errors, and (iii) we always use 8-byte encryption keys to ensure sufficient key entropy.

Our hardware extension allows us to provide strong protection with low overhead. HARD's overhead is just 6.61% on average, which is 6 times lower than a software-only implementation of the same policy. The specialized hardware also protects encryption keys from information disclosure attacks.

Chapter 6

Contributions to Papers

Some of the chapters included in this thesis are based on papers which are the result of collaborative work. The below details my involvement in each of these works.

Chapter 3 BinRec is joint work with my co-first authors Taddeus Kroes and Anil Aaltitnay. I wrote the majority (>70%) of the text for the publication of the paper, and have greatly expanded the content for this thesis. I drove the main direction for the publication. I implemented the ELF stitching code, dynamic linking support, and external code trampoline prototypes, as well as significant development and evaluation tooling support.

Chapter 5 The first author of this work is Brian Belleville. I created data only attacks to evaluate the efficacy of the HARD defense and wrote the security evaluation. I also supported external code compatibility by implementing wrappers to decrypt data flowing to libc.

Chapter 4 The first author of this work is Nathan Burow. I performed the performance evaluation of the CFI works described in the paper. I wrote the performance section and outlined other sections of the paper.

Chapter 7

Conclusion

Let us discuss the conclusions we have reached on the research questions proposed in the introduction of this thesis.

- **How does dynamic analysis compare with static analysis for binary rewriting?** Dynamic analysis based binary rewriting holds promise to place control back in the hands of the users. It lets users overcome more barriers to reverse engineering and modification on top of static analysis based rewriting, like code obfuscation, see 3.6.2. On a more fundamental level, dynamic analysis has different challenges than static. It easily overcomes the 5 challenges for static rewriting, see 3.2, but has its own difficulties, namely achieving coverage, see 3.3.1. Lifting to a standardized IR is a great advantage for binary rewriting, because it allows reuse of analysis and transformation across frameworks. Dynamic analysis allow the production of a binary that allows a more minimal set of behaviors than static approaches while still remaining correct, which improves performance, see 3.5.2
- **What are the limits of the protection against memory corruption we can apply to binaries? Can we protect both data and control flows?** Binary rewriting enables attack surface reduction, as well as restricting the control and data flows to what the programmer

intended by retrofitting hardening transformation to off-the-shelf binaries. With proper engineering, binary rewriting frameworks can utilize security hardening transformations originally written for source programs, see 3.6. Control flow integrity, see Chapter 4, based on static analysis faces a larger challenge in binaries than in source programs, but dynamic analysis enables the application of a much more restrictive control-flow integrity policy than is a achievable statically, even with source code (see 3.6.1). Rewritten binaries have a unique emulation representation of the programs that transforms control-flow into data-flow. In Chapter 5 we introduced a state of the art defense that protects against data-only attacks with context-sensitive, hardware assisted data space randomization. It would be excellent to apply the protection to binaries, but the effective application of the technique depends on future progress towards the symbolization of semantic values in the rewritten programs, see 3.7.

Bibliography

[1] CodeVirtualizer. https://www.oreans.com/codevirtualizer.php.

[2] VMProtect. https://vmpsoft.com/.

[3] How to fix a program without the source code? patch the binary directly. https://arstechnica.com/gadgets/2017/11/microsoft-patches-equation-editor-flaw-without-fixing-the-source-code/, 2017.

[4] Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/support/, 2020.

[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ICFEM'05, 2005.

[7] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[8] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, H. Bos, C. Giuffrida, and M. Franz. Binrec: Attack surface reduction through dynamic binary recovery. In *Proceedings of the Fifteenth EuroSys Conference (EUROSYS)*. Association for Computing Machinery, 2020.

[9] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Eurosys*, 2013.

[10] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *ACM DRM*, pages 47–58, 2006.

[11] S. Andersen and V. Abella. Data execution prevention. changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies. http://support.microsoft.com/kb/875352/EN-US, 2004.

[12] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX SEC*, 2016.

[13] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFIX: Hardware-assisted flow integrity extension. In *Annual Design Automation Conference (DAC)*, 2015.

[14] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical report, University of California, Berkeley, Apr 2016.

