UNIVERSITY OF CALIFORNIA,
IRVINE

Dynamic Binary Lifting and Recompilation

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Anil Altinay

Dissertation Committee:
Professor Michael Franz, Chair
Professor Alex Nicolau
Professor Isaac D. Scherson

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

VITA **vii**

ABSTRACT OF THE DISSERTATION **viii**

1 Introduction **1**
1.1 Overview 1
1.2 Contributions 5

2 Background **7**
2.1 The x86 Instruction Set Architecture 7
2.2 Stripped Binaries 8
2.3 Memory Corruption 8
2.4 Dynamic Linking Structures 9
2.5 Static vs Dynamic Disassembly 11
2.5.1 Static Disassembly 11
2.5.2 Dynamic Disassembly 14
2.6 Execution Driving Methods 15
2.6.1 Symbolic Execution 15
2.6.2 Concolic Execution 17
2.6.3 Selective Symbolic Execution with S^{2}E 18

3 BinRec **20**
3.1 Introduction 20
3.2 Current Limitations in Binary Lifting 23
3.2.1 $\boxed{\text{C1}}$ Code vs Data, and Reference Ambiguity 24
3.2.2 $\boxed{\text{C2}}$ Indirect Control Flow 24
3.2.3 $\boxed{\text{C3}}$ External Entry Points 25
3.2.4 $\boxed{\text{C4}}$ Ill-formed code 26
3.2.5 $\boxed{\text{C5}}$ Obfuscation 26

3.3 Design 27
3.3.1 Key Considerations for Dynamic Lifting 28
3.3.2 Dynamic Lifting Engine 29
3.3.3 Canonicalization 31
3.3.4 Lowering 36
3.3.5 Control Flow Miss Handling 37
3.4 Implementation 39
3.4.1 Parallel Tracing 39
3.4.2 Library Calls 40
3.4.3 Optimization 45
3.5 Evaluation 46
3.5.1 Comparison with static lifters 47
3.5.2 Performance 49
3.5.3 Code Coverage 50
3.5.4 Lifting Time 52
3.6 Applications 54
3.6.1 Control-flow Hijacking Mitigation 55
3.6.2 Virtualization-deobfuscation 57
3.6.3 AddressSanitizer 59
3.6.4 SafeStack 59
3.6.5 Attack Surface Reduction 60

4 Discussion and Future Work 63
4.1 Lessons Learned 63
4.2 Stack Symbolization with Static Analysis 64
4.3 Future Work 70

5 Related Work 73
5.1 Low-level binary analysis and rewriting 73
5.2 Code transformations using dynamic traces 74
5.3 Binary code lifting 75
5.4 Virtualization-deobfuscation 75
5.5 Sanitizers 76
5.6 Control Data Attack Mitigations 77

6 Conclusion 80

Bibliography 82

Appendix A Stack Unwinding Optimization 90

LIST OF FIGURES

Page

2.1 The first time that the function is called the got entry redirects to the plt instruction. 10
2.2 The Second time, GOT entry is filled with the address of memcpy. 11
2.3 Symbolic execution of Listing 2.1. `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. 17

3.1 The steps of binary recovery: lifting to compiler IR, transformation on the IR, and lowering back to machine code. 29
3.2 Address space layout of recovered code. 35
3.3 Execution time improvement from CPU state variable de-aliasing and global variable promotion. 51
3.4 Coverage with respect to the original binaries. The input set is the *ref* workload of SPEC CPU2006. 52
3.5 Incremental lifting progression of *bzip2*. 53
3.6 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. 56
3.7 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. 58

4.1 Ideal lifting 64

LIST OF TABLES

Page

3.1 Measured execution time normalized to the original binaries. Rev.ng results are reported from publication [49]. 49

3.2 Time in seconds to capture LLVM IR from input binaries with BinRec and McSema toolchains, alongside execution time for S^2E without BinRec instrumentation. For BinRec and S^2E we report the maximum time among the reference workloads from SPEC CPU2006. 54

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 [111]. 55

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`. 59

3.5 Attack surface reduction 62

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Professor Michael Franz. My time at UC Irvine has been a period of great professional and personal development, and I am extremely grateful for his guidance and encouragement throughout my studies at UCI. I had the opportunity to collaborate and work with an amazing group of researchers and friends in Secure Systems and Software Lab.

I would like to thank my past and current postdocs, Dr. Yeoul Na, Professor Stijn Volckaert (previously postdoc at UCI), Dr. David Gens, Dr. Adrian Dabrowski, Dr. Per Larsen for their contributions, ideas and advice that have facilitated my research efforts.

I also want to thank lab alumni, Andrei Homescu, Stephen Crane, Brian Belleville, Mohaned Qunaibit, Julian Lettner and Dixin Zhou for their invaluable support. In addition, I would like to thank my friends in the lab for being great colleagues. Taemin, Alex, Prabhu, Joseph, Dokyung, Min, Mitch, Fabian, Chinmay and Paul.

Finally, I would like to give special thanks to my doctoral committee members, Professor Isaac D. Scherson, Professor Alex Nicolau for their time and consideration.

Portions of this dissertation have been previously published in conference proceedings. I would like to thank my coauthors on these projects for their contributions on these publications, Professor Herbert Bos, Professor Cristiano Giuffrida and Taddeus Kroes.

This material is based upon work partially supported by the Defense Advanced Research ProjectsAgency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the UnitedStates Office of Naval Research (ONR) under contract N00014-17-1-2782, by the National ScienceFoundation under awards CNS-1619211and CNS-1513837. Any opinions, findings, and conclusionsor recommendations expressed in this material are those of the authors and do not necessarily reflectthe 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 anyother agency of the U.S. Government.

VITA

Anil Altinay

EDUCATION

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

Bachelor of Computer Engineering **2015**
Koc University *Istanbul, Turkey*

RESEARCH EXPERIENCE

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

Intern **Summer 2017**
Google, Inc. *New York*

TEACHING EXPERIENCE

Teaching Assistant **Fall 2016**
ICS 45C: Programming in C/C++ as a Second Language
University of California, Irvine *Irvine, California*

Teaching Assistant **Winter 2017**
ICS 31: Introduction to Programming
University of California, Irvine *Irvine, California*

PUBLICATIONS

BinRec: Dynamic Binary Lifting and Recompilation **EuroSys 2020**
A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz
In Proceedings of the Fifteenth European Conference on Computer Systems

BinRec: Attack Surface Reduction Through Dynamic Binary Recovery **FEAST 2018**
T. Kroes, **A. Altinay**, J. Nash, Y. Na, S. Volckaert, H. Bos, M. Franz, and C. Giuffrida
In Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation

ABSTRACT OF THE DISSERTATION

Dynamic Binary Lifting and Recompilation

By

Anil Altinay

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael Franz, Chair

Legacy binaries that do not have source code remain a vital part of our software ecosystem. Lifting and recompilation of legacy binaries allows for a wide range of late program transformations such as security hardening, deobfuscation, and reoptimization even when the source code is unavailable. Existing binary lifting approaches are based on static binary disassembly which has several limitations. Distinguishing code from data statically, for example, is undecidable in the general case [56]. Static disassembly must rely on heuristics and assumptions to disassemble binaries in the absence of dynamic information and high-level language semantics. Consequently, static disassembly cannot reliably handle indirect jumps, inline assembly, and obfuscated code. Lifting approaches that rely on static disassembly, therefore, often produce unsound binaries.

Dynamic disassembly of binaries can circumvent the limitations of static disassembly including the ability to handle obfuscated and encrypted binary code. In this dissertation, we present BinRec, a new approach to heuristic-free binary recompilation which lifts *dynamic traces* of binaries to a compiler-level intermediate representation (IR); the lifted IR is lowered to a "recovered" binary by taking advantage of the existing compiler toolchain. Our approach allows applying rich program transformations, such as compiler-based hardening and 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 front end can accurately disassemble and lift binaries without heuristics, and we can successfully recover all SPEC INT 2006 benchmarks including C++ applications. We evaluate our approach in four application domains: i) binary reoptimization, ii) deobfuscation (by recovering partial program semantics from virtualization-obfuscated code), iii) binary hardening (by applying existing compiler-level passes such as AddressSanitizer and SafeStack on binary code), and iv) attack surface reduction in the recovered binary (by removing unused program paths).

Chapter 1

Introduction

1.1 Overview

Software systems are an amalgamation of several software components working together. Some of those components either do not have their source code available for recompilation or the toolchain used for their compilation is not supported anymore and they are referred to as *legacy binaries*. The security guarantees of our software ecosystems are only as strong as its weakest link; protecting legacy binaries from exploits is a notoriously hard problem for a number of reasons. Primary among them is that we cannot take advantage of the developments in compiler toolchain to be applied to these binaries.

Memory corruption errors, for example, are a major threat to software security. Exploiting memory corruption errors can lead to system compromise; particularly the software written in unsafe languages such as C and C++ are prone to memory corruption errors as they do not automatically check for memory accesses and any illegal access to memory is considered undefined behavior. Though alternative memory safe languages are available, C and C++ are widely used due to efficiency, direct control over hardware resources, developer familiarity, and compatibility with existing software and systems. Memory unsafe languages leave the responsibility of memory management to developers and programming errors can lead to memory corruption vulnerabilities. There are several varied root causes of memory corruption errors such as dereferencing freed pointer or use of uninitialized pointer [96, 99], etc., besides the above mentioned undefined behavior.

Even when the source code is available for analysis, memory corruption errors may remain undetected for a long time due to two reasons: 1. The actual memory corruption error may only surface under specific conditions and reproducing the error consistently becomes harder as systems become more complex; 2. The actual effects can often come to light far from the source of the error, and hence, correlating cause and effect may be difficult. These bugs are typically introduced due to human error, and hence, practically all software systems that use memory unsafe languages suffer from such bugs [99, 96].

In the past, attackers have found ways of using memory corruption errors to their advantage. A popular mode of exploit is remote code execution, i.e., the ability to divert the legal control-flow of the target application and execute arbitrary code. Control-flow exploits can perform Turing-complete operations [84] by stitching together small instruction sequences (gadgets) to effect privilege escalation, information leak or denial of service attacks. Over the years, attackers have improved their attack methods and exploited new classes of memory corruption errors in their exploitations. As a result, a previously benign memory corruption error can become exploitable when a new exploitation method is discovered.

One way to avoid memory corruption errors is to rewrite the software using memory safe languages. In practice, however, this is unrealistic because there are already millions of lines of existing C/C++ code; further replicating low-level features of C/C++ would be necessary to produce performant systems software (e.g. operating systems). Identifying and eliminating memory corruption errors can help reduce the attack surface available for exploits. We have seen improvements in source analysis and bug finding tools in the form of fuzzers and sanitizers. In spite of the developments in identifying memory corruption errors, eliminating all errors is not possible for programs written in C and C++ programs and we must assume the presence of memory corruption errors in software shipped to production.

Exploit mitigations that thwart attacks have been proposed to address this problem. In the arms race between attacks and defenses [96], the recent decade of security research has produced fruitful control-flow mitigations such as Data Execution Prevention(DEP) [10], Address Space Layout Randomization(ASLR) [78], and Control Flow Integrity(CFI) [6]. These mitigations may employ static analysis, dynamic analysis, or a combination and typically rely on the source code of the software. Adoption of these mitigations is widespread and the adversaries now look to exploit legacy binaries as they cannot be protected using source code based modern mitigations. For example, Microsoft equation editor is a legacy binary that is part of the Microsoft Office suite and distributed widely; while the rest of the software components with source code are well protected the equation editor contained several exploitable memory corruption errors. Microsoft employed binary rewriting to modify the executable of the equation editor to fix these errors [79].

The existing literature proposed methods through which we can retrofit legacy binaries with modern mitigations. To modify any binary file, the binary code must first be disassembled correctly. *Binary rewriting* techniques modify the disassembled binary code to fix security errors or add mitigations [111, 69, 98, 100, 80, 27]. This approach modifies binary at machine code level or low-level representation [104, 103, 44] and retain the original code and data layout the program to an extent to produce sound patched binaries. This approach is effective and useful but cannot reuse the developments in compiler toolchain. To address this, the *Binary Lifting* approach was proposed; binary lifting approaches raise the machine instructions to a higher level intermediate representation such as LLVM IR [8, 43, 42]. This promising approach can not only overcome the limited expressiveness of assembly code but can go further and reuse the modern developments in compiler toolchain to perform binary hardening and reoptimization.

