Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE **Abstraction Recovery for Scalable Static Binary Analysis**

PRESENTED BY **Edward J. Schwartz**

ACCEPTED BY THE DEPARTMENT OF

Electrical and Computer Engineering

David Brumley — ADVISOR, MAJOR PROFESSOR — 4/20/14 DATE

Jelena Kovacevik — DEPARTMENT HEAD — 4/20/14 DATE

APPROVED BY THE COLLEGE COUNCIL

Vijayakumar Bhagavatula — DEAN — 4/20/14 DATE

Abstraction Recovery for Scalable Static Binary Analysis

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Edward J. Schwartz

B.S., Computer Science, Millersville University
M.S., Information Security, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2014

© 2014 Edward J. Schwartz.
All rights reserved.

Acknowledgments

To put it simply, this dissertation would not exist without the help and support of my family, friends, colleagues, and mentors.

First, I would like to thank my colleagues: Thanassis, Jiyong, JongHyup, Sang Kil, Ivan, Spencer, Maverick, Manuel, Wesley, and many others. I thank them for the lively discussions, their valuable insights, and the good company that they have afforded me through the years. I came to understand many problems only after we brainstormed together, often drawing on whiteboards and refining our understanding for hours at a time. I can only hope my future colleagues live up to the highest of standards set by their example.

I would also like to thank my mentors: Drs. Kevin Mills, Adrian Perrig, William Ratcliff, and Thomas Willemain, and especially my thesis committee: Drs. Anupam Datta, Virgil Gligor, and Hovav Shacham. Their teaching and research are what inspired me to pursue a Ph.D. I hope that one day I will be able to follow in their footsteps.

I would like to express my gratitude to my adviser, Dr. David Brumley. Dr. Brumley always managed to find the time to help me with any problem. He gave me enough guidance so that I would not get stuck, but enough independence to grow and develop. He taught me the difficult but important lesson that it can *always* be improved. And finally: he convinced me that program analysis might not be so bad.

I would also like to thank my parents, who always believed that I could succeed, even when I did not. Their unwavering support means more to me than I can express in words.

Finally, I would like to thank my wife, for her love, support, and unfathomable patience. During my Ph.D., I had to spend large amounts of time away from our family, often working on papers instead of helping with housework. I thank her for bearing the weight of my absence so that I could pursue one of my life goals. (Believe me: I had the easier end of the deal!) As our time apart draws to a close, I am eagerly awaiting the next stages of our journey through life together.

Funding

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR00111220009. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

Abstract

Many source code tools help software programmers analyze programs as they are being developed, but such tools can no longer be applied once the final programs are shipped to the user. This greatly limits users, security experts, and anyone other than the programmer who wishes to perform additional testing and program analysis. This dissertation is concerned with the development of scalable techniques for statically analyzing binary programs, which can be employed by anyone who has access to the binary. Unfortunately, static binary analysis is often more difficult than static source code analysis because the abstractions that are the basis of source code programs, such as variables, types, functions, and control flow structure, are not explicitly present in binary programs. Previous approaches work around the the lack of abstractions by reasoning about the program at a lower level, but this approach has not scaled as well as equivalent source code techniques that use abstractions.

This dissertation investigates an alternative approach to static binary analysis which is called abstraction recovery. The premise of abstraction recovery is that since many binaries are actually compiled from an abstract source language which is more suitable for analysis, the first step of static binary analysis should be to recover such abstractions. Abstraction recovery is shown to be feasible in two real-world applications. First, C abstractions are recovered by a newly developed decompiler. The second application recovers gadget abstractions to automatically generate return-oriented programming (ROP) attacks. Experiments using the decompiler demonstrate that recovering C abstractions improves scalability over low-level analysis, with applications such as verification and detection of buffer overflows seeing an average of $17\times$ improvement. Similarly, gadget abstractions speed up automated ROP attacks by $99\times$. Though some binary analysis problems do not lend themselves to abstraction recovery because they reason about low-level or syntactic details, abstraction recovery is an attractive alternative to conventional low-level analysis when users are interested in the behavior of the original abstract program from which a binary was compiled, which is often the case.

Contents

Contents v

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Trade-offs of Static and Dynamic Analysis . . . 2

1.2 Advantages of Binary Analysis . . . 3

1.3 Challenges of Binary Analysis . . . 5

1.4 Existing Approaches to Binary Analysis . . . 6

1.5 Abstraction Recovery . . . 8

1.6 The Abstraction Recovery Problem . . . 9

1.7 Contributions and Outline . . . 15

I Source-code Abstractions 17

2 The Phoenix Decompiler 18

2.1 Introduction . . . 19

2.2 Background . . . 20

2.3 Overview . . . 22

2.4 Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring . . . 26

2.5 Evaluation . . . 35

2.6 Limitations and Future Work . . . 41

2.7 Related Work . . . 42

2.8 Conclusion . . . 44

3 Forward Verification Conditions 45
3.1 Introduction 45
3.2 Background 48
3.3 Forward Verification Conditions 52
3.4 Evaluation 57
3.5 Discussion 63
3.6 Related Work 63
3.7 Conclusion 64

II General Abstractions 65

4 Gadget Abstractions 66
4.1 Introduction 67
4.2 Background 70
4.3 System Overview 73
4.4 Automatically Generating Return-Oriented Payloads 74
4.5 Creating Exploits that Bypass ASLR and DEP 83
4.6 Implementation 86
4.7 Evaluation 86
4.8 Discussion 94
4.9 Related Work 95
4.10 Conclusion 97

5 Scalability 98
5.1 Source-Code Abstractions 99
5.2 Gadget Abstractions 115

6 Conclusions 121
6.1 Future Work 123
6.2 Conclusion 123

III Appendices 125

A Abstraction Recovery Proofs 126

A.1 Proof of Theorem 1.1 126
A.2 Proof of Theorem 1.2 127
A.3 Proof of Theorem 1.3 128

B Forward Verification Conditions Isabelle/HOL Proofs **131**
B.1 Arithmetic and Boolean Expressions 131
B.2 Forward Verification Conditions 134

C Benchmark Programs **140**
C.1 `berkeley` 140
C.2 `barber` 141
C.3 `binary` 143
C.4 `bubble` 144
C.5 `cars` 145
C.6 `efm` 146
C.7 `ex18` 147
C.8 `ex30` 148
C.9 `ex39` 148
C.10 `fib` 149
C.11 `inf3` 150
C.12 `prime` 151
C.13 `sum` 152