[15] T. M. Austin, S. E. Breach, and G. S. Sohi. *Efficient Detection of All Pointer and Array Access Errors*, volume 29. ACM, 1994.

[16] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[17] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324–341, oct 1996.

[18] E. Bauman, Z. Lin, and K. W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[19] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, jun 1973.

[20] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC*, 2005.

[21] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, and M. Franz. Hardware assisted randomization of data. In *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID*, 2018.

[22] S. Bhatkar and R. Sekar. Data space randomization. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[23] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC)*, New York, New York, USA, 2011.

[24] A. Bougacha, G. Aubey, P. Collet, T. Coudray, J. Salwan, and A. de la Vieuville. Dagger: Decompiling to IR. `https://llvm.org/devmtg/2013-04/bougacha-slides.pdf`, April 2013.

[25] D. Bounov, R. Kici, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[26] D. Bruening, E. Duesterwald, and S. Amarasinghe. DynamoRIO. `https://dynamorio.org`.

[27] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *IJHPCA*, 2000.

[28] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 2017.

[29] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[30] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[31] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, 2011.

[32] 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 (USENIX Security 15)*, pages 161–176, Washington, D.C., 2015. USENIX Association.

[33] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX SEC*, 2015.

[34] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[35] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX Symp. Oper. Syst. Des. and Implementation (OSDI)*, 2006.

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

[37] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[38] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[39] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC*, 2006.

[40] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Network And Distributed System Security Symposium*, NDSS '14, 2014.

[41] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[42] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys*, 2010.

[43] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *EUROSYS*, 2012.

[44] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. HCFI: Hardware-enforced Control-Flow Integrity. In *CODASPY*, 2016.

[45] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 2000.

[46] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report,

Department of Computer Science, The University of Auckland, New Zealand, 1997.

[47] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM POPL*, pages 184–196, 1998.

[48] P. Collingbourne. LLVM — control flow integrity, 2015. `http://clang.llvm.org/docs/ControlFlowIntegrity.html`.

[49] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[50] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *CCS*, 2011.

[51] K. Cook. Introduce struct layout randomization plugin. `http://www.openwall.com/lists/kernel-hardening/2017/04/06/14`, 2017.

[52] J. Corbet. User-space page fault handling. `https://lwn.net/Articles/636226/`, 2015.

[53] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[54] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[55] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[56] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[57] T. H. Dang, P. Maniatis, and D. Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security Symposium*, 2017.

[58] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[59] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference (DAC)*, 2014.

[60] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[61] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM TOPLAS*, 2005.

[62] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[63] E. H. Debaere and J. M. van Campenhout. *Interpretation and instruction path coprocessing*. Computer systems. MIT Press, 1990.

[64] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.

[65] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Int. Conf. on Software Engineering (ICSE)*, 2006.

[66] A. Di Federico and G. Agosta. A jump-target identification method for multi-architecture static binary translation. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10. IEEE, 2016.

[67] A. Di Federico, M. Payer, and G. Agosta. Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 131–141, New York, NY, USA, 2017. ACM.

[68] A. Dinaburg and A. Ruef. McSema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[69] S. Dinesh, N. Burow, D. Xu, and M. Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and

sanitization. In *S&P*, 2020.

[70] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee. Tappan zee (north) bridge: Mining memory accesses for introspection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2013.

[71] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[72] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *USENIX TCON*, 1995.

[73] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[74] 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*, 2015.

[75] F. Falcon. Exploiting adobe flash player in the era of control flow guard. BlackHat EU'15 `https://www.blackhat.com/docs/eu-15/materials/eu-15-Falcon-Exploiting-Adobe-Flash-Player-In-The-Era-Of-Control-Flow-Guard.pdf`, 2015.

[76] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. `http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`, 2012.

[77] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, ACSAC '14, 2014.

[78] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symp. on Security and Privacy*, 2016.

[79] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman. Architectural support for low overhead detection of memory violations. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2009.

[80] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[81] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, S&P '14, 2014.

[82] Google. V8. `https://v8.dev`.

[83] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, nov 2001.