The existing lifting approaches employ static disassembly to retrieve machine instructions from binary code which are then translated to LLVM IR. Static disassembly, however, suffers from a number of fundamental limitations. Distinguishing code from data statically, for example, is undecidable in the general case [56]. Static disassembly must rely on heuristics and assumptions to disassemble binaries in the absence of dynamic information and high-level language semantics. Consequently, static disassembly cannot reliably handle indirect jumps, inline assembly, and obfuscated code [56, 11]. Some of these challenges can be addressed through *heuristics* and by making specific assumptions [8, 104]. However, binaries from source code are generated using a variety of compilers and differing optimization techniques. The heuristics based disassembly, therefore, are not generic enough to address the variance we find in real-world legacy binaries. To compound the problem further, software vendors and malware authors often intentionally obfuscate or encrypt their binary code which cannot be handled by static disassemblers without the loss of generality. Existing lifting approaches that rely on static disassembly, hence, often produce unsound binaries.

In contrast to static approaches, dynamic analysis based solutions analyze concrete executions of the target program, and thus can seamlessly handle all statically unknown elements such as mixed code and data, and indirect control-flow targets. However, existing dynamic analysis approaches have limited usability because they incur relatively high overhead [68], and the rewritten binary code is tailored to the tool's environment. This makes the distribution of rewritten binaries not as straightforward as with static analysis based approaches. Pin [68], DynamoRIO [2] and Valgrind [76] are some of the well-known dynamic binary analysis platforms.

To produce reliable, sound disassembly, dynamic analysis of binaries is necessary. In this dissertation, we propose a dynamic lifting approach that produces new *recovered* binaries from the observed execution traces of the *original* input binaries. We propose solutions to overcome some of the hard problems in employing scalable dynamic analysis that can be used to produce a lifting approach that is compiler agnostic. We present our implementation prototype BinRec—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, our approach is the first binary lifting framework based on dynamic disassembly, enabling lifting of statically unknown code for the first time. Additionally, our solution represents the first dynamic binary rewriting approach that persists its transformations in a standalone output binary.

1.2 Contributions

This dissertation makes the following contributions:

- We present BinRec, the first dynamic binary lifting framework. BinRec uses 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.

- We show that BinRec 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 BinRec in four application domains: i) Binary reoptimization, 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 AddressSanitizer and SafeStack. iii) Binary deobfuscation through successful recovery of partial program semantics in virtualization-obfuscated binaries.

iv) Attack surface reduction in the recovered binary by removing unused program paths and by reducing the number of ROP gadgets.

- We discuss future directions for improved analysis of lifted code that can produce more optimal binaries.

Chapter 2

Background

2.1 The x86 Instruction Set Architecture

In this thesis, we used x86 executables for our experiments which is perhaps the hardest to disassemble. We studied x86 binaries for two main reasons: 1. The x86 ISA is very common in the consumer market such as desktop computers and in binary analysis research partly because of its popularity. 2. The x86 ISA is very complex and has its own unique challenges for disassembly.

The x86 ISA is a Complex Instruction Set Computing (CISC) architecture. The majority of byte values in x86 ISA represent a valid opcode since it has a very dense instruction set. This makes it harder for disassemblers to distinguish code from data. In addition, instruction size is variable length, and unaligned memory accesses for all valid word sizes are allowed. These features of x86 ISA allows programs with complex constructs; instructions can be overlap which also makes disassembly of x86 instructions difficult.

2.2 Stripped Binaries

During compilation, compilers are capable of emitting symbol information about functions and variables. Software vendors often modify the compiler configuration to *strip* the symbol and debug information from the binaries that are shipped to customers to prevent reverse engineering of binaries. The symbols among other things can describe how functions and variables are mapped within the binary. For example, a function symbol may have information about function names, their start addresses, and sizes. This information is mainly used by the linker to combine object files.

Symbol information may provide extensive details about the semantics of the program to assist the debugging process. For ELF binary format which we used in our experiments, debug information is emitted in DWARF [34], a standardized debug data format. Some of the previous works rely on such symbol information for disassembly and analysis. Our approach is generic and can handle stripped binaries. Approaches that assume presence of symbol information will likely fail to handle real-world stripped binaries.

2.3 Memory Corruption

System software suffers from memory corruption errors due to the use of unsafe languages. These errors are classified as either spatial or temporal memory safety violations [12, 75, 74]. A spatial memory safety violation is caused by dereferencing a pointer that points outside the bounds of its referent. Dereferencing uninitialized pointers and out-of-bound array accesses are examples of spatial errors. A temporal violation is caused by dereferencing a pointer whose referent has been deallocated. One of the examples of temporal memory errors is references to temporarily allocated memory with an expired lifetime, such as references to stack variables of callee after callee returned. Another example is accesses to `free`'d memory (e.g. use-after-free and double-free bugs). A program is said to be memory safe in the absence of any spatial and temporal memory errors.

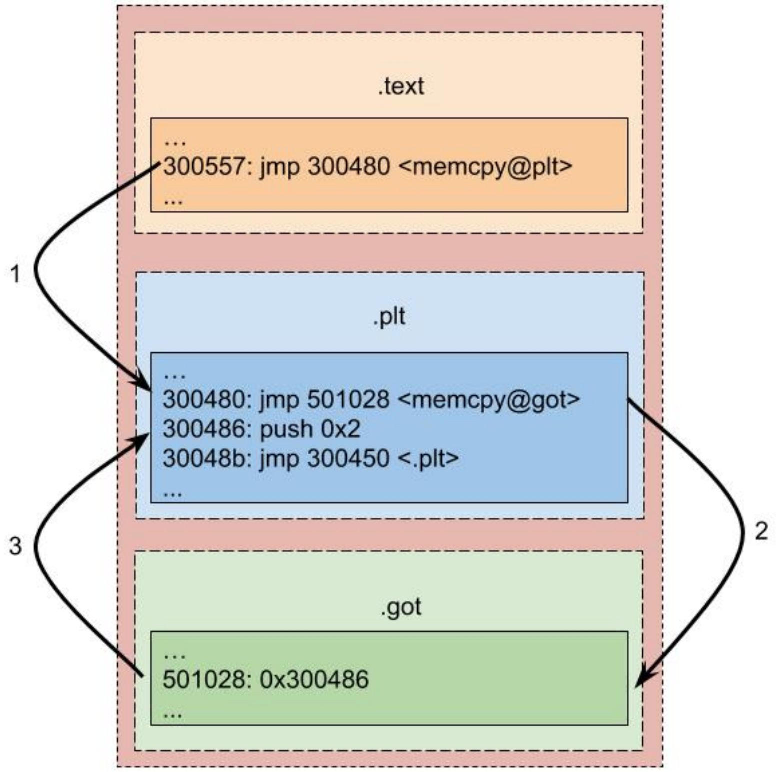
There are many techniques to detect memory safety violations [7, 93], however, they are still a continuing source of problems in modern software [96, 99]. Moreover, attackers have improved their exploitation methods and incorporated new classes of bugs in their exploitations. For example, the use of uninitialized memory-previously thought benign- has been categorized as a major attack vector against the Linux kernel [67, 71].

2.4 Dynamic Linking Structures

Binaries are either statically linked or dynamically linked. Statically linked binaries contain all the code and dependencies within the binary itself. They do not rely on external libraries. In contrast, dynamically linked binaries depend on external libraries (shared libraries) for full functionality. Shared libraries allow multiple processes to use the same code and eliminates the need to store shared library code in every binary.

In dynamically linked binaries, calls to shared library functions rely on information from the import table. The import table consists of a list of symbols of library functions created by the linker. These symbols are mapped to addresses in the corresponding loaded libraries by the dynamic linker. This import table is called the Global Offset Table (GOT) in ELF binaries.

When a dynamically linked binary calls a shared library function (`memcpy`), the call actually goes to Procedure Linkage Table (PLT). PLT is located at the .plt section within the binary. The PLT contains code stubs that direct the execution to GOT to find the corresponding address for shared library function (`memcpy`).



```
...
300557: jmp 300480 <memcpy@plt>
...
```

```
...
300480: jmp 501028 <memcpy@got>
300486: push 0x2
30048b: jmp 300450 <.plt>
...
```

```
...
501028: 0x300486
...
```

Figure 2.1: The first time that the function is called the got entry redirects to the plt instruction.

If a shared library function is called the first time, the GOT will have a pointer back to PLT to invoke a dynamic linker. The dynamic linker will locate the actual address of the shared library function and write it to the GOT as illustrated 2.1.

In later calls, since the GOT is already filled with the address of the shared library function, the dynamic linker will not be invoked as we illustrated in 2.2. This is called lazy binding. Invoking the dynamic linker is computationally expensive and lazy binding delays this cost to when the function is actually called. Also, this cost will never occur for functions that are never called.

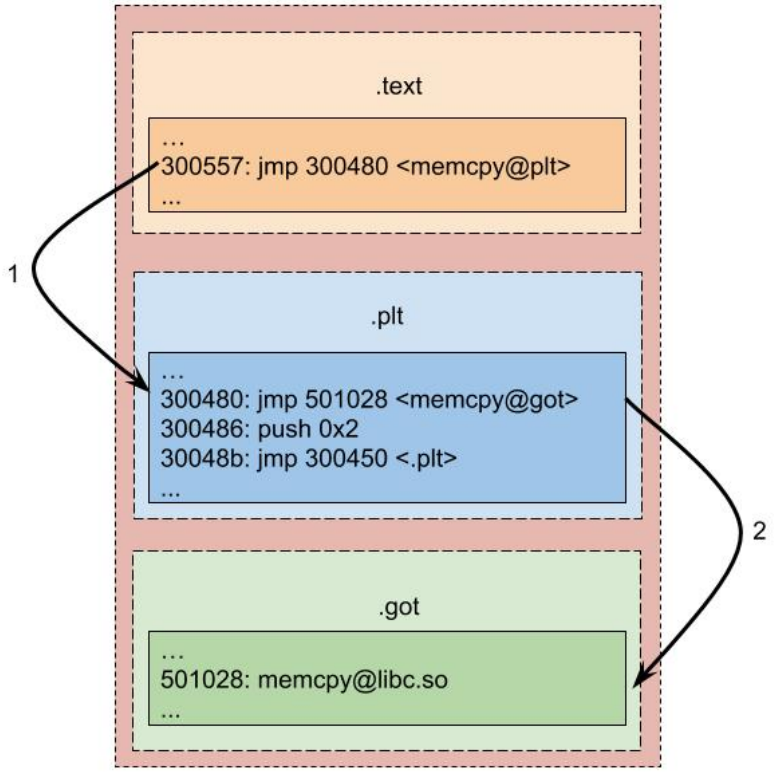

.text
...
300557: jmp 300480 <memcpy@plt>
...
1
.plt
...
300480: jmp 501028 <memcpy@got>
300486: push 0x2
30048b: jmp 300450 <.plt>
...
2
.got
...
501028: memcpy@libc.so
...

Figure 2.2: The Second time, GOT entry is filled with the address of memcpy.

2.5 Static vs Dynamic Disassembly

In order to lift machine code from a binary to higher-level IR, first, we need to identify which bytes of the binary are actual code. Then we can generate higher-level IR from the disassembled code. Broadly speaking, we can disassemble the binary in two ways: static and dynamic. Both approaches have their own advantages and disadvantages.

2.5.1 Static Disassembly

Static disassembly tries to disassemble all the code in the binary without executing it. Before we talk about different static disassembly approaches, we will first present some of the challenges.

Overlapping basic blocks Different functions may share the same basic blocks. This could hinder the analysis for distinguishing functions during disassembly.

Overlapping instructions The x86 instruction-set architecture allows unaligned memory accesses and instructions size is variable length. This allows jumps to target any offset within a multi-byte instruction. As a result, the same code bytes can be used for multiple instructions and may confuse disassemblers.

Data and padding bytes in code Binaries, especially compiled with a high level of optimizations,, contain padding code such as nop instructions between functions and basic blocks within a function. The purpose of padding code is to create proper alignment for functions and basic blocks in memory so that optimal execution will be achieved. Moreover, some compilers like Visual Studio mix data such as jump tables with code to create better-optimized binary. However, such data can be interpreted as code by disassemblers and cause the wrong disassembly.

Indirect control flow transfers Indirect control flow transfers such as indirect branches and calls are resolved at runtime and resolving their target addresses statically is a very hard problem. Functions and basic blocks reached only with such instructions can cause incomplete disassembly when recursive disassembly is employed since this approach disassembles the code by following control flow transfers.

Non-contiguous functions Functions do not have to be laid out in a single contiguous memory range. Especially in higher optimization levels, compilers may emit functions on multiple disjoint memory ranges. In this case, disassemblers which assume that each function is laid out in a single contiguous memory range may fail.