Bibliography **153**

List of Figures

1.1 Partition of the program analysis space, and sample of analysis techniques and applications. . . 2
1.2 Simple assembly program. 5
1.3 Example of intermediate language. 7
1.4 Commutative diagram summarizing abstraction recovery. 10

2.1 Example of structural analysis. 21
2.2 Overview of Phoenix's design. 23
2.3 Structural analysis failing without semantics-preservation. 27
2.4 Terminating loop. 28
2.5 Acyclic region types. 29
2.6 Tail regions. 30
2.7 Relationship between complete and incomplete switches. 31
2.8 Cyclic region types. 32
2.9 Loop refinement with and without new loop membership definition. 34
2.10 Recompilability measurements from the coreutils experiment. 38
2.11 Structuredness measurements from the coreutils experiment. 40

3.1 Simple loop. 46
3.2 Guarded command language (GCL). 48
3.3 Forward symbolic execution (FSE). 49
3.4 Weakest preconditions (WP) and weakest liberal preconditions (WLP). 50
3.5 Predicates of the forward verification conditions (FVC) algorithm. 52
3.6 Definition of total GCL programs. 54
3.7 MS^C: MS predicate with concrete evaluation and concrete path pruning. 55
3.8 Total verification time compared to number of loop unrolls. 61
3.9 Formula size compared to program size. 62

4.1 Traditional code injection exploit. 70
4.2 Example return-oriented payload. 73
4.3 Overview of Q's design. 74
4.4 Gadgets assigned from `rsync` to implement the gadget arrangement in Figure 4.8. 75
4.5 Two different arrangements that increment a value in memory. 79
4.6 ML pseudo-code implementing a munch rule. 80
4.7 Return-oriented payload for `apt-get`. 81
4.8 Gadget arrangement and schedule for storing a constant value to a constant address. 82
4.9 Probability that Q can generate payloads as a function of source file size. 87
4.10 Distribution of gadget discovery times. 88
4.11 Gadget discovery time compared to binary size. 89
4.12 Distribution of time elapsed during gadget arrangement and assignment, organized by payload type and whether a payload was successfully produced. 90
4.13 Time elapsed during gadget arrangement and assignment as a function of program size. 91
4.14 Distribution of file sizes in `/usr/bin`. 92
4.15 Distribution of gadgets discovered in each `/usr/bin` program larger than 20 kB. 93

5.1 Overview of source-code scalability experiment. 99
5.2 Total time required to verify correctness of benchmark programs compared to number of loop unrolls. 102
5.3 Total time required to test benchmark programs for buffer overflows compared to number of loop unrolls. 103
5.4 Total time required to verify correctness of benchmark programs compared to formula size. 105
5.5 Formula size of VCs that verify correctness of benchmark programs compared to number of loop unrolls. 106
5.6 Total time required to verify correctness of benchmark programs compared to number of loop unrolls. 108
5.7 Total time required to verify correctness of benchmark programs compared to number of loop unrolls on original and recovered source code. 111
5.8 Total time required to test benchmark programs for buffer overflows compared to number of loop unrolls on original and recovered source code. 112
5.9 Symbolic state space in a code reuse attack. 117
5.10 Code reuse test program. 119

List of Tables

2.1 Binary analysis platform intermediate language (BIL). 24

2.2 High-level intermediate language (HIL). 25

2.3 Correctness summary of the coreutils experiment. 39

2.4 Structuredness summary of the coreutils experiment. 40

3.1 Benchmarks programs tested in correctness experiments. 59

3.2 Verification times for each VC algorithm. 59

4.1 Comparison of defenses on modern operating systems. 71

4.2 Gadget types. 76

4.3 Q's high level language, QooL. 79

4.4 Public exploits hardened by Q. 94

5.1 Maximum number of unrolls before failure for bin, dec, and artificial transformations notypes, novars, and inflate. 109

5.2 Failures during verification. 113

5.3 Failures during overflow checking. 114

5.4 Gadgets added to the code reuse test programs. 118

5.5 Goal states. 118

5.6 Experimental comparison between ROP (Q) and generalized code reuse (Mayhem). 120

Chapter 1

Introduction

Programs are the cornerstone of computer science. Without programs, computers would be doomed to serve as glorified calculators or as simple machines that perform only a single, fixed function. Instead, programmers can instruct a computer to evaluate any computable function by constructing the appropriate program, which allows programmers to experiment with new ideas quickly and cheaply. The ability to rapidly experiment and adapt has led inventors to develop to a diverse set of applications that benefit society, from online, collaborative encyclopedias, to distributed computing projects that unravel the mysteries of protein folding.

Unfortunately, some programs are buggy, which can undermine the utility that computer programs otherwise offer. Some programs may crash; others may hang. A program may also appear to operate correctly, but actually compute a value incorrectly, which is one of the most serious problems that could happen. For example, a program that incorrectly computes the static stresses of a building, or inaccurately forecasts the likelihood of a hurricane could lead to physical damage, financial loss, and even death. Unfortunately, such outcomes can also be caused by individuals with less benign intentions as well. Many programs contain software vulnerabilities, or bugs that can enable an adversary to influence a program in ways the programmer never intended. Such vulnerabilities can allow hackers to leak secret information, modify sensitive data, and even run arbitrary commands on the vulnerable system.

Programmers and researchers have proposed automated techniques for analyzing programs and detecting these types of bugs during development, so that the programmers can fix them before they cause a problem. Automated techniques offer several compelling advantages over manual analysis and review. First, the cost of manual program analysis is very high. Human analysts can require long periods of time to understand complex programs. Even small changes to complex programs can have far-reaching side effects, which a human may take a long time to grasp. In contrast, automated analyses are often designed

to complete in hours or days, and can be repeatedly applied often and at relatively low cost. Second, even when a human does have time to fully analyze a program, he can make mistakes. After all, humans are "just human." Some programming behaviors are so arcane and complex that many programmers have trouble understanding the corner cases, such as the rules for integer overflow and promotions in C. In contrast, automated techniques can be sound, which ensures that every bug the technique reports actually does exist, or complete, which guarantees every bug in the program will be reported by the technique.

This dissertation is about static, binary program analyses. Researchers categorize program analyses along two dimensions: dynamic vs. static, and binary vs. source. These dimensions are illustrated in Figure 1.1. A dynamic analysis executes a program and then examines artifacts of the execution to check for certain properties or bugs. In contrast, a static analysis never executes the program to be analyzed; program properties are instead inferred by reasoning about what the program *would* do if it executed. A source code analysis examines the source code representation of a program for bugs or other properties. On the other hand, a binary analysis examines the compiled or binary representation of a program instead.