[84] E. Güler, C. Aschermann, A. Abbasi, and T. Holz. Antifuzz: Impeding fuzzing audits of binary executables. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[85] A. Gussoni, A. D. Federico, P. Fezzardi, and G. Agosta. Performance, correctness, exceptions: Pick three. In *Binary Analysis Research Workshop*, 2019.

[86] B. Hackett and A. Aiken. How is aliasing used in systems software? *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 69–80, 2006.

[87] B. Hardekopf and C. Lin. The ant and the grasshopper. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*, volume 42, page 290, New York, New York, USA, 2007. ACM Press.

[88] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298. IEEE, apr 2011.

[89] M. Hind. Pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '01*, pages 54–61, New York, New York, USA, 2001. ACM Press.

[90] M. Hind and A. Pioli. Which pointer analysis should I use? *ACM SIGSOFT Software Engineering Notes*, 25(5):113–123, sep 2000.

[91] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[92] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. Hqemu: A

multi-threaded and retargetable dynamic binary translator on multicores. In *CGO*, 2012.

[93] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 1980.

[94] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *USENIX SEC*, 2015.

[95] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[96] Intel Inc. Intel 64 and IA-32 architectures. software developer's manual, 2013.

[97] Intel R. Corporation. Control-flow enforcement technology preview, 2016.

[98] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Network And Distributed System Security Symposium*, NDSS '14, 2014.

[99] J. C. King. Symbolic execution and program testing. *CACM*, 1976.

[100] V. Kiriansky. Secure execution environment via program shepherding. Master's thesis, Massachusetts Institute of Technology, 2013.

[101] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[102] P. M. Kogge. An Architectural Trail to Threaded-Code Systems. *Computer*, 15(3):22–32, mar 1982.

[103] T. Kroes, A. Altinay, J. Nash, Y. Na, S. Volckaert, H. Bos, M. Franz, and C. Giuffrida. Binrec: Attack surface reduction through dynamic binary recovery. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[104] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. Sgxbounds: Memory safety for shielded execution. In *European Conference on Computer Systems (EuroSys)*, 2017.

[105] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[106] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[107] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P '14, 2014.

[108] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2004 International Symposium on Code Generation and Optimization*, CGO '04, page 75, Palo Alto, CA, 2004. IEEE Computer Society.

[109] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[110] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[111] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS*, 2010.

[112] D. Lea. A memory allocator, 1996.

[113] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[114] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC)*, 2014.

[115] E. Levy. Smashing the stack for fun and profit. *Phrack Magazine*, 7, 1996.

[116] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? *Compiler Construction*, pages 47–64, 2006.

[117] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM CCS*, pages 290–299, 2003.

[118] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[119] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[120] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, page 65–76, USA, 2006. IEEE Computer Society.

[121] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[122] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[123] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Trans. Comput. Syst.*, 2001.

[124] Microsoft. Visual studio 2015 — compiler options — enable control flow guard, 2015. `https://msdn.microsoft.com/en-us/library/dn919635.aspx`.

[125] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *ACM SIGSOFT Software Engineering Notes*, 27(4):1, 2002.

[126] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[127] Miscosoft. Setprocessvalidcalltargets function. `https://msdn.microsoft.com/en-us/enus/library/windows/desktop/dn934202(v=vs.85).aspx`, 2015.

[128] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets. In *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, 2001.

[129] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[130] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007.

[131] Mozilla. Spidermonkey. `https://ftp.mozilla.org/pub/spidermonkey/prereleases/60/pre3`, 2018.

[132] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[133] S. Nagarakatte, M. M. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. *ACM SIGARCH Comp. Arch. News*, 40(3):189–200, 2012.

[134] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[135] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Int. Symp. on Memory Management (ISMM)*, 2010.

[136] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[137] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[138] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2009.

[139] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[140] B. Niu and G. Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[141] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[142] B. Niu and G. Tan. Mcfi readme. `https://github.com/mcfi/MCFI/blob/master/README.md`,

2015.

[143] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015.

[144] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *CGO*, 2019.

[145] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[146] B. Patel. Intel releases new technology specifications to protect against rop attacks, 2016. `http://blogs.intel.com/evangelists/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/`.

[147] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw. Pract. Exper.*, 27(1):87–110, Jan. 1997.