Uncommon prologues/epilogues Some disassemblers rely on well-known signatures of function prologue and epilogue. However, function prologue and epilogue becomes hard to recognize when the binary is compiled at high optimization levels and cause mis-identification of such functions.

Obfuscated code While compilers may emit complex constructs in high optimization levels, there are also obfuscation techniques that create more complex constructs on purpose even at the expense of creating a slower binary. There are many different obfuscation techniques which are often used in malware or in proprietary software to prevent reverse engineering [31]. Skype is one of the examples of such software.

There are three main approaches to static disassembly: linear, recursive and superset disassembly.

Linear disassembly is one of the simplest approaches. It decodes all bytes consecutively in the code segments and parses them into a list of instructions. One of the main problems of this approach is that there may be bytes that are not instructions. Some of the modern compilers mix data, such as jump tables, with code to have a better-optimized binary. The disassembler may not be able to distinguish this data from code and generate invalid opcodes during parsing of this data. Sometimes this data may correspond to valid opcodes and cause the disassembler to desynchronize with respect to the correct instruction sequence. This would cause instructions following the data to be disassembled in the wrong way. Such a scenario is very probable on dense ISAs with variable instruction size such as x86. Eventually, the disassembler will resynchronize but instructions just after the data may be missed.

Recursive disassembly avoids the problems of linear disassembly by following control flow instructions. It begins to disassemble the binary from known entry points such as the main function similar to linear disassembly until it reaches control flow instructions. Then it recursively follows control flow instructions to find new code. This way recursive disassembly prevents interpreting data bytes as code. However, recursive disassembly has its own limitations. The target address of indirect calls and jumps are not known during static analysis. Therefore, basic blocks and functions that are reachable only with indirect control flow instructions are likely to be missed. While some of the well-known disassemblers like IDA Pro utilize recursive disassembly to address this limitation by employing compiler specific heuristics, they suffer from the error prone nature of these heuristics.

Superset disassembly has been introduced to generically disassemble a binary without heuristics by keeping a superset of legal instructions [13]. It creates the superset of legal instructions by disassembling the binary from each offset. While this makes the recovered binary include some illegal instructions resulting from incorrect disassembly, only valid ones will be reached at run time. This approach is generic and effective for low-level binary rewriting. This, however, may not be suitable when we lift a binary to high-level IR code, because IR having invalid instructions confuses many compiler analyses including the control-flow analysis and this may hamper compiler optimization or even lead to an incorrect compilation result. Relying on static disassembly, therefore, lifting real-world binaries and recompiling them remain ineffective in practice.

2.5.2 Dynamic Disassembly

In contrast to static disassembly, dynamic disassembly executes the binary under analysis and discovers instructions by tracing them as they are executed. Executing the binary gives a lot of extra information that can address the problem of static disassembly. For example, the target of indirect control flow transfer instructions that were not available during static disassembly becomes clear since we can see which instruction is executed after. Moroever, the problem of interpreting data as code is not possible since execution would never execute data as code.

However, dynamic disassembly has its own disadvantages. The main disadvantage is code coverage [72]. Each execution of the binary exercises some part of the binary and this leads to partial code coverage. Complete code coverage is a very challenging problem given that there are an infinite number of possible inputs a program can take and we do not know which input would execute the code that we have not observed in previous runs.

2.6 Execution Driving Methods

The main challenge of dynamic analysis is that driving the execution through all desired code paths [72]. Desired code paths, however, depends on the aim of analysis. If the aim of the analysis is to test the binary, driving the execution through all the possible code paths may be desirable. However, if the aim is debloating the binary, only those paths that are needed for specific functionality should be explored.

2.6.1 Symbolic Execution

Symbolic Execution is a popular program analysis technique used in automated software testing [18] and malware analysis [72] to find program inputs that increase code coverage. When the program is symbolically executed, each input to the program is represented with a symbolic value (α, β, ..) instead of a concrete value ("hello", 5, ..) which the program would normally get. Symbolic execution allows exploration of multiple control flow paths in parallel that program would normally take with different concrete inputs.

Listing 2.1: Excerpt of *eq.c*: eq example in C.

```c

#include <stdio.h>
int main(int argc, char **argv) {
    if(argc == 3){
        char a = argv[1][0];
        char b = argv[2][0];
        if (a == b) {
            puts(”leading␣characters␣are␣equal”);
        }
    }
    return 0 ;
}
```

Symbolic execution is performed by a symbolic execution engine which maintains a first-order Boolean formula and a symbolic memory store for each explored control flow path. First-order Boolean formula represents a list of branch conditions satisfied along the path and Symbolic memory store maps variables to symbolic expressions or values. The symbolic memory store is updated after assignments while the Boolean formula is updated after branch execution. The Boolean formula created along the path can be given to satisfiability modulo theories (SMT) solver [59] to verify if the path is reachable.

The symbolic execution engine forks the program state when it reaches a branch condition that involves symbolic values. Each forked state represents a different control flow path with its own version of the program state.

In order to illustrate symbolic execution we have an example program called eq in Listing 2.1. eq checks whether the user provided exactly two parameters and compares the first byte of these two parameters. If they are equal, it writes ”leading characters are equal” to the stdout.

Figure 2.3 shows how we symbolically execute Listing 2.1 by passing two symbolic arguments of one byte each, recovering all possible code paths.

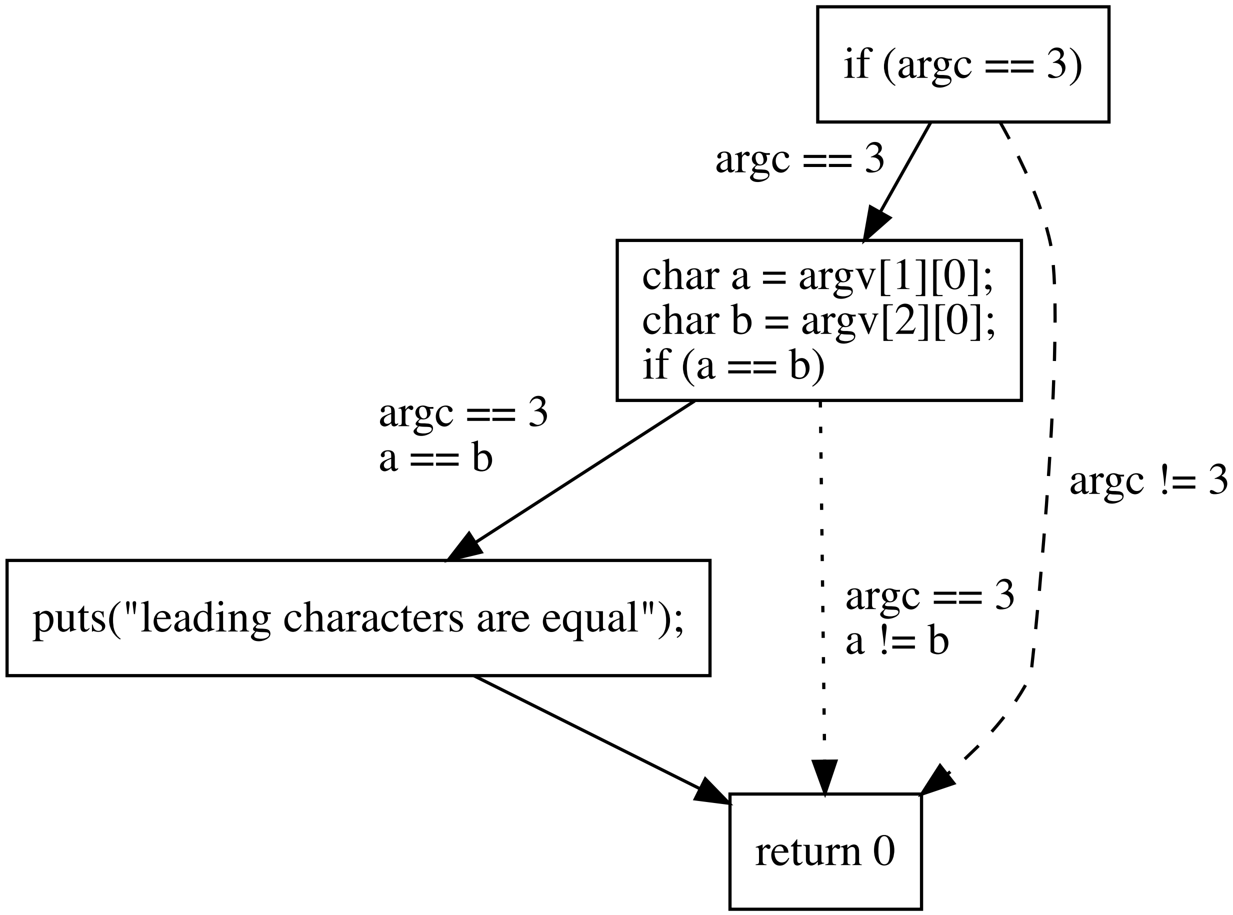


Figure 2.3: Symbolic execution of Listing 2.1. `argc`, `a` and `b` are symbolic values, causing the execution state to fork twice as represented by the different arrow styles. Edge labels show the constraints recorded in each execution state.

2.6.2 Concolic Execution

As we have shown in the previous section, in theory, symbolic execution can generate all possible control flow paths that the program could take with concrete inputs. However, this is infeasible in practice, particularly on real-world programs.

One of the limitations of symbolic execution is that the possible number of control flow paths to be explored increases exponentially when program size increases linearly. This is called a state explosion problem. Another limitation is that constraint solver may take a lot of time when solving complex constraints. Because of these reasons, a symbolic execution engine may terminate before it explores all the control flow paths in the program. Moreover, real-world programs are not independent of their environment: it is hard for symbolic execution engines to analyze the whole software stack and evaluate any possible side effects. Some of the proposed solutions to these problems are path finding heuristics to increase code coverage and parallel execution of independent states to lower execution time. KLEE [22], one of the popular symbolic execution engines offers path finding heuristics to reach optimal code coverage and to direct execution towards certain target code by prioritizing one path over others.

Concolic execution is another solution proposed to make symbolic execution feasible. Since symbolic execution is slow compared to concrete execution, a combination of two called concolic execution is proposed to make symbolic execution feasible for real world programs. One of the popular approaches of concolic execution is called dynamic symbolic execution (DSE) or dynamic test generation [46] in which concrete execution directs symbolic execution. Another popular approach is called selective symbolic execution [29] which allows symbolic execution of some components of the software stack while executing the other components of the software stack concretely.

2.6.3 Selective Symbolic Execution with S^{2}E

S^{2}E is a framework that allows concolic execution of binary programs with the illusion of full-system symbolic execution. It is developed on top of QEMU and KLEE. By default, code runs in DBT mode to have high performance. Once a symbolic memory value is used, a custom LLVM back-end for the Tiny Code Generator (TCG) of QEMU is invoked to generate LLVM IR. Then generated LLVM IR is symbolically executed with KLEE. Execution can seamlessly switch back and forth between the symbolic execution and the concrete execution, and the system state is suitably converted on every boundary crossing.

S^{2}E framework allows exploration of code paths in multiple execution modes for binaries. Moreover, it allows creation of LLVM IR for executed code. Therefore, it provides a solid foundation for our dynamic binary lifting and recompilation framework.

Chapter 3

BinRec

3.1 Introduction

Binary rewriting [104, 103, 44] has many applications such as post-installation program hardening [111, 69, 98, 100, 80, 27], (de)obfuscation [107, 35, 106], and reoptimization [39]. 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 [8, 43, 42]. 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 [56, 11].

While heuristics have been used to successfully circumvent these limitations for certain binaries that adhere to specific assumptions [8, 104], 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 [13].

In contrast to static translation methods, dynamic binary translation (DBT) tools such as Pin [68], DynamoRIO [2] and Valgrind [76] 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 BinRec—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 reacahes a code path that was not covered during lifting. Much like page faults are handled by a page fault handler in modern operating systems [36], our technique handles control flow misses by 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, our approach allows us to harness the power of existing IR-level compiler analyses and transformations on binaries where static lifting fails. Our evaluation on SPEC CPU2006 shows that our approach successfully lifts code patterns in optimized input binaries that state-of-the-art static lifters such as McSema [43] and Rev.ng [42] cannot. To demonstrate the immediate benefits of lifting binary code to compiler IR, we show that our method improves performance of some of our non-optimized input binaries and successfully applies two security transformations available in LLVM—SafeStack [61] and AddressSanitizer [87]—to our lifted IR. In contrast to previous binary rewriting approaches, our approach 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.