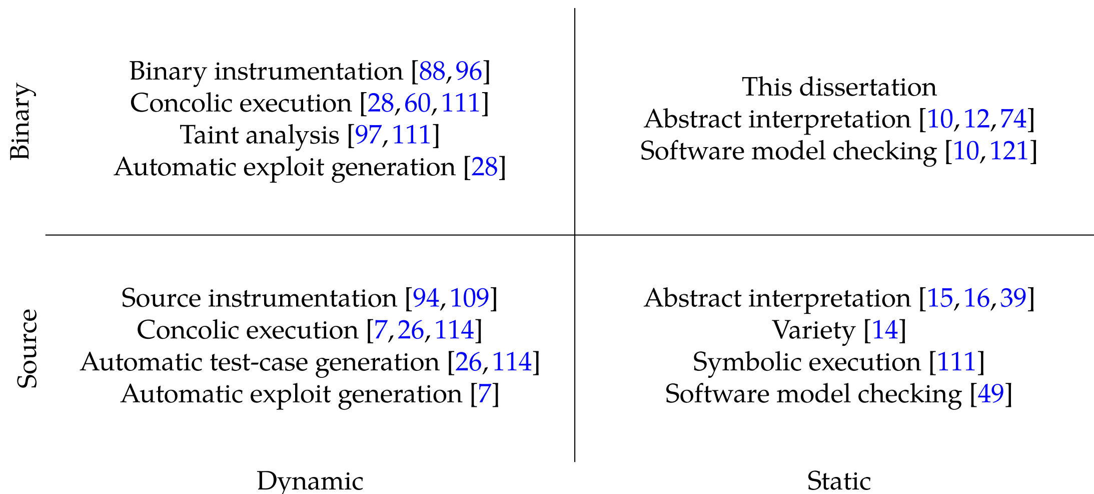


Figure 1.1: Partition of the program analysis space, and sample of analysis techniques and applications.

1.1 Trade-offs of Static and Dynamic Analysis

This dissertation focuses on static analysis, but each type of analysis is well suited to different situations and goals. This section discusses the most notable trade-offs between static and dynamic analysis.

Precision A dynamic analysis can make precise judgments about program behavior because it examines real program executions. Observing a behavior during a real execution is sufficient evidence to conclude that some executions of the program have this behavior. For instance, a dynamic analysis can judge that a

program has a buffer overflow vulnerability when it actually observes the buffer overflow in a real execution. In contrast, static analyses reason about program behavior without actually executing it, which often introduces some imprecision.

Coverage The trade-off for the precision of dynamic analyses is that they are generally unable to make judgments about inputs or environments other than the one the analysis observed. For example, if a programmer runs a dynamic analysis on her program and finds that it does not exhibit buggy behavior, the program could still encounter a bug when run on another input or in a different environment. Static analyses do not need to actually run the program for each execution reasoned about, and are free to reason about many or all executions at the same time. Static analysis can thus make judgments about all or many program executions.

Environment Static analyses often need to model external components in the program environment, such as external libraries or operating system system calls, whereas dynamic analyses do not. Dynamic analyses are able to use information from the actual program execution to understand how external components behaved. Static analyses do not have this option, however, and the behavior of external components must often be manually modeled into the analysis itself. Accurate modeling of the environment is often critical for effective program analysis. For example, an analysis that searches for buffer overflows might need to model the effects of the `strcpy` function in libc, which copies a string from one buffer to another. Without understanding the behavior of `strcpy`, the analysis might be unable to detect many buffer overflows.

Side effects Static analysis can detect program behaviors before they have a chance to affect the system they run on. This is useful when untrusted programs must be analyzed, since such programs can be safely analyzed even if they are buggy, dangerous, or malicious. Dynamic analysis delays the time of detection until after the program has already started executing, and as a result it is possible for the program to perform irreversible, dangerous, or malicious actions.

1.2 Advantages of Binary Analysis

This dissertation advocates for binary analysis as an attractive option for developing program analyses. This section discusses some of the most attractive features of binary analysis. The next section discusses some of the challenges and disadvantages of binary analysis.

End-user applicability One of the most compelling advantages of binary analysis is that it enables parties other than the programmer to analyze the program, because source code is not required. In theory, binary analysis allows a user to analyze every binary that she can execute. This ability is especially important in security settings, since a user may not completely trust a programmer to provide a correct or bug-free program, and may want to verify this herself. In contrast, a source analysis requires source code to operate, and source code is unavailable to users for many popular software packages. A user of such a software package would be unable to use a source analysis to analyze the program.

Undefined behavior Binary analysis can reason about the actual behavior of the program as it executes on the processor. This is important because program behavior is often undefined or ambiguous at the source level: a common example is that aggressive optimizers in C compilers often remove code intended to detect integer overflows [128]. Even mature code bases are not immune to such problems; for example, Python, QEMU, and the Linux kernel all contained undefined behavior [128]. It is difficult to write a source analysis that accurately models such behavior, since the program that eventually runs on the processor may act differently depending on the compiler, optimization level, and flags. Even when the program is undefined at the source level, compilers still select a well defined binary program to implement it.[1] A binary analysis can thus avoid ambiguity in source by analyzing the compiled program, which is well defined.

Binary features Some program analyses can only be performed at the binary level, because the properties of interest are not present at the source level. One example is automatic exploit generation (AEG): the process of checking a program for exploitable bugs and creating exploits for each [7,8,28]. AEG uses binary level details such as stack frame layout and padding, and thus would not be possible at the source level, since an exploit is specific to the binary executable. Interestingly, the first implementation of AEG operated on source code [7], but also had to include a binary analysis to infer such binary level details.

Language-agnosticism A binary analysis can reason about binary programs that were compiled from many source languages, and those written directly in assembly language. This is important because programmers of source program analyses often must create a new program analysis for each source language, which is often a complex, time consuming, and tedious effort. In practice, many source analyses are instead developed for a single source language. Developing binary analyses is an attractive alternative, since many source languages can be compiled to a binary, and thus in theory be analyzed by a single binary analysis.

[1] Processors do have undefined behaviors, but compilers rarely generate code that invokes them. This is true even for programs that are undefined at the source level.

1.3 Challenges of Binary Analysis

This section discusses the major challenges of static binary analysis, and the next describes each challenge's proposed solution.