[148] PaX. Address space layout randomization (ASLR).
`https://pax.grsecurity.net/docs/aslr.txt`, 2003.

[149] PaX. *Homepage of The PaX Team*, 2009. `http://pax.grsecurity.net`.

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

[151] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, 2015.

[152] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[153] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Network and Distributed System Security Symposium (NDSS)*, NDSS '15, 2015.

[154] M. Prasad and T.-c. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX ATC*, 2003.

[155] L. V. Put, D. Chanet, B. D. Bus, B. D. Sutter, and K. D. Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *ISSPIT*, 2005.

[156] C. Qian, H. Hu, M. A. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A framework for post-deployment software debloating. In *USENIX SEC*, 2019.

[157] T. Rains, M. Miller, and D. Weston. Exploitation trends: From potential risk to actual risk. In *RSA Conference*, 2015.

[158] R. Ramakesavan, D. Zimmerman, P. Singaravelu, G. Kuan, B. Vajda, S. Gibbons, and G. Beeraka. Intel memory protection extensions enabling guide, 2016.

[159] J. Reinders. Processor tracing. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2013.

[160] M. Rigger, R. Schatz, R. Mayrhofer, M. Grimmer, and H. Mossenbock. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. In *ASPLOS*, 2018.

[161] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15, 2012.

[162] E. Rohou, B. N. Swamy, and A. Seznec. Branch prediction and the performance of interpreters: Don't trust folklore. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[163] R. Rolles. Unpacking virtualization obfuscators. In *WOOT*, 2009.

[164] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '04*, page 14, New York, New York, USA, 2004. ACM Press.

[165] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, jan 2003.

[166] M. Schoeberl. Design and implementation of an efficient stack machine. In *IEEE International Parallel and Distributed Processing Symposium*, 2005.

[167] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[168] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the intel ia-32 architecture. In *WBT*, 2001.

[169] K. Serebryany. Sanitize, fuzz, and harden your C++ code. San Francisco, CA, 2016. USENIX Association.

[170] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[171] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[172] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07*, 2007.

[173] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. TRIMMER: application specialization for code debloating. In *IEEE ASE*. ACM, 2018.

[174] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *S&P*, 2009.

[175] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice Hall, 1981.

[176] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang. LLBT: an LLVM-based static binary translator. In *CASES*, 2012.

[177] B.-Y. Shen, W.-C. Hsu, and W. Yang. A retargetable static binary translator for the ARM architecture. *TACO*, 2014.

[178] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P*, 2016.

[179] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[180] Y. Smaragdakis and G. Balatsouras. Pointer Analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[181] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well. *ACM SIGPLAN Notices*, 46(1):17, jan 2011.

[182] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[183] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[184] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *Annual Design Automation Conference (DAC)*, 2016.

[185] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[186] 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 *USENIX Security Symposium*, 2014.

[187] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, oct 2000.

[188] A. van de Ven and I. Molnar. Exec shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[189] E. van der Kouwe, V. Nigade, and C. Giuffrida. Dangsan: Scalable use-after-free detection. In *European Conference on Computer Systems (EuroSys)*, 2017.

[190] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. PathArmor: Practical ROP protection using context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[191] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *CCS*, 2015.

[192] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *S&P*, 2016.

[193] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, 2007.

[194] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *IEEE/IFIP DSN*, pages 193–202, 2001.

[195] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[196] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *USENIX SEC*, 2015.

[197] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P'10*, 2010.

[198] 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*, 2012.

[199] D. Weston and M. Miller. Windows 10 mitigation improvements. BlackHat'16 `https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf`, 2016.

[200] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. Mimimorphism: A new approach to binary code obfuscation. In *CCS*, 2010.

[201] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2012.

[202] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[203] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin. ConFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proceedings of the 28th USENIX Security*, pages 1805–1821, Santa Clara, California, August 2019.

[204] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *S&P*, 2015.

[205] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[206] E. Yardimci and M. Franz. Mostly static program partitioning of binary executables. In *ACM TOPLAS*, 2009.

[207] Y. Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[208] P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[209] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX SEC*, 2018.

[210] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Network and Distributed System Security Symposium (NDSS)*, NDSS '15, 2015.

[211] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[212] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX SEC*, 2013.