In summary, our contributions are the following:

- We present BinRec, the first dynamic binary lifting framework. BinRec uses 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.
- We show that our approach 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 our approach in three application domains: i) Binary reoptimization, 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 AddressSanitizer and SafeStack. iii) Binary deobfuscation through successful recovery of partial program semantics in virtualization-obfuscated binaries.

3.2 Current Limitations in 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 [11, 56], 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 C1 Code vs Data, and Reference Ambiguity

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 [56], so state-of-the-art analysis tools employ heuristics to approximate the correct label set [104, 111, 103]. 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 [103] 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 C2 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 [56]. Static approaches do achieve high accuracy when identifying the potential targets of iCFT instructions that load their destination address from jump tables [41, 111]. Resolving indirect function calls and returns, on the other hand, remains a challenge. Wang et al. [104] argue handling iCFTs can be supported through their approach, but their prototype Uroboros does not handle iCFTs. The underlying analyses [41] used in Rev.Ng [42] claim 90-95% jump target recovery depending on architecture.

Meanwhile, Qian et al. [82] as well as Zhang and Sekar [111] 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 C3 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 [42] and incurs significant overhead [13]. 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 [8, 105]. However, they cannot correctly handle external callbacks through unknown interfaces. Multiverse instead implements runtime lookup tables to handle callbacks [13] 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 $\boxed{\text{C4}}$ 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 [11].

Overlapping instructions are a classic anti-disassembly technique [65] but occasionally appear in highly-optimized libraries too [11]. 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 $\boxed{\text{C5}}$ 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 [32, 1, 5], 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 [9, 1]. 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 [33], control-flow flattening [32], and aliasing [102]. 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 [85, 89, 35, 107]. In the other cases, a dynamic tool can remove dead code and spurious aliases.

3.3 Design

Our design for BinRec 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 Key Considerations for 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 [72]. Which code paths are desirable, however, depends on the type and goal 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 BinRec 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). BinRec can 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. Our technique 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).

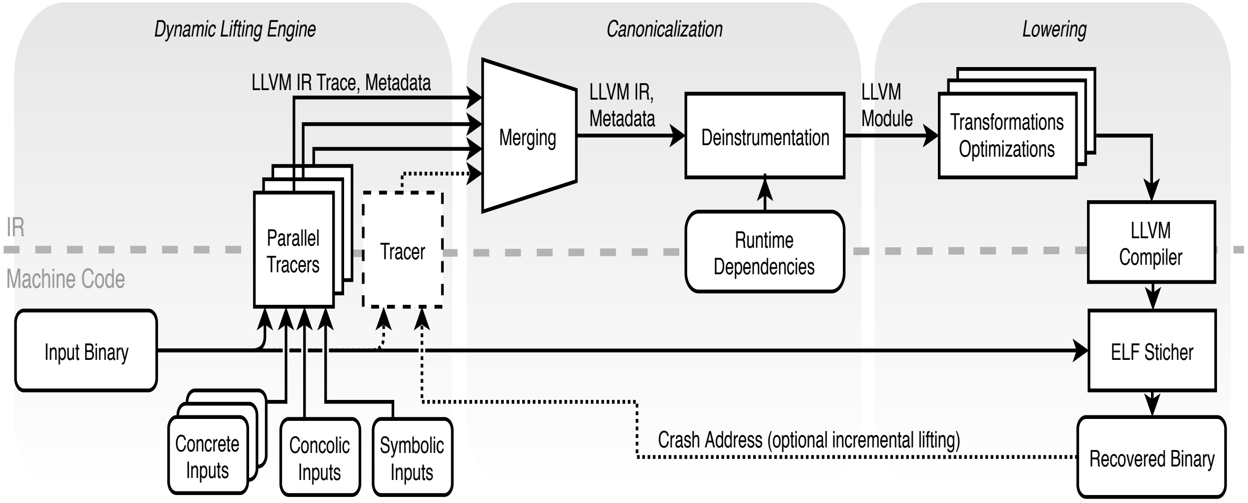


Figure 3.1: The steps of binary recovery: lifting to compiler IR, transformation on the IR, and lowering back to machine code.

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 BinRec such that it can record multiple, independent, traces of the binary (resulting from multiple executions of the binary with different inputs). Our method 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).

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 [29] 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 [47, 109]. 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.

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 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 BinRec 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 $\boxed{\text{C4}}$ and some aspects of obfuscation $\boxed{\text{C5}}$. 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 Our approach 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.

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 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.

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.

Deinstrumentation Our technique 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* 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.

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 C2). 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 [11] and CFI literature [19, 24], and previous work has found even the most precise static analysis overapproximates the set of possible targets.

Library Calls Our approach 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 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 $\boxed{\text{C3}}$ is both sound and performant. Only a dynamic lifting approach can achieve both these properties at once. In the lifting front-end, our approach detects execution of the binary under analysis, and records 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

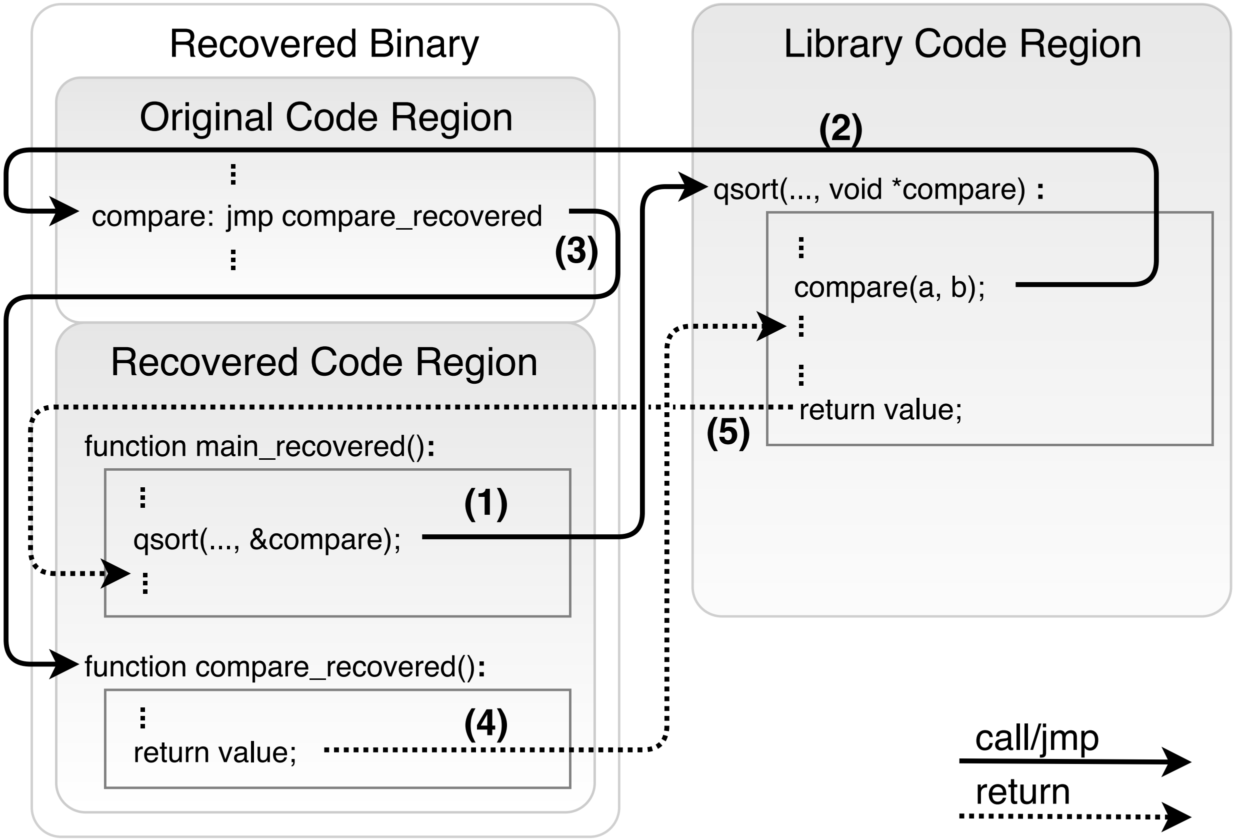


Figure 3.2: The address space of a recovered program that calls the `qsort` library function. Control flows as follows: (1) Call to library with original function pointer; (2) Callback via function pointer; (3) Original function was replaced with jump to recovered code; (4)(5) Returns.

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 3.2 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 [94]. 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. In BinRec, 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

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 Control Flow Miss Handling

Binaries recovered with our approach may encounter unrecovered paths during testing or after deployment due to the coverage limitation of dynamic analysis (see Section 3.3.1). Our approach 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 our method 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^2E [29], a framework that facilitates symbolic execution of a single process running in the QEMU virtual machine [14]. Code is translated to LLVM IR in order to be symbolically executed by the KLEE symbolic executor [23]. S^2E 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 Tracing

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 Library Calls

Recovered IR we get from frontend does not support calling library functions as is. The first reason is that recovered IR only contains the addresses of called library functions from the import table and the import table is not in the recovered IR. The second reason is that registers in the recovered IR are emulated and represented as global variables. However, library code expects input parameters in physical registers and also returns values in physical registers.

Listing 3.1: Excerpt of *eq2.c*: eq2 example in C.

```c

#include <stdio.h>
int main(int argc, char **argv) {
    if(argc == 3){
        char a = argv[1][0];
        char b = argv[2][0];
        if (a == b) {
            puts("leading␣characters␣are␣equal");
        }
        else {
            printf("leading␣chars␣%c␣and␣%c␣are␣not␣equal\n",a,b);
        }
        return 0 ;
    }
}
```

To be able to support library functions in recovered code, first, we annotate addresses that are in PLT as library calls. This information can be collected from frontend during lifting or simply *objdump* utility can be used. Then we insert our own helper functions that will allow transition between recovered code and library code.

Listing 3.2: Empty basic block representing puts function call in recovered IR.

```

BB_8048310:
    store i32 134513424, i32* @PC, align 4
    ret void
```

In order to better understand how we implemented support for calling library functions from recovered IR, we use the eq2 program shown in Listing 3.1. With this example, we also illustrate how we support variadic functions. After we lift eq2 with inputs that cover the whole CFG, the recovered IR has a basic block shown in Listing 3.2 which corresponds to `puts` function call in the original program. This basic block only sets the emulated program counter(PC) to PLT entry of puts function.

Listing 3.2 is where we need to insert our helper function that will call puts function and also provide seamless transition between recovered code and library code. Although we could implement our helper function in LLVM IR, we have implemented it in C to make it easier to understand and implement. Then we compiled it to LLVM IR and linked it with the recovered code.

Listing 3.3: Helper function to support library calls

```

void __attribute__((always_inline)) helper_extern_stub() {
    // return address should be on top of emulated stack,
    // pop it
    addr_t retaddr = __ldl_mmu(R_ESP, 0);
    R_ESP += sizeof (stackword_t);

    // PC should contain the address of the target function,
    // call it
    uint64_t ret = helper_stub_trampoline(R_EDX, R_ECX, R_ESP, PC);

    // copy return value of library function to emulated environment
    R_EAX = ret & 0xffffffff;
    R_EDX = ret >> 32;

    // jump to the return address
    PC = retaddr;
}
```

Listings 3.3 and 3.4 show the helper functions that are used in our implementation. We instrumented the lifted IR to replace the original calls to external library functions with a call to `helper_extern_stub` function.

Function `helper_extern_stub` pops the first word from the emulated stack and stores it as the return address. The first word on the emulated stack is the return address of library call(`puts`) for the emulated program and above it are the input parameters(char*). Then it calls `helper_stub_trampoline`.

Function `helper_stub_trampoline` sets the hardware stack pointer(esp) to point to emulated stack so that library function can get correct input parameters from the emulated stack. Then it makes a call to PLT entry of library function(`puts`). Once the `puts` returns to the caller (`helper_stub_trampoline`), the return value of puts will be in eax hardware register since it is a scalar value. Also, note that `helper_stub_trampoline` handles other types of return values such as float and structures as well. Then `helper_stub_trampoline` restores esp to

Listing 3.4: Helper function to support library calls

```

uint64_t __attribute__((noinline, fastcall)) helper_stub_trampoline(
    const reg_t edx, const reg_t ecx, const reg_t esp,
    const addr_t targetpc) {

    struct { reg_t eax; reg_t edx; } ret;

    // set stack pointer and call library function
    asm (
        // use ebx to hold esp until the function returns
        "movl␣%%esp,␣%%ebx\n\t"
        "movl␣%0,␣%%esp\n\t"
        "calll␣*%1\n\t"
        :: "g"(esp), "r"(targetpc), "c"(ecx), "d"(edx)
        : "ebx", "esp"
    );

    // put fpstt in register to avoid double memory load
    const register unsigned int tmp_fpstt = fpstt;

    // copy return value of library function to emulated environment,
    // return edx/eax by value so that we don't use pointers
    // to emulated registers
    asm (
        "fstpt␣%2\n\t"
        "movl␣%%ebx,␣%%esp"      // restore saved stack pointer
        : "=a"(ret.eax), "=d"(ret.edx), "=m"(fpregs[tmp_fpstt].d)
        :: "eax", "edx", "esp"
    );

    fptags[tmp_fpstt] = 0;

    // return result
    return ((uint64_t)ret.edx << 32) | ret.eax;
}
```

its value before we called puts. As the last step, it returns the return value of `puts` back to `helper_extern_stub` to be stored in emulated return registers. Then `helper_extern_stub` sets the program counter to return address of `puts` call in original binary.