Complex semantics Binary code is comprised of assembly or machine instructions, each of which encodes a single step of a computation so that a computer processor can execute it. Unfortunately, modern Intel x86 and x86-64 processors understand hundreds of different instructions and even seemingly simple instructions often have complex side effects. For instance, the program in Figure 1.2 adds two numbers, performs a shift, and then jumps to the target if the carry flag is set. However, the assembly code representation of the program does not reveal when the carry flag is set; the behavior of the carry flag flag is *implicit*. In this program, the implicit behavior would make it difficult for an analysis to determine the precise conditions under which `target` would be executed.

```
add %eax, %ebx
shl %cl, %ebx
jc target
```

Figure 1.2: Simple assembly program.

Lack of abstractions Static analyses for source programs are difficult to adapt to compiled binary programs, because compilation removes the abstractions that such analyses depend on, such as variables, types, functions, and control-flow structure. These abstractions are removed during compilation and replaced with concrete implementations that will run on a processor. Unfortunately, these concrete implementations are generally not as amenable to analysis as the abstract source language. For example, a source analysis can see local variables, and thus can easily compute the data dependencies between them, since variables are in the grammar of the source language. During compilation, local variables are often implemented by assigning storage locations inside of a function's stack frame. Unfortunately, a binary analysis has no easy way to distinguish individual variables inside of the stack frame, so computing precise data dependencies between variables becomes a challenging problem. Challenges like these make it difficult to adapt source analyses to analyze compiled programs. It is even more difficult to adapt source analyses to analyze binary programs not compiled from the same source language, as these binary programs may not even have a natural representation in the source language.

Indirect jumps The indirect jumps in binary code make it difficult to put binary programs into the control flow graph (CFG) representation often used by static analyses. Indirect jumps are jumps to computed tar-

gets, such as `jmp *%eax`, rather than addresses explicitly specified in the instruction, such as `ja $0x1000`. Unfortunately, precisely resolving the successors of indirect jumps is known to be an undecidable problem [66], and many static analyses become unsound or imprecise when they are given an incomplete or over-connected CFG because they utilize a CFG to reason about possible state transitions [74].

1.4 Existing Approaches to Binary Analysis

Many of the challenges in the previous section have been addressed by prior work.

Intermediate languages Researchers have mitigated the complexities of modern instruction sets by modeling instruction behavior in an intermediate language (IL) [20, 22, 33]. Intermediate languages were first introduced in compilers to connect front-end parsers for specific languages to back-ends that perform optimization, analysis, and code generation. This design allows the compiler back-ends to be easily reused with new front-ends. In binary analysis, the front-end converts machine code from different architectures into IL, which is then analyzed by the back-end, allowing the same analyses to analyze binaries from different architectures. The IL used for binary analysis is typically simple to ease the creation of new analyses; in contrast, the ILs used in compilers are often more sophisticated to enable highly specialized code generation. During lifting, each instruction is converted to one or more IL statements, with side effects such as flag computations being explicitly represented in the IL.

Intermediate languages also allow analyses to cope with a lack of abstractions because they describe binary semantics at a low-level. Source analyses analyze programs in terms of their high-level abstractions, which are compiled to concrete behaviors implemented by machine code. After the machine code is lifted to IL, the IL represents the semantics of these low-level concrete behaviors, which can be analyzed instead of the source abstractions. Although IL is a language and has abstractions such as variables and control flow, it does not have the original program abstractions. For example, variables in the IL correspond to processor state, including registers, flags, and memory. An assignment to a variable in source might be represented in the IL by a write to memory that updates the corresponding memory cell; variables in source generally do not have corresponding variables in the IL.

The assembly program from Figure 1.2 is lifted to the Binary Analysis Platform (BAP) [22] IL (BIL) and shown in Figure 1.3. Note that several flags are explicitly computed in the BIL after the `add` and `shl` instructions. Also note the complex behavior of the flags computed by `shl`: when the shift amount is 0, the flags are not modified.

```
addr 0x0 @asm "add    %eax,%ebx"
label pc_0x0
T_t1:u32 = low:u32(R_RBX:u64)
T_t2:u32 = low:u32(R_RAX:u64)
R_RBX:u64 = pad:u64(low:u32(R_RBX:u64) + T_t2:u32)
R_CF:bool = low:u32(R_RBX:u64) < T_t1:u32
R_OF:bool =
  high:bool((T_t1:u32 ^ ~T_t2:u32) & (T_t1:u32 ^ low:u32(R_RBX:u64)))
R_AF:bool =
  0x10:u32 == (0x10:u32 & (low:u32(R_RBX:u64) ^ T_t1:u32 ^ T_t2:u32))
R_PF:bool =
  ~low:bool(let T_acc:u32 := low:u32(R_RBX:u64) >> 4:u32 ^ low:u32(R_RBX:u64) in
            let T_acc:u32 := T_acc:u32 >> 2:u32 ^ T_acc:u32 in
            T_acc:u32 >> 1:u32 ^ T_acc:u32)
R_SF:bool = high:bool(low:u32(R_RBX:u64))
R_ZF:bool = 0:u32 == low:u32(R_RBX:u64)
addr 0x2 @asm "shl    %cl,%ebx"
label pc_0x2
T_origDEST:u32 = low:u32(R_RBX:u64)
T_origCOUNT:u32 = low:u32(R_RCX:u64) & 0x1f:u32
R_RBX:u64 = pad:u64(low:u32(R_RBX:u64) << (low:u32(R_RCX:u64) & 0x1f:u32))
R_CF:bool =
  if T_origCOUNT:u32 == 0:u32 then R_CF:bool else
  low:bool(T_origDEST:u32 >> 0x20:u32 - T_origCOUNT:u32)
R_OF:bool =
  if T_origCOUNT:u32 == 0:u32 then R_OF:bool else
  if T_origCOUNT:u32 == 1:u32 then high:bool(low:u32(R_RBX:u64)) ^ R_CF:bool
  else unknown "OF undefined after shift":bool
R_SF:bool =
  if T_origCOUNT:u32 == 0:u32 then R_SF:bool else
  high:bool(low:u32(R_RBX:u64))
R_ZF:bool =
  if T_origCOUNT:u32 == 0:u32 then R_ZF:bool else 0:u32 == low:u32(R_RBX:u64)
R_PF:bool =
  if T_origCOUNT:u32 == 0:u32 then R_PF:bool else
  ~low:bool(let T_acc_113:u32 :=
                                  low:u32(R_RBX:u64) >> 4:u32 ^
                                  low:u32(R_RBX:u64) in
            let T_acc_113:u32 := T_acc_113:u32 >> 2:u32 ^ T_acc_113:u32 in
            T_acc_113:u32 >> 1:u32 ^ T_acc_113:u32)
R_AF:bool =
  if T_origCOUNT:u32 == 0:u32 then R_AF:bool else
  unknown "AF undefined after shift":bool
addr 0x4 @asm "jb     0x000000000000000a"
label pc_0x4
cjmp R_CF:bool, 0xa:u64, "nocjmp0"
label nocjmp0
```

Figure 1.3: Example of intermediate language.