Recovered IR in Listing 3.5 shows recovered IR after we insert our helper function to call puts. Similarly, variadic functions such as `printf` in *eq2* is supported the same way as puts function.

Listing 3.5: Basic block for puts function call after we insert helper function in recovered IR.

```

BB_8048310:
    store i32 134513424, i32* @PC, align 4
    %24 = load i32, i32* @R_ESP, align 4
    %25 = call i32 @__ldl_mmu(i32 %24, i32 0)
    %26 = load i32, i32* @R_ESP, align 4
    %27 = add i32 %26, 4
    store i32 %27, i32* @R_ESP, align 4
    %28 = load i32, i32* @R_EDX, align 4
    %29 = load i32, i32* @R_ECX, align 4
    %30 = load i32, i32* @PC, align 4
    %31 = call x86_fastcallcc i64 @helper_stub_trampoline
                (i32 inreg %28, i32 inreg %29, i32 %27, i32 %30)
    %32 = trunc i64 %31 to i32
    store i32 %32, i32* @R_EAX, align 4
    %33 = lshr i64 %31, 32
    %34 = trunc i64 %33 to i32
    store i32 %34, i32* @R_EDX, align 4
    store i32 %25, i32* @PC, align 4
    ret void
```

So far we handled all the library functions with the same technique. However, we can make improvements to non-variadic functions. We can use the known signatures of library functions to optimize these function calls. Our prototype implementation of BinRec uses signature information to optimize calls to the C library, for example. Listing 3.6 shows the recovered IR after we optimize the `puts` call. However, we can't apply this optimization to variadic functions such as `printf`.

Listing 3.6: Basic block for puts function call after optimization.

```

BB_8048310:
    store i32 134513424, i32* @PC, align 4
    %64 = load i32 , i32* @R_ESP, align 4
    %65 = getelementptr inbounds i8 , i8* @memory, i32 %64
    %66 = bitcast i8* %65 to i32*
    %67 = load i32 , i32* %66
    %68 = load i32 , i32* @R_ESP, align 4
    %69 = add i32 %68, 4
    store i32 %69, i32* @R_ESP, align 4
    %esp1 = load i32 , i32* @R_ESP
    %addr2 = add i32 %esp1 , 0
    %70 = getelementptr inbounds i8 , i8* @memory, i32 %addr2
    %71 = bitcast i8* %70 to i32*
    %arg3 = load i32 , i32* %71
    %72 = call i32 @__stub_puts(i32 %arg3)  //puts call
    store i32 %72, i32* @R_EAX, align 4
    store i32 %67, i32* @PC, align 4
    %pc6 = load i32 , i32* @PC
```

3.4.3 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 [40]. 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 3.3 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. Recovered 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. For implementation details see Appendix A.

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 with our approach 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 [42, 8, 13], 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.

Listing 3.7: Excerpt of *decompress.c*: libjpeg example in C.

```
void callback_func(j_common_ptr cinfo) {
  printf(”.”);
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; //jpeg info
    struct jpeg_progress_mgr  progress;
    ...
    //After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_counter = 0L;

    info.progress = &progress;
    jpeg_start_decompress( &info );

    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
```

3.5.1 Comparison with static lifters

Our technique 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 our approach to McSema [43] and Rev.ng [42], popular state-of-the-art binary lifting frameworks.[1] We limit our comparative study to active, open source binary lifters which, like BinRec, aim to be compiler-agnostic.

We found McSema [43] 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

[1] Code snapshot on July 25th, 2019

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 C1 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 3.7: 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 [42]. While we managed to get some of the binaries running by reducing the optimization level to O0 (a classic example of C4—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 our technique to Rev.ng using the most recent published results [49]. 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.

[2] The error message indicated failed assertions in the IsolateFunctionsImpl class upon replacement of indirect branch targets, strongly hinting towards an instance of C2. We contacted the developers but did not get any detailed feedback in time for the submission.

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 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.

Table 3.1: Measured execution time normalized to the original binaries. Rev.ng results are reported from publication [49].

	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

3.5.2 Performance

Table 3.1 presents the performance of binaries lifted with our approach. For every input program we compiled both optimized (O3) and unoptimized (O0) binaries which produce correct output in the test cases. Our results show that there is a potential for performance improvement by using our technique as a post-release optimizer—particularly, if the original was not optimized at the source level. With our approach, six benchmarks – *bzip2*, *mcf*, *hmmer*, *sjeng*, *h264ref*, and *astar* – run faster than the unoptimized client binaries. In some cases, our technique 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 3.4.3 and show their benefit in Figure 3.3. 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 recovered binaries compared to unoptimized and optimized input binaries is 0.98x and 1.29x, respectively.

3.5.3 Code Coverage

Figure 3.4 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

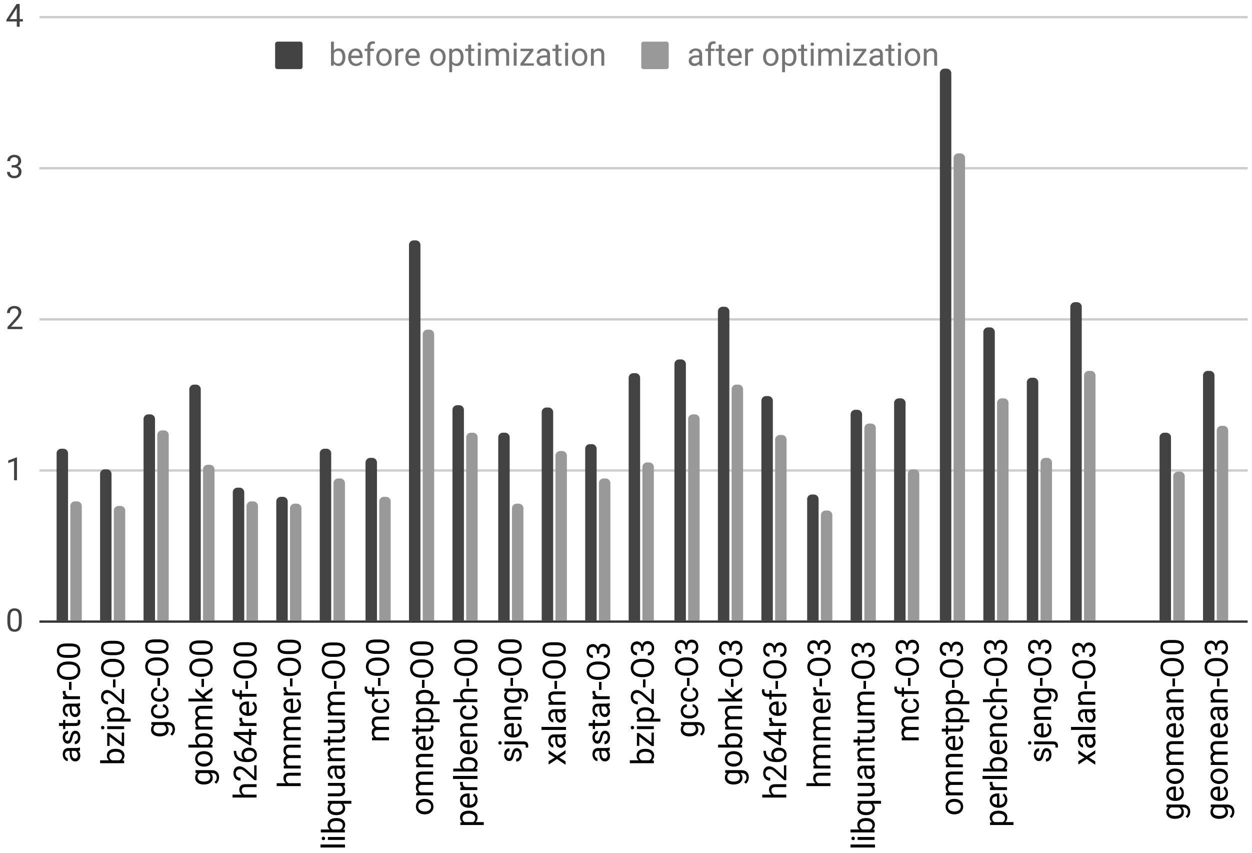


Figure 3.3: Execution time improvement from CPU state variable de-aliasing and global variable promotion.

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.

Incremental Lifting To show the effectiveness of incremental lifting, we conducted an experiment with *bzip2* as illustrated in Figure 3.5. 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 3.5 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.

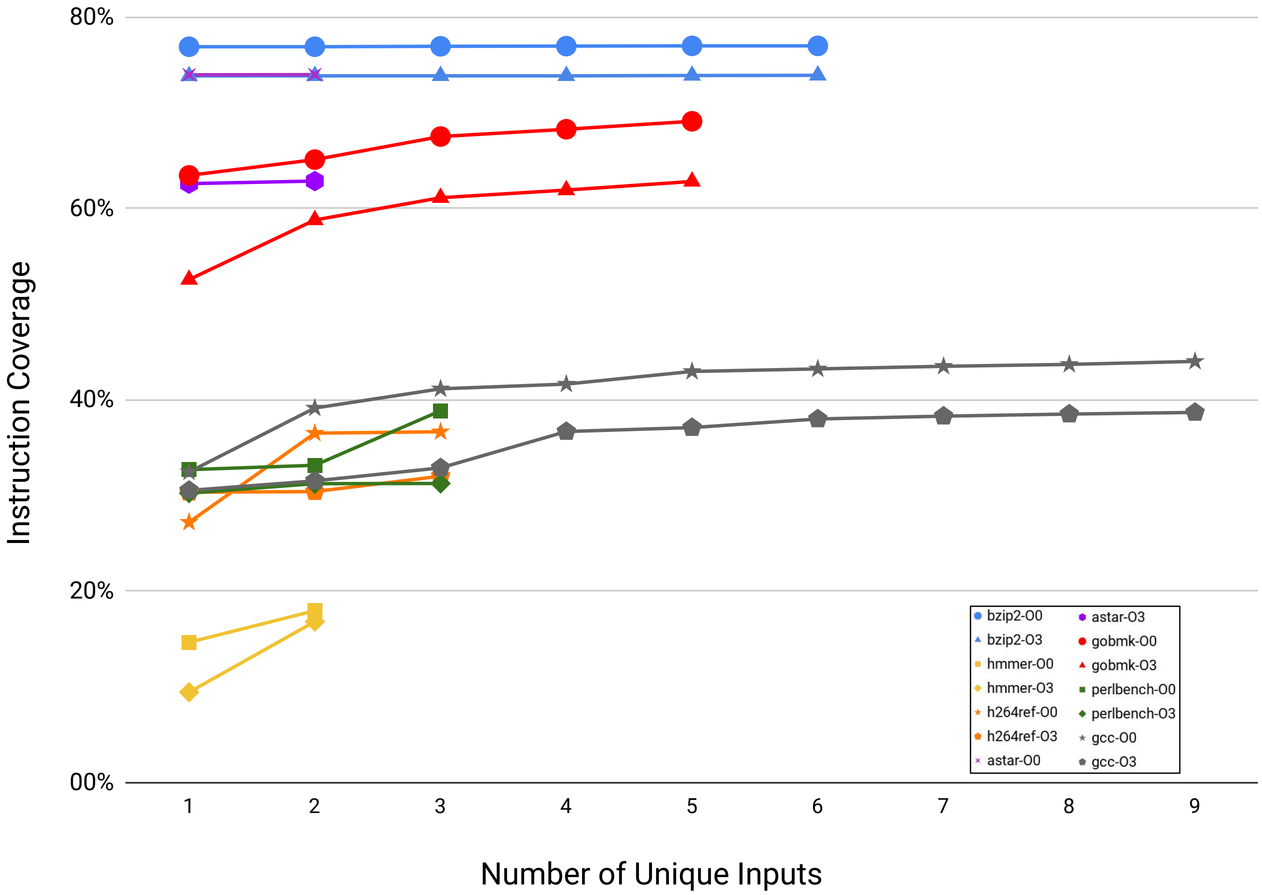


Figure 3.4: Coverage with respect to the original binaries. The input set is the *ref* workload of SPEC CPU2006.

3.5.4 Lifting 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^2E [29] as its tracing frontend, we also present the time to execute those workloads without instrumentation in S^2E. The lifting time of static lifting toolchains, such as McSema,

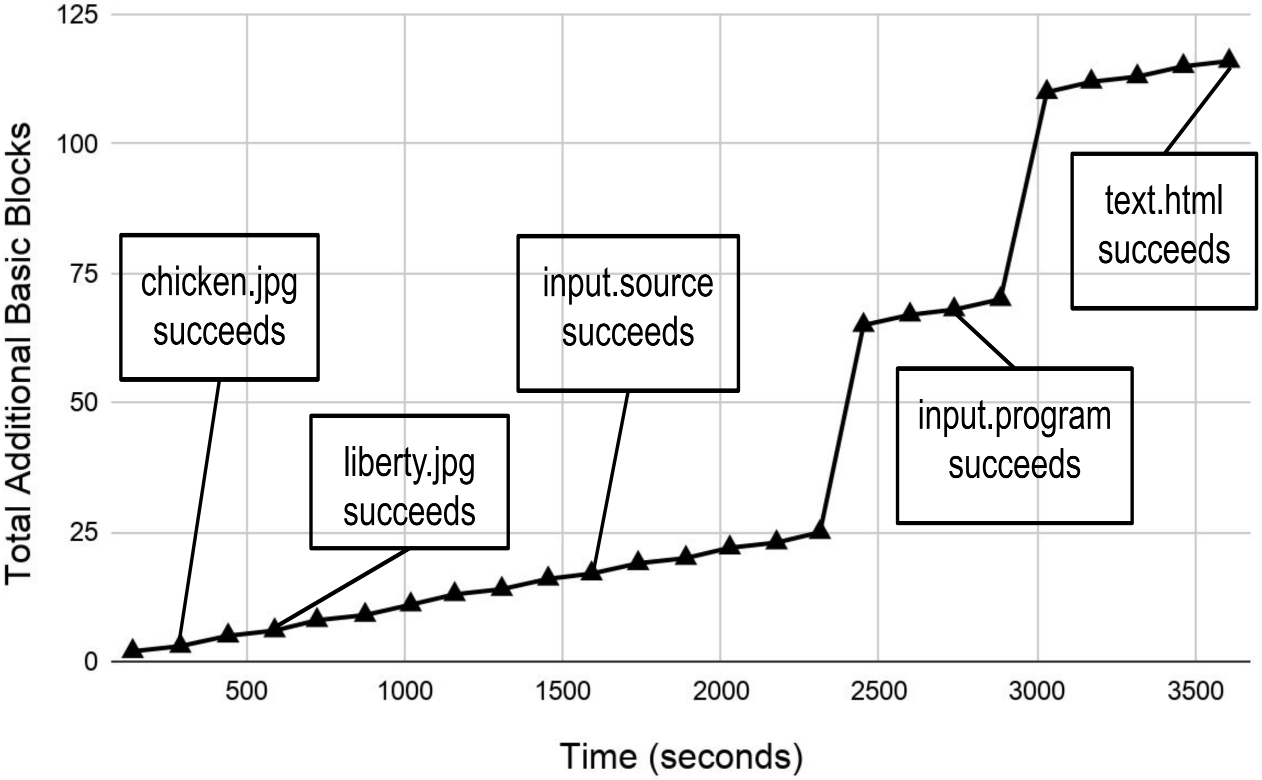


Figure 3.5: Incremental lifting progression of *bzip2*.

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.

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

3.6 Applications

One of the main goals of our approach is to enable complex transformations on real-world program binaries. Since our approach 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 recovered binaries. 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 our approach 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.

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 [111].

	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

3.6.1 Control-flow Hijacking Mitigation

Even without additional compiler-based transformations, our approach has an endogenous ability to mitigate memory-corruption vulnerabilities in the original program. A recovered program emulates 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. Our technique does not natively mitigate data-only attacks [57], though they may be mitigated using additional transformation on the IR.

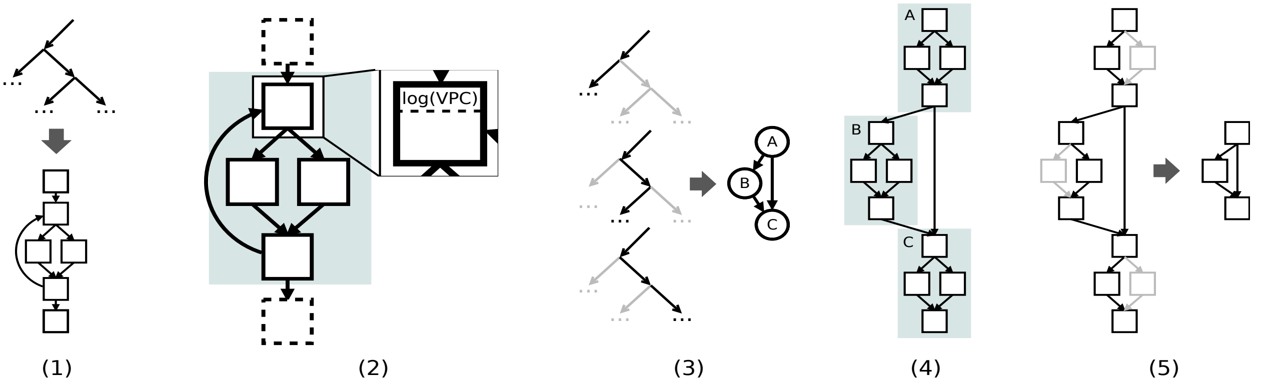


Figure 3.6: Our deobfuscation approach. (1) We lift the binary using symbolic execution or high-coverage inputs. (2) We identify the lifted interpreter loop and instrument it to log the virtual program counter (VPC) at the entry. (3) The instrumented binary is exercised for all uncovered code paths, 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.

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 known as context-insensitive control-flow integrity on forward and backward edges [19].

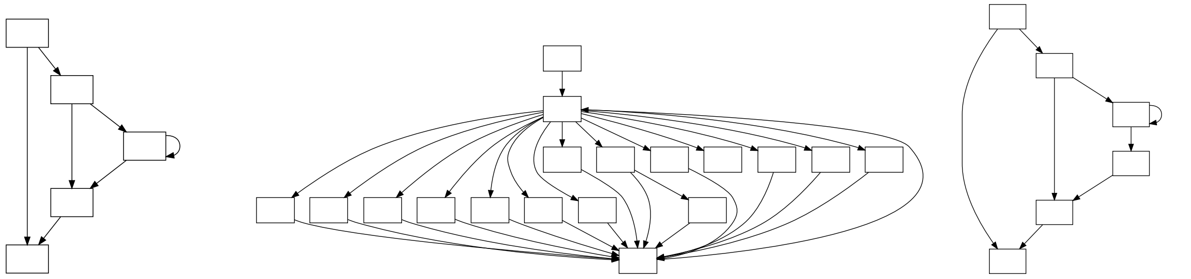
Consider an original binary with a vulnerable stack buffer overflow using an unsanitized `strcpy` call, that can be used to overwrite a return address. In the recovered program, that buffer is located in a `@memory` array which emulates the memory of the original program. The `strcpy` call will proceed 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 [24], 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 3.5 for easier comparison with other approaches. Our results show that our approach 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 [111], 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 our approach.

3.6.2 Virtualization-deobfuscation

We used BinRec to lift programs obfuscated by virtualization (cf., Section 3.2.5). Figure 3.6 illustrates our deobfuscation approach. For this use case, we detect the Virtual Program Counter (VPC) and virtual interpreter loop through known techniques [89]. 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 3.7 depicts the `fib` program, showing its control flow graph obfuscation and subsequent deobfuscation.


(a) Original

(b) Obfuscated

(c) Deobfuscated

Figure 3.7: 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 [87]. 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 recovered binaries, 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 [44], we note that this required a re-implementation of both the analysis and instrumentation passes—a substantial disadvantage in maintainability compared to our approach. 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 [61]. 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, our approach 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 our approach 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 4.2).

3.6.5 Attack Surface Reduction

As a software program evolves over time, developers often introduce additional features to address various user expectations and to improve compatibility with other software or hardware. Many of these additional features remain unused by regular users, however, and security vulnerabilities in them often remain undiscovered for years. In a recent study, Wagner et al. found that 83% of security vulnerabilities that were assigned a CVE number in 2014 laid in "cold code" [101]. To discover bugs lurking in cold code, one current practice is to insert sanity checks into a program using dynamic bug detection tools (e.g., sanitizers), and to fuzz the program or run test cases to exercise all parts of the program. As program complexity grows, these approaches are less suitable for detecting bugs hidden in deep execution paths.

An alternative approach to eliminate latent bugs is to specialize the program for specific use cases through manual feature pruning [64], compile-time specialization [60], or link-time code compaction [25]. Assisted by either static or dynamic analysis to determine which features are unnecessary for the target use case, these techniques can achieve substantial reductions of the program's code size, while also removing the features that are the most likely to contain security vulnerabilities. The downside of these techniques is that they are heavily tailored to the kernel, exploiting the fact that the kernel has a small set of easily recognizable interfaces to external code (i.e., user-space programs, peripheral devices, etc.), and thousands of preprocessor options to enable or disable features at the source code level.

We propose a more generic approach for attack surface reduction that works for binary commercial off-the-shelf (COTS) programs. Instead of trying to find vulnerabilities in cold code, or running coarse-grained analyses to identify unnecessary features and remove them at compile time, we use fine-grained dynamic profiling to determine with greater precision which parts of a program's code are actually used. We then lift these parts of the code into a compiler intermediate representation (IR) format, and use a mainstream compiler to compile the IR to a new binary executable.

Although progress is being made at quantifying the attack surface of the kernel [60], we are not aware of any universally accepted attack surface reduction metrics for user-space code. We therefore chose to measure the attack surface of the recovered programs by calculating the fraction of original program instructions that are lifted to IR code, and by comparing the number of ROP gadgets in the recovered code with the number of gadgets in the original code. Table 3.5 shows our findings. We evaluated only benchmarks that can be correctly recovered by BinRec; all input binaries are optimized (O3). Each program binary is lifted using all reference inputs available in the SPEC CPU 2006 benchmark suite. We then

Table 3.5: Attack surface reduction

Benchmark	% recovered instructions	ROP gadgets		
		# original	# recovered (gmean)	% reduction
astar	49.09%	1029	804	21.87%
bzip	55.91%	1070	581.67	45.64%
gobmk	19.70%	20564	4583.6	77.71%
h264ref	21.81%	9035	2315.67	74.37%
hmmer	8.17%	5488	1100	79.96%
libquantum	26.92%	1397	179	48.53%
mcf	57.20%	549	433	21.13%
sjeng	25.84%	2269	1013	55.35%
gmean	28.05%			47.56%

measured geometric means of results from different reference inputs. *astar*, for example, has two reference inputs *BigRakes2048* and *rivers*, thereby two recovered binaries were generated and measured for this benchmark. With our approach, only 28% of the original instructions are lifted to LLVM IR on average. We measured ROP gadget reduction in the recovered binaries using a ROP gadget finder tool called Ropper [3]. We found that the recovered binaries contain 48% fewer ROP gadgets than the original binaries.

Chapter 4

Discussion and Future Work

While there are many challenges that Binrec's design is uniquely able to solve through its novel, dynamic lifting approach, there are certain problems we encountered that remain unsolved. In this section, we will explain some of these problems in detail and also lay out how they could be tackled in the future.

4.1 Lessons Learned

Our framework uses S^2E as a front end to have flexibility of driving the execution in a number of ways. However, collecting traces with S^2E is very slow compared to other dynamic analysis frameworks such as Pin because S^2E emulates the full system. On top of emulation, we generate LLVM IR in the frontend as well and transfer it to the host system. This causes more slowdown and complexity. If symbolic execution is not desired, using a faster front end for dynamic analysis and generating IR after the analysis is over would probably increase our development speed.

Recovered binaries have worse runtime when target binaries are optimized. We observed that there are 2 main reasons for the performance overhead. The first reason is that S^2E generates IR in which floating point instructions are emulated. Inlining the float emulation functions helps to reduce the performance overhead of floating functions but using a front end that doesn't emulate the floating point instructions would be better design choice.