Control flow reconstruction Researchers have also been studying control flow reconstruction [10, 12, 74] techniques for recovering CFGs of binary programs in the presence of indirect jumps. Although precisely resolving the CFG is undecidable [66], researchers have developed techniques to recover over-approximations of the CFG, which they have also shown is sufficient to create sound analyses. These techniques have been demonstrated on a variety of code bases, including device drivers [10, 74], small hand-written C programs [12], and an embedded aeronautics program [12].

1.5 Abstraction Recovery

Researchers have made great strides in creating static binary analyses by reasoning at a low-level to cope with the lack of abstractions. This dissertation explores an alternative approach to static binary analysis called *abstraction recovery*. The premise of abstraction recovery is that since many binaries are actually compiled from an abstract source language which is more suitable for analysis, the first step of static binary analysis should be to recover such abstractions. Then, using the recovered abstractions, the program can be analyzed. This dissertation supports the thesis that:

> *For C and gadget abstractions, (T1) it is possible to compute observable properties of an abstract program's behavior by recovering abstractions from its compiled implementation, and furthermore, (T2) it is faster to recover the abstractions and analyze them than to analyze the implementation directly.*

Abstraction recovery is largely motivated by the success of static source analysis, which empirically demonstrates that modern programs can be statically analyzed when given the right program abstractions. A source program and its compiled binary only differ in their level of abstraction; both programs still perform the same computation. Unfortunately, this seemingly small difference has a large effect in practice. For example, Chapter 5 demonstrates that recovering C abstractions allows programs to be verified $17\times$ faster than only analyzing the low-level binary representation, and that return-oriented programming attacks can be constructed $99\times$ faster using gadget abstractions.

Binary analysis researchers state that these abstractions are lost during compilation [20, 74], so it may seem unclear why it is possible to recover abstractions from binaries in the first place. Source abstractions are explicitly represented in the syntax of programming language because programmers use them to describe the programs they wish to construct. Thus, a source analysis only needs to parse the source code of a program to gain access to source abstractions. When a source program is compiled to a binary, these abstractions are no longer *explicitly* represented, which means that a binary analysis cannot simply parse the binary to gain access to the abstractions; this is the sense in which the abstractions are "lost". However,

a compiled binary remains an *implementation* of the original source abstractions, and one of the hypotheses of abstraction recovery is that it is possible to recover abstractions based solely on the behavior of the binary; though the binary does not explicitly describe the abstractions, it implicitly describes them through the behavior of its low-level implementation.

One of the general disadvantages of abstraction is that some objects do not have an abstract representation, and this problem applies to abstraction recovery as well. Traditional binary analysis can analyze any binary in theory,[2] but some abstractions cannot describe all binaries. For example, an assembly program may contain functionality that cannot be expressed in C, and so it does not make sense to recover C abstractions for hand-written assembly programs. However, there are other types of abstractions beyond source abstractions.

General abstractions summarize the low-level actions of binaries that can be expected in all binaries. For example, the gadget abstraction (Chapter 4) describes gadgets, which are high-level semantic actions such as moving data between registers, adding numbers, and writing to memory. All binaries can be expected to contain these types of behaviors, which allows the gadget abstraction to be recovered on any binary, since gadgets are usually created unintentionally.

1.6 The Abstraction Recovery Problem

This section formally defines the abstraction recovery problem, and then uses it to prove several theorems that explain when and why abstraction recovery is possible.

An instance $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, [\![-]\!] \rangle$ of the abstraction recovery problem is defined in terms of an original analysis function $[\![-]\!] : \mathbb{A} \to \mathbb{O}$, where $\mathbb{A}$ is the set of abstract programs (e.g., source code programs) the analysis is defined on, and $\mathbb{O}$ is the set of analysis outcomes. At a high level, $[\![-]\!]$ describes the desired analysis result for every abstract program. For instance, $\mathbb{A}$ might be the set of C programs, and $\mathbb{O}$ the set $\{\mathsf{true}, \mathsf{false}\}$, denoting whether the program contains a buffer overflow vulnerability.

The binary aspect of abstraction recovery is modeled by a set of concrete programs (e.g., binary programs) $\mathbb{C}$ and a concretization relation $\gamma : \mathbb{A} \to \mathcal{P}(\mathbb{C})$, which defines the acceptable or equivalent concrete implementations for each abstract program. Note that there can be many acceptable concrete implementations for a single abstract program. For simplicity, it is required that all concrete programs represent at least one abstract program:

$$\forall c \in \mathbb{C}.\, \exists a \in \mathbb{A}.\, c \in \gamma(a). \tag{1.1}$$

A solution to an instance of an abstraction recovery problem is a tuple $\langle \mathbb{R}, \alpha, \lfloor\!\lfloor - \rfloor\!\rfloor \rangle$ consisting of:

[2]In practice, applying static binary analysis to atypical binaries often produces results so imprecise that they are not useful.

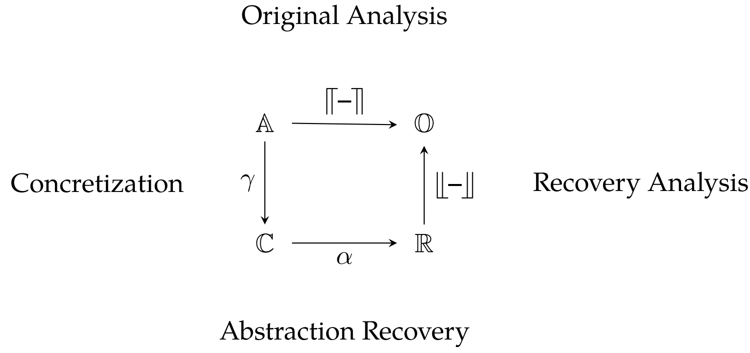


Figure 1.4: Commutative diagram summarizing abstraction recovery.

- a set $\mathbb{R}$ of recovered abstractions,
- an abstraction recovery function $\alpha : \mathbb{C} \to \mathbb{R}$ that recovers abstract representations from concrete programs, and
- a recovery analysis function $\llparenthesis - \rrparenthesis : \mathbb{R} \to \mathbb{O}$ that returns the analysis outcome for the recovered abstraction.

A solution $(\mathbb{R}, \alpha, \llparenthesis - \rrparenthesis)$ must return the same outcome for each low-level program as any abstract program it implements:

$$\forall a \in \mathbb{A}.\, \forall c \in \gamma(a).\ \llbracket a \rrbracket = \llparenthesis \alpha(c) \rrparenthesis. \tag{1.2}$$

The abstraction recovery problem is depicted via a commutative diagram in Figure 1.4.

1.6.1 Can abstraction recovery problems always be solved?

The abstraction recovery definition can be used to explain several practical issues surrounding abstraction recovery, including the seemingly negative result that some abstraction recovery instances are unsolvable:

Theorem 1.1. Given $\mathbb{A}$, $\mathbb{C}$, $\mathbb{O}$, and γ, a solution to the abstraction recovery problem $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ exists for all $\llbracket - \rrbracket$ if and only if γ is injective:

$$\forall a_1, a_2 \in \mathbb{A}.\, a_1 \neq a_2 \implies \gamma(a_1) \cap \gamma(a_2) = \varnothing. \tag{1.3}$$

Proof. See Appendix A. □

Theorem 1.1 states that if the concretization function γ is not injective, then there exists an abstraction recovery problem that cannot be solved. Intuitively, a non-injective concretization function implies that

multiple abstract programs share implementations. This is a problem because each abstract program might have a different analysis outcome. For instance, consider the analysis "How many times is multiplication used?" and the abstract programs $x+x$ and $2*x$. Because both abstract expressions can be computed by the same low-level implementation, and each implementation uses multiplication a different number of times, it is impossible to always answer correctly. Other types of abstractions that are generally lost during the compilation process include variable names and comments.

1.6.2 Which abstraction recovery problems can be solved?

Theorem 1.1 may seem like a negative result, because it reveals that abstraction recovery is not always possible. Theorem 1.2 adds additional detail, by explaining the conditions under which abstractions can be recovered:

Theorem 1.2. Let $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ be an instance of the abstraction recovery problem, and let $\mathbb{A}/\!\sim$ be the equivalence class defined by $\sim$ where $x \sim y := \llbracket x \rrbracket = \llbracket y \rrbracket$. Then, a solution to the abstraction recovery problem $\langle \mathbb{R}, \alpha, \llfloor - \rrfloor \rangle$ exists if and only if each concrete program only implements abstract programs from one equivalence class:

$$\forall c \in \mathbb{C}.\, \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E. \tag{1.4}$$

Proof. See Appendix A. □

The intuition behind Theorem 1.2 is that even though a concrete program might implement multiple abstract programs, if they all have the same analysis outcome finding the exact abstract program which the concrete program implements is unnecessary. For example, consider the analysis "Is this program equivalent to $x + x$?" and the abstract programs $x + x$ and $2 * x$. If both of these abstract programs are the only programs that compile to some concrete program c, then the analysis result for c is true.

1.6.3 What properties of program behavior are observable?

Theorems 1.1 and 1.2 explain what abstraction recovery problems can be solved for arbitrary concretization relations γ. However, in many cases, γ describes a compiler, and $\llbracket - \rrbracket$ is a property of a program's semantics, or behavior. Intuitively, unlike an arbitrary concretization relation, a compiler must preserve some properties of an abstract program's behavior, since it is creating an implementation of that program. A natural question to ask is what properties of an abstract program's behavior are preserved by compilation, and can thus be observed in a concrete implementation. For example, can an analysis always determine whether a

program can return zero, or if a particular line of code can be executed, by looking at a concrete implementation?

A fully abstract denotational semantics can help answer these types of questions. Denotational semantics are one way to define a program's meaning by mapping each program construct, such as programs or expressions, to a mathematical object. A denotational semantics can be *fully abstract* with respect to some type of observation, which means that two programs have the same denotation if and only if executions of both programs are observationally indistinguishable under any possible context (e.g., initial environment or input) [102]. For example, two programs might be considered indistinguishable if and only if both programs return the same output for all inputs.

A fully abstract semantics precisely characterizes *observable properties*, which are those properties of the original abstract program that must be preserved by any implementation. If these properties are not preserved, the implementation would be distinguishable from the original abstract program, and thus not a correct implementation. More precisely, an analysis $\llparenthesis - \rrparenthesis : \mathbb{A} \to \mathbb{O}$ computes an *observable property* $\mathbb{O}$ with respect to a fully abstract denotational semantics $\llbracket - \rrbracket : \mathbb{A} \to \mathbb{S}$ if and only if

$$\forall a_1, a_2 \in \mathbb{A}.\ \llbracket a_1 \rrbracket = \llbracket a_2 \rrbracket \implies \llparenthesis a_1 \rrparenthesis = \llparenthesis a_2 \rrparenthesis. \tag{1.5}$$

In other words, an analysis returns an observable property if it is a function of a program's fully abstract semantics.

On the other hand, a concretization function γ *respects the semantics of* $\llbracket - \rrbracket$ if and only if all pairs of abstract programs with different denotations also have mutually exclusive implementations:

$$\forall a_1, a_2 \in \mathbb{A}.\ \llbracket a_1 \rrbracket \neq \llbracket a_2 \rrbracket \implies \gamma(a_1) \cap \gamma(a_2) = \varnothing. \tag{1.6}$$

The intuition behind this property is that if two programs have distinct denotations then they are also observationally distinct, and thus they cannot use the same implementation. Using these definitions it is possible to show that a solution to an abstraction recovery problem exists if and only if the analysis computes an observable property:

Theorem 1.3. Given $\mathbb{A}$, $\mathbb{C}$, $\mathbb{O}$, $\llparenthesis - \rrparenthesis$ and $\llbracket - \rrbracket$, a fully abstract semantics of $\mathbb{A}$, a solution to the abstraction recovery problem $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llparenthesis - \rrparenthesis \rangle$ exists for all γ that respect the semantics of $\llbracket - \rrbracket$ if and only if $\llparenthesis - \rrparenthesis$ computes an observable property with respect to $\llbracket - \rrbracket$.

Proof. See Appendix A. □

Observable properties of C

The properties of a language that are observable in practice depend on the language's fully abstract semantics and the definition of observational indistinguishability. For example, the C11 standard defines the conditions under which a program execution is indistinguishable from an execution on an ideal abstract machine as: [70, 5.1.2.3.6]:

> The least requirements on a conforming implementation are:
>
> - Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
> - At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
> - The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.
>
> This is the *observable behavior* of the program.