The precision of the analysis performed on the lifted IR can be greatly improved if the global emulated stack is split into individual stack frames and if the variables in the emulated stack are promoted to represent the original semantic representation of these variables in the source of the program. This latter process is termed symbolization of stack variables. In our framework, we worked on symbolization of stack variables with no heuristics by static analysis of lifted traces. We explain our approach to symbolization of stack variables, its limitations in section 4.2 and how we can overcome these limitations by employing dynamic methods in the future.

4.2 Stack Symbolization with Static Analysis

Programs during execution store their function local variables, register spills, return addresses and functions arguments in an *explicit* stack memory region. Semantics of source code are analyzed by the compiler to produce assembly instructions which access the explicit stack memory region. While source code refers to local variables and function arguments with their symbol names, assembly instructions access local variables and function arguments with their addresses on the stack.

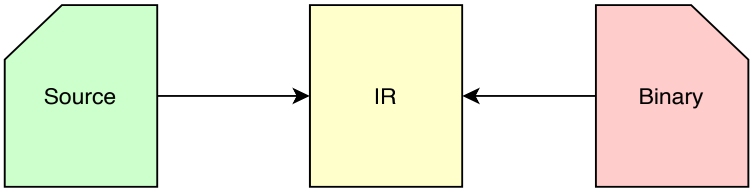


Figure 4.1: Ideal lifting

Listing 4.1: Example that illustrates simplified lifted IR in pseudo code.

```

int main(){
    int a = sum(1, 2);
    return a;
}

int sum(int a, int b){
    int c = a + b;
    return c;
}


// Lifted IR pseudo

int stack[STACK_SIZE];
R_ESP = &stack[STACK_SIZE - 1];

int recovered_main(){
    R_ESP = R_ESP-3; // allocate space for 2 args and 1 local variable
    R_ESP[1] = 2; // arg2
    R_ESP[0] = 1; // arg1
    int tmp = recovered_sum();
    R_ESP[2] = tmp; // redundant store for illustration
    return tmp;
}

int recovered_sum(){
    R_ESP--;
    R_ESP[0] = *R_EBP; //save base pointer
    R_EBP = R_ESP;
    R_ESP--; // allocate space for c
    int tmp1 = R_EBP[0];
    int tmp2 = R_EBP[1];
    int tmp3 = tmp1 + tmp2;
    R_ESP = tmp3;  // store result to c
    R_ESP++; // deallocate c
    R_EBP = R_ESP[0];
    R_ESP++;
    return tmp3;
}
```

To be able to fully apply advanced compiler transformations with the purpose of optimization and hardening, we need to lift the binaries to IR that is similar to when we generate IR from source as shown in figure 4.1. In its initial form, however, lifted IR resembles disassembly of binary as shown in listing 4.1. The lifted code is essentially an emulator for the original program. The explicit stack of the original program is represented as a *global array* and we refer to it as *emulated* stack in lifted IR. This emulated stack grows and shrinks with accordance to the execution of the program to represent the behavior of the original explicit stack.

This causes a number of limitations. One of the limitations is that we cannot change the layout of the stack. Hardening transformations often require change on the layout of the stack. For example, stack canaries hardening techniques insert a secret value on stack frames and checks this value before returning to the caller. If the secret value has changed, the program aborts to stop a potential attack [4].

Another limitation of having an emulated stack in lifted IR is that it hinders dataflow analysis. When an instruction accesses the emulated stack with an offset computed at runtime, the compiler cannot infer which offset the instruction accesses. In that case, all the disjoint dataflows in the original program interfere in the lifted program as they access this shared global array which represents the emulated stack region.

If the memory locations in the emulated stack are promoted to represent the original semantic representation of the variables in the source of the program, compiler analysis and transformations on the lifted IR can be significantly improved. This process is termed symbolization of stack variables. In our framework, we worked on symbolization of stack variables with no heuristics by static analysis of lifted traces. We present our approach to symbolization of stack variables and its limitations.

One of the challenges we faced is the detection of function arguments in static disassembly.

Listing 4.2: Example 4.1 after function argument symbolization

```

int stack[STACK_SIZE];
R_ESP = &stack[STACK_SIZE - 1];

int recovered_main(){
    R_ESP = R_ESP-1; // allocate space for 1 local variable
    int tmp = recovered_sum(1, 2);
    R_ESP[0] = tmp; // redundant store for illustration
    return tmp;
}

int recovered_sum(int a, int b){
    R_ESP--;
    R_ESP[0] = *R_EBP; //save base pointer
    R_EBP = R_ESP;
    R_ESP--; // allocate space for c
    int tmp3 = a + b;
    R_ESP = tmp3;   // store result to c
    R_ESP++; // deallocate c
    R_EBP = R_ESP[0];
    R_ESP++;
    return tmp3;
}
```

Listing 4.3: Challenging example for function argument symbolization.

```

void func(int a, int b){
    int* aPtr = nullptr;
    aPtr = &a;
    ...
    temp = b;
    ...
}

func:
    subl $32, %esp   // alloc 36 byte
    lea 36(%esp), 24(%esp)   // move &a to aPtr
    ...
    movl 24(%esp), %edx   // move aPtr to %edx
    movl 4(%edx), %ecx   // move b to %ecx
```

The calling convention of the target architecture determines how the arguments are passed to functions. In our context, it is the ABI of the target architecture to which the original program was compiled for. For example, function arguments are passed via the stack memory on x86_32 platform. In listing 4.1 *recovered_sum* accesses input arguments on line 31 and 32. If we can symbolize these arguments, we can remove them from the emulated stack and pass them with a function call as shown in listing 4.2.

While it is possible to identify the arguments of functions for simple cases using static analysis, it is not always possible. Listing 4.3 shows an example code where the first argument *a* can be discovered with static analysis, while the second argument *b* cannot. In the assembly code of the example, address of *a* is first stored to local variable *24(%esp)* on line 12. After many instructions, it is also stored to temporary register *%edx*. In the last instruction, *b* is stored to temporary register *%ecx*. However, through static analysis, reliably resolving that *4(%edx)* corresponds argument *b* is a very challenging problem because some intermediate computations could modify *4(%edx)* on line 13.

The second challenge we faced is resolving stack frame size. If we resolve stack frame sizes, we can divide the emulated stack in listing 4.3 for each function as illustrated in listing 4.4. This will allow each function to access a disjoint emulated stack memory region instead of a shared global emulated stack. Successful implementation of such a solution, we believe, will improve data-flow and alias analysis. In the example shown, splitting the emulated stack will yield a stack frame size of 4 bytes for *recovered_main* and 8 bytes for *recovered_sum*.

While it is straightforward to recognize stack size of each function in listing 4.3 with static analysis, it is not always possible. Especially, when stack pointer is decremented by a value loaded from memory instead of a program constant. Another such example is when external library functions use the emulated stack instead of the hardware stack. In this case, external library functions expand the emulated stack. External library functions use the emulated stack when the API is unknown or the external library function is a variadic function such

Listing 4.4: Example 4.2 after dividing the global emulated stack

```c

int recovered_main(){
    int stack[1];
    R_ESP1 = &stack[1];
    R_ESP1 = R_ESP1-1; // allocate space for 1 local variable
    int tmp = recovered_sum(1, 2);
    R_ESP1[0] = tmp; // redundant store for illustration
    return tmp;
}

int recovered_sum(int a, int b){
    int stack[2];
    R_ESP2 = &stack[2];
    R_ESP2--;
    R_ESP2[0] = *R_EBP; //save base pointer
    R_EBP = R_ESP2;
    R_ESP2--; // allocate space for c
    int tmp3 = a + b;
    R_ESP2 = tmp3;  // store result to c
    R_ESP2++; // deallocate c
    R_EBP = R_ESP[0];
    R_ESP2++;
    return tmp3;
}
```

as *printf* (see 3.4.2). One of the ways we believe we could address this issue is by making the local stack large enough to have sufficient space for stack frames of the external library functions. However, this would have a negative effect on runtime of the recovered binary; the stack space that a function would require is runtime dependent and hard to determine.

Listing 4.5: Example 4.4 after symbolizing stack variables

```

int recovered_main(){
    tmp = alloca i32, 4
    tmp = recovered_sum(1, 2);
    return tmp;
}

int recovered_sum(int a, int b){
    tmp = alloca i32, 4
    tmp = a + b;
    return tmp;
}
```

The next challenge in symbolization is to determine local variables allocated within each stack frame. For this we must break the function stack frames into individual memory ranges and assign each region with a symbol. Then we must replace references to individual memory ranges with that of the newly created symbols from the previous step. In other words, we would like to transform listing 4.4 to listing 4.5. While it might be possible to divide the stack frame into memory ranges, it is impossible to update references to these memory ranges with symbols in static analysis. Even a single indirect reference would break the static analysis because as its target in memory cannot be determined statically. Therefore, not all the stack frames can be symbolized into individual variables with static analysis.

4.3 Future Work

In the implementation we built as part of this project and that is evaluated here, we keep the emulated stack as is. This definitely hinders the compiler optimizations and also security techniques that we can apply. For stack symbolization and self-modifying code, however, we outline some ideas for future work to potentially tackle those open problems leveraging traces.

Generating the most optimal recovered binaries and being able to apply any compiler transformation including the hardening techniques would bring the biggest improvement to our current approach. This is possible only if we can symbolize the emulated stack. We have shown that there are a number of challenges we have faced when we tried to symbolize the emulated stack with static analysis. However, we believe that dynamic analysis has a potential to address some of the limitations of static analysis for the purpose of symbolizing the emulated stack. For example, given that we were able to break a stack frame into an array *A* and a variable *i*, we can tell if an indirect reference accesses A or i by analysing memory accesses at runtime. If our solution captured a trace, in which the indirect access observed in that trace was to A[5], we can safely assume that the indirect access will always reference to A under the assumption that the program has no memory corruption. However, this wouldn't hold if any external sources of state are used in the computation of the indirect access as shown in listing 4.6

Listing 4.6: Indeterministic code example

```

int main(){
    int A[4];
    int i;

    int rand = get_random_number();
    int add_a = (unsigned long)A;
    int add_i = (unsigned long)&i;
    void *_a = (void *)((rand%2)*add_a + ((rand+1)%2)*add_i);
    *_a = 17; // A or i ?
}
```

Another item we have to leave for future work is self-modifying code. Self-modifying code rewrites the code at some program counter with new code. In our current implementation, we keep a cache of the executed basic blocks with their start address as a key to prevent IR generation for the same code. Therefore, the current implementation does not support self-modifying code. To support it, we would need to add 'version labels' to each start address. When execution reaches an adress that was visited before, we can assign a new version number if the code is different from the previous versions. As a result, we can identify basic blocks using both start address and version number. This would take some additional lifting time and add more complexity while merging traces into one CFG. We don't think that this is a limitation of our approach.

Chapter 5

Related Work

5.1 Low-level binary analysis and rewriting

Many projects target the problem of low-level binary analysis and rewriting. PEBIL [62], UQBT [30] and Uroboros [104] 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 [92] supports static and dynamic analysis techniques, including symbolic execution, but does not target code rewriting (unlike BinRec). Earlier work such as ATOM [45], PLTO [86], Diablo [81], and Vulcan [95] 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. [13] 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 [108] 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. Our approach 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.

5.2 Code transformations using dynamic traces

Dynamic instrumentation tools such as PIN [68], Dyninst [17], DynamoRIO [2] and Valgrind [76] 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 [77], and require specific assembly-level transformation passes for each application, whereas our approach leverages existing techniques present in production compilers.

Just-in-time (JIT) compilers such as V8 [48] and SpiderMonkey [73] collect dynamic traces to determine which code to optimize and to speculate dynamic data types. Sulong [83] is a frontend for the Graal compiler that effectively creates an LLVM bitcode execution engine. Similar to our approach, 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.

5.3 Binary code lifting

LLBT [90, 91] statically retargets binaries to different ISAs after lifting them to LLVM IR. McSema [43], Dagger [15], Rev.ng [42] and RevNIC [28] (based on S^2E) and SecondWrite [8] lift machine code for the purpose of high-level static binary translation on LLVM IR.

HQEMU [55] 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.

5.4 Virtualization-deobfuscation

Current deobfuscation approaches follow two general directions. Rolles [85] and Sharif et al. [89] assume the presence of an interpreter and (respectively manually or automatically) identify the virtual program counter variable in memory, using this to find the interpreter loop and subsequently reconstruct bytecode control flow. Although Sharif et al. show that this can be done automatically for state-of-the-art obfuscators, even producing a CFG of uncovered bytecode, semantics of bytecode instructions cannot be determined automatically by existing work. Rolles assigns semantics by manually analyzing dispatch handlers to determine the type of implemented instructions.