In other words, a C program and its implementation are observationally indistinguishable if their volatile objects, files, and input/output dynamics agree (where agreement is defined in the standard). Theorem 1.3 explains that the only analyses for which abstraction recovery is always possible are for observable properties, which in this case are functions over the values of volatile objects, the contents of files, and the input/output behavior of a program.

It should not be surprising that some properties are not observable. As one example, the values of local, non-volatile variables are *not* observable. This means that properties such as whether the variable x can ever take the value zero are only observable if x is volatile. This rules out analyses that reason about the reachability of intermediate execution states, such as whether a particular line of code will be reached.

The good news is that many analyses are based on observable properties. For example, input/output behavior is an observable property, and so it is possible to verify that the output of a program matches some correctness property, such as ensuring that the output of a sorting implementation is always sorted.

1.6.4 Decompilation

The abstraction recovery definition allows a concise definition of *decompilation*, which occurs when a solution has $\mathbb{R} = \mathbb{A}$. In other words, decompilation recovers the same abstraction of the program that the

original analysis is defined in. This is a natural approach to abstraction recovery because it does not lose any information; the proof of Theorem 1.2 in Section A.2 shows that decompilation is sufficient to solve an instance of abstraction recovery as long as a solution exists. A practical advantage of decompilation is that there are often existing algorithms for computing the original analysis function $\llbracket - \rrbracket$, and these implementations can be utilized by defining $\llfloor c \rrfloor := \llbracket \alpha(c) \rrbracket$. In this dissertation, each instance of abstraction recovery has $\mathbb{R} = \mathbb{A}$, but this is not a general requirement.

1.6.5 Generality

Abstraction recovery can also describe traditional binary analysis problems, which reason about the properties of concrete programs instead of abstract programs. This can be used to represent syntactic binary analysis problems, which are used in malware clustering [71], for example. Such problem instances are defined by setting $\mathbb{A} = \mathbb{C}$ and γ to the identity function. A solution to the problem then simplifies to a tuple $(\mathbb{R}, \alpha, \llfloor - \rrfloor)$ such that

$$\forall c \in \mathbb{C}. \llbracket c \rrbracket = \llfloor \alpha(c) \rrfloor. \tag{1.7}$$

It is also possible to use low-level binary analysis algorithms, which do not explicitly recover abstractions, to solve abstraction recovery instances by setting $\mathbb{R} = \mathbb{C}$ and α to the identity function. In this case, a solution to abstraction recovery simplifies to a recovery analysis function $\llfloor - \rrfloor$ such that

$$\forall a \in \mathbb{A}. \forall c \in \gamma(a). \llbracket a \rrbracket = \llfloor c \rrfloor. \tag{1.8}$$

While this type of solution is possible, the goal of this dissertation is to highlight the utility of explicitly reasoning about programs using abstractions.

1.6.6 Approximation

The abstraction recovery definition assumes that each abstract program has a unique analysis outcome. In practice, many analyses are not designed with exact solutions because the exact analyses are undecidable. Instead, approximations are often used, which may be decidable even when an exact solution is not. Analysis outcomes can be approximated by making $\mathbb{O}$ a partially ordered set $\langle \mathbb{O}, \sqsubseteq_{\mathbb{O}} \rangle$, where $\sqsubseteq_{\mathbb{O}}$ is the partial ordering relation for $\mathbb{O}$ such that $a \sqsubseteq_{\mathbb{O}} b$ denotes that b over-approximates (is more general than) a, or a under-approximates (is more specific than) b. Using the partial ordering, it is then possible to specify that a solution to abstraction recovery should be an over-approximation:

$$\forall a \in \mathbb{A}. \forall c \in \gamma(a). \llbracket a \rrbracket \sqsubseteq \llfloor \alpha(c) \rrfloor \tag{1.9}$$

Unfortunately, the price of this more general definition is that Theorems 1.2 and 1.3 no longer apply. The high-level reason is that under the new definition there can be a solution even if two abstract programs a_1 and a_2 are implemented by the same concrete program c and have different analysis outcomes ($[\![a_1]\!] \neq [\![a_2]\!]$). If there is an analysis outcome o such that $[\![a_1]\!] \sqsubseteq o$ and $[\![a_2]\!] \sqsubseteq o$, then o would be a valid outcome for c. Unfortunately, this property changes the structure of the problem.

Abstract interpretation [39] is a framework commonly used for approximating program properties by employing abstraction. At a high-level, abstract interpretation establishes a connection between concrete program states and abstract program states, which is similar to the connection between the set of abstract $\mathbb{A}$ and concrete $\mathbb{C}$ programs in abstraction recovery. However, abstract interpretation is concerned with reasoning about the behavior of a *concrete* program by employing abstraction to achieve computability and efficiency. In abstraction recovery, the goal is to understand the behavior of an *abstract* program by examining a concrete implementation. Rather than employing abstraction for approximation, abstraction is used to remove implementation details that are specific to the concrete representation, so that properties of the original abstract program can be recovered instead. These two uses of abstractions can be combined to approximate the properties of an abstract program from its compiled implementation: an abstract program representation (e.g., C abstractions) could first be recovered, and these abstractions could be further abstracted using abstract interpretation to approximate a program property of interest.

1.7 Contributions and Outline

The principal contributions of this dissertation are:

- The proposal of abstraction recovery, a new framework for designing static binary analyses (Chapter 1).
- The design and implementation of a x86 to C decompiler optimized for correct and effective abstraction recovery, called Phoenix (Chapter 2). Phoenix recovers C abstractions from compiled C programs and enables their use in abstraction recovery.
- The development of an efficient weakest preconditions algorithm, FVC, that leverages variable and type abstractions to optimize analysis performance (Chapter 3). FVC can find bugs and verify programs at both the concrete, low-level binary representation and the abstract source representation, and so can quantify the performance benefits of abstraction recovery.
- The design and development of an automatic system for return-oriented programming (Chapter 4), which demonstrates the utility and feasibility of general abstractions.

- A performance comparison of abstraction recovery-based algorithms and low-level algorithms which demonstrates that C and gadget abstractions provide speedups of $17\times$ and $99\times$respectively (Chapter 5).

Chapter 2 and 4 are based on papers that have already been published [112, 113].

Part I

Source-code Abstractions

Chapter 2

The Phoenix Decompiler

This part of the dissertation focuses on the recovery and use of *source* abstractions, and this chapter describes the design, implementation, and evaluation of a decompiler for recovering such abstractions. At a high level, decompilers undo the decompilation process; they take a compiled program as input and return equivalent source code as output. In abstraction recovery terminology, decompilation corresponds to the abstraction recovery function α when both the original abstraction $\mathbb{A}$ and recovered abstraction $\mathbb{R}$ are source programs.