Coogan et al. [35] and Yadegari et al. [107] instead do not make any assumptions about the type of obfuscation being used, or even if any obfuscation is used at all. They instead reason about the semantics of code by looking at observable behaviour: system calls with which the obfuscated program interacts with the operating system. The general idea is to apply semantic-preserving transformations on the code in such a way that the resulting CFG approximates that of the bytecode. Although these approaches have the advantage of including semantics, they suffer from imprecision: the resemblance of the deobfuscated code to the original program code is only as high as can be approximated from the observable behaviour.

5.5 Sanitizers

We applied AddressSanitizer that is part of the LLVM compiler framework to recovered binaries. There are many other sanitizers. MemorySanitizer detects uninitialized reads. It consists of a compiler instrumentation module and a run-time library. Although we did not include it in our evaluation, we could apply it to recovered programs as we applied AddressSanitizer.

UndefinedBehaviorSanitizer(UBSan) [66] detects undefined behaviors. UBSan modifies the program at compile-time to catch different kinds of undefined behavior during program execution. Using a misaligned or null pointer, signed integer overflow, and conversion between floating-point types that would overflow the destination are some of the undefined behaviors UBSan can catch.

ThreadSanitizer is another llvm tool that detects data races, deadlocks, and misuses of thread synchronization primitives (e.g., pthread mutexes) in multi-threaded programs by instrumenting memory accesses and atomic operations. deadlocks, and misuses of thread synchronization primitives in multi-threaded programs [88]. However, our current approach doesn't handle multi-threaded programs. Therefore it is not applicable to recovered binaries.

There are also number of sanitizers that detect type confusion bugs in C++ programs by detecting downcasts of a base class pointer into an illegally derived class pointer. Some of these type confusion sanitizers are CaVer [63], TypeSan [51], and HexType [58]. Since our current technique doesn't symbolize the stack and has no type info for memory locations, these sanitizers are not applicable to recovered binaries.

5.6 Control Data Attack Mitigations

Control data defenses are powerful at stopping code reuse attacks. Broadly they can be categorized into randomization and enforcement based defenses. Randomization based approaches provide probabilistic protection against attacks by randomizing low level details of a program, while enforcement mechanisms enforce a security policy.

Control flow integrity (CFI) is one of the examples of enforcement based mitigation proposed by Abadi et al. in 2005 [6]. We have showed that recovered programs already come with CFI because of the nature of dynamic lifting. At a high level, CFI restricts the control-flow of a program to valid execution traces by inserting a check at every indirect control flow transfer to make sure the target is valid.

Indirect control flow transfers can be divided as ”forward” and ”backward” edges. The forward edges are indirect calls and jumps. The backward edges are return instructions. Integrity of the forward edges can be enforced by checking if the target is in the set of allowed targets. The backward edges must be paired with the most recently called function and they can be verified by a shadow stack. The shadow stack is used to store valid return addresses and it is allocated in a protected memory region separate from the program's runtime stack. Before a function returns to caller, the return address on program's runtime stack is compared with the return address on the shadow stack. Several different techniques have been proposed to implement the shadow stack and compute the set of allowed targets [20].

There are number of efforts to improve original CFI work. Some of the efforts to improve CFI include applying CFI using binary rewriting [100, 98], adapt CFI to protect C++ virtual dispatch [110, 50, 97], integrate CFI with fine-grained randomization [79], and enforce CFI using cryptography [70]. CFI research has also find adoption in open source community. LLVM and gcc, two of the most popular open source compilers, include CFI implementations [97].

Another policy similar to CFI is object type integrity (OTI) [21]. While CFI verifies the targets of indirect control flow, OTI protects object identities. It is proposed to stop attacks that corrupt virtual table pointers. OTI can stop these attacks by enforcing that the vtable pointer written to the object in the constructor function matches the one used for dynamic dispatch.

Code-pointer integrity (CPI) and code-pointer separation (CPS) are another set of policies that can prevent code reuse attacks [61]. They check the integrity of code pointers in the program. CPI ensures that all the code pointers including pointers that can be used to reach a code pointer cannot be modified by an attacker. CPS is less stricter version of CPI to reduce run-time overheads. It only prevents modifying the code pointers directly and leaves the pointers that may refer to the other code pointers unprotected.

Randomization based defenses are an alternative to enforcement-based defenses. Randomization based defenses add randomness to the low-level details of the program without changing the semantics of the program. The most widely deployed randomization based defense is address space layout randomization (ASLR) [78]. When ASLR is enabled, the base address of different memory regions in the program are chosen randomly every time the program is loaded on memory. This would shift gadgets in executable code and make their location unpredictable, therefore, prevent code reuse attacks. However, if a single pointer to a known memory region gets leaked, attackers can use this information to derandomize entire memory region.

There are number of techniques to add more fine-grained randomization to the programs.

One such technique is random insertion of NOP instructions throughout the program's code [54]. Another technique is to randomize the location of every instruction [52]. Just-in-time compilers can also apply fine-grained randomization techniques [53].

However, attackers can still perform code reuse attack if they can disclose the code loaded on memory. There have been several solutions proposed to address this weakness. One of the solutions is Isomeron that provides randomization strategy resilient to code disclosure [38]. Another solution is enforcing execute only memory for code. The memory used to hold executable code is generally readable and if the read permission is removed, the attacker cannot leak the code. One of the systems that employs execute only code is Readactor [37]. There have also been efforts to enable execute-only code for legacy binaries [26] and on systems without memory management units [16].

Chapter 6

Conclusion

Programs written in unsafe languages like C and C++ are prone to memory corruption errors. These memory errors can be exploited by attackers and the attackers can take complete control of a vulnerable program. Extensive existing research presented many bug finding techniques and mitigations for programs written in unsafe languages. However, most of the existing work is for programs for which source code exists. Software is as safe as its weakest link and in our software ecosystem, we have several components working together. Some of these components do not have source code available and it is very important to protect these components as well.

In this dissertation, we presented BinRec, a new solution for binary lifting based on dynamic analysis. Our approach lifts a program to compiler-level intermediate code for ease of analysis, while ensuring that it can still compile the resulting IR. Compared to existing static analysis-based techniques, our approach can seamlessly handle indirect control flow transfers, handwritten assembly, and code obfuscation. Our approach tries to overcome the coverage issue of dynamic analysis by using trace merging and incremental recovery. We demonstrate the powerful applications made possible by our approach: recovering program semantics of virtualization-obfuscated binaries, attack surface reduction in the recovered binary, and applying compiler-level optimizations and hardening transformations to stripped binaries.

Bibliography

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

[2] DynamoRIO. https://dynamorio.org.

[3] Ropper. https://scoding.de/ropper/.

[4] StackProtector. http://www.llvm.org/doxygen/StackProtector_8h_source.html.

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

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[7] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th USENIX Security Symposium*, SSYM '09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

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

[18] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 183–198, New York, NY, USA, 2011. Association for Computing Machinery.

[19] 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.

[20] 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.*, 50(1), Apr. 2017.

[21] N. Burow, D. McKee, S. A. Carr, and M. Payer. Cfixx: Object type integrity for c++. In *NDSS*, 2018.

[22] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

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

[24] 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.

[25] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the linux kernel. *ACM SIGPLAN Notices*, 2005.

[26] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen. Norax: Enabling execute-only memory for cots binaries on aarch64. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 304–319, 2017.

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

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

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

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

[31] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 1st edition, 2009.

[32] 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.

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

[34] D. D. I. F. Committee. Dwarf debugging information format version 4. `http://www.dwarfstd.org/doc/DWARF4.pdf`.

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

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

[37] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780, 2015.

[38] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

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

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

[41] 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.

[42] 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.

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

[44] S. Dinesh, N. Burow, D. Xu, and M. Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *S&P*, 2020.

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

[46] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

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

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

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

[50] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, page 341–350, New York, NY, USA, 2015. Association for Computing Machinery.

[51] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. ACM.

[52] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585, 2012.

[53] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *CCS '13*, 2013.

[54] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[55] 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.

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

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

[58] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387. ACM, 2017.

[59] A. Knowledge, L. M. Society, C. Barrett, D. Kroening, and T. Melham. Problem solving for the 21st century efficient solvers for satisfiability modulo theories.

[60] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *NDSS*, 2013.

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

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

[63] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium*, SSYM '15, pages 81–96, Austin, TX, 2015. USENIX Association.

[64] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee. An application-oriented linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 2004.

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

[66] LLVM Developers. Undefined behavior sanitizer.
`https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, 2017.

[67] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[68] 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.

[69] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In *CCS*, 2015.

[70] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery.

[71] A. Milburn, H. Bos, and C. Giuffrida. SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)(San Diego, CA*, 2017.

[72] 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.

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

[74] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for c. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[75] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

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

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

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

[79] PaX. Microsoft equation editor.
`https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html`, 2018.

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

[81] 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.

[82] 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.

[83] 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.

[84] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. 15, 2012.

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

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

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

[88] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *2009 Workshop on Binary Instrumentation and Applications*, WBIA'09, pages 62–71, New York, NY, 2009. ACM.

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

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

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

[92] 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.

[93] M. S. Simpson and R. K. Barua. MemSafe: Ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

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

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

[96] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[97] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 941–955, USA, 2014. USENIX Association.

[98] 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.

[99] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In D. Balzarotti, S. J. Stolfo, and M. Cova, editors, *Research in Attacks, Intrusions, and Defenses*, pages 86–106, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[100] 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.

[101] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.

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

[103] 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.

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

[105] 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.

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

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

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

[109] 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.

[110] C. Zhang, S. A. Carr, Y. Ding, and D. Song. Dr af t vtrust : Regaining trust on virtual calls. 2015.

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

Appendix A

Stack Unwinding Optimization

A.1 Musl libc's setjmp and longjmp Implementation for x86_32

Listing A.1: Source of *setjmp.s*

```

.global ___setjmp
.hidden ___setjmp
.global __setjmp
.global _setjmp
.global setjmp
.type __setjmp,@function
.type _setjmp,@function
.type setjmp,@function
___setjmp:
__setjmp:
_setjmp:
setjmp:
        mov 4(%esp), %eax
        mov    %ebx, (%eax)
        mov    %esi, 4(%eax)
        mov    %edi, 8(%eax)
        mov    %ebp, 12(%eax)
        lea 4(%esp), %ecx
        mov    %ecx, 16(%eax)
        mov  (%esp), %ecx
        mov    %ecx, 20(%eax)
        xor    %eax, %eax
        ret
```

Listing A.2: Source of Binrec's setjmp

```
void __attribute__((always_inline)) nonlib_setjmp(){

    addr_t retaddr = __ldl_mmu(R_ESP, 0);
    R_ESP += sizeof (stackword_t);

    reg_t *r_esp = (reg_t *)R_ESP;
    unsigned long *buf = (unsigned long *)*r_esp;
    buf[0] = R_EBX;
    buf[1] = R_ESI;
    buf[2] = R_EDI;
    buf[3] = R_EBP;
    buf[4] = R_ESP;
    buf[5] = retaddr;
    R_EAX = 0;
    PC = retaddr;
}
```

Listing A.3: Source of *longjmp.s*

```
.global _longjmp
.global longjmp
.type _longjmp,@function
.type longjmp,@function
_longjmp:
longjmp:
    mov  4(%esp),%edx
    mov  8(%esp),%eax
    test    %eax,%eax
    jnz 1f
    inc     %eax
1:
    mov   (%edx),%ebx
    mov  4(%edx),%esi
    mov  8(%edx),%edi
    mov 12(%edx),%ebp
    mov 16(%edx),%ecx
    mov     %ecx,%esp
    mov 20(%edx),%ecx
    jmp *%ecx
```

Listing A.4: Source of Binrec's longjmp

```

void  __attribute__((always_inline)) nonlib_longjmp(){

    R_ESP += sizeof (stackword_t);
    //get first param(buffer)
    reg_t *r_esp = (reg_t *)R_ESP;
    unsigned long *buf = (unsigned long *)*r_esp;
    //get second param(ret val for setjmp)
    R_ESP += sizeof (stackword_t);
    r_esp = (reg_t *)R_ESP;
    R_EAX = *r_esp;

    if(!R_EAX)
        R_EAX++;
    R_EBX = buf[0];
    R_ESI = buf[1];
    R_EDI = buf[2];
    R_EBP = buf[3];
    R_ESP = buf[4];
    PC = buf[5];
}
```