The use of decompilation for abstraction recovery is largely motivated by the success of existing tools and techniques for analyzing source programs [14–16, 21, 39, 49, 81, 127]. The scalability and capabilities of the state of the art tools are impressive. As one example, Coverity has statically analyzed over a *billion* lines of code in search of bugs, vulnerabilities, and undefined behaviors. Unfortunately, almost all such tools make extensive use of abstractions to help scale their analysis, making them difficult to adapt to compiled programs, which lack abstractions. As a result, these tools are primarily of use to programmers, and are not directly useful to a user who does not have source code.

Decompilation is a natural way for users to leverage these existing source code tools when they do not have access to the original source code. The user can run the decompiler on the binary she wishes to analyze to recover the source code for the program. The recovered source code does not need to be identical, but in many scenarios it is important that the recovered source code is semantically equivalent to the original source code. The user then runs her source code tool of choice on the recovered source code. This process may sound deceptively simple, but it is effective: Chapter 5 demonstrates that recovering source code and analyzing it can be significantly faster than reasoning about the binary at a low-level.

2.1 Introduction

Decompilation is the process of recovering source code from a compiled program. The ability to recover source code has many potential applications, but most work to date has focused on manual reverse engineering, or recovering source code that humans can read to better understand the binary program. Unfortunately, having been largely focused on reverse engineering, current research in decompilation does not directly cater to the needs of abstraction recovery. To be used for abstraction recovery, a decompiler should focus on two properties. First, it should recover the most abstract representation possible to minimize the complexity that must be handled by the actual analysis that follows. Second, it should aim to recover these abstractions correctly. As surprising as it may sound, previous work on decompilation almost never evaluated correctness. For example, Cifuentes et al.'s pioneering work [33] and numerous subsequent works [29, 34, 36, 125] all measured either how much smaller the output C code was in comparison to the input assembly or another subjective readability metric, and did not evaluate correctness.

This chapter demonstrates that C source abstractions can be recovered in a principled fashion by building a new end-to-end binary-to-C decompiler called *Phoenix*[1] using new techniques for accurately and effectively recovering abstractions. Decompilation requires the recovery of two types of abstractions: data type abstractions and control flow abstractions. Recent work such as TIE [82], REWARDS [86], and Howard [119] have largely addressed principled methods for recovering data types. This chapter is primarily focused on new techniques for recovering high-level control flow structure such as if-then-else blocks and while loops.

Previous work has proposed algorithms for recovering high-level control flow based on the well known structural analysis algorithm and its predecessors [52, 62, 125]. However, these algorithms are problematic for decompilation because they (P1) do not feature a correctness property that is necessary to be safely used for decompilation, and (P2) miss opportunities for recovering control flow structure. These problems can cause an analysis that uses the recovered abstractions to scale poorly, lose precision, or even become unsound, and motivated the new control flow structuring algorithm developed for Phoenix. Phoenix's algorithm is based on structural analysis, but avoids the problems identified in earlier work:

- Control flow structuring algorithms should have a *semantics-preservation* property to be safely used for decompilation, because it ensures that an analysis of the structured program will also apply to the original. Surprisingly, one of the most common structuring algorithms, structural analysis [91, p. 203], does *not* have the semantics-preservation property. As a result, using structural analysis can lead to

[1] Phoenix is named in honor of the famous "Dragon Book" [2] on compilers. According to Chinese mythology, the phoenix is a supreme bird that complements the dragon (compilation). In Greek mythology, the phoenix can be reborn from the ashes of its predecessor. Similarly, a decompiler can recover source code and abstractions from the compiled form of a binary, even when these artifacts seem to have been destroyed.

incorrect decompilation and analysis. Phoenix's structuring algorithm addresses this problem, and increases the number of utilities that Phoenix is able to correctly decompile by 30% (Section 2.5).

- Structural analysis algorithms are unable to effectively recover structure from unstructured program fragments, which must be constructed using gotos and similar statements. Phoenix's algorithm uses *iterative refinement* to discover structure in such problem regions. The basic idea of iterative refinement is to temporarily remove the edges corresponding to gotos and similar statements during the control flow structuring process. After removing these edges from the graph, Phoenix's algorithm can recover additional structure, whereas the structural analysis algorithm would halt on that part of the program. Phoenix's evaluation demonstrates that iterative refinement recovers 30× more structure than structural analysis without iterative refinement (Section 2.5).

Contribution The major contribution of this chapter is the design, implementation, and evaluation of the Phoenix x86 to C decompiler. Phoenix features a new control flow structuring algorithm that addresses problems observed in previous approaches, which can cause incorrect decompilation and fewer abstractions to be recovered. Phoenix was evaluated in the largest systematic end-to-end decompiler evaluation of correctness and control flow to date, and recovered 30× more control-flow structure than existing research in the literature [52, 91, 117], and 28% more than the *de facto* industry standard decompiler Hex-Rays [62]. Phoenix also decompiled over 2× as many programs that pass the coreutils test suite as Hex-Rays.

2.2 Background

2.2.1 Control Flow Analysis

A *control flow graph* (CFG) of a program P is a directed graph $G = (\mathbb{N}, \mathbb{E}, n_s, n_e)$. The node set $\mathbb{N}$ contains basic blocks of program statements in P. Each basic block must have exactly one entrance at the beginning and one exit at the end. Each time the first instruction of a basic block is executed, the remaining instructions must also be executed in order. The nodes $n_s \in \mathbb{N}$ and $n_e \in \mathbb{N}$ represent the entrance and the exit basic blocks of P respectively. An edge (n_i, n_j) exists in the edge set $\mathbb{E}$ if $n_i \in \mathbb{N}$ *may* transfer control to $n_j \in \mathbb{N}$. Each edge (n_i, n_j) has a label ℓ that specifies the logical predicate that is sufficient for n_i to transfer control to n_j.

Domination is a key concept in control flow analysis. Let n be any node. A node d dominates n, denoted d **dom** n, iff every path in G from n_s to n includes d. Furthermore, every node dominates itself. A node p post-dominates n, denoted p **pdom** n, iff every path in G from n to n_e includes p. For any node n other than n_s, the immediate dominator of n is the unique node d that strictly dominates n (i.e., d **dom** n and $d \neq n$)

but does not strictly dominate any other node that strictly dominates n. The immediate post-dominator of n is defined similarly.

Loops are defined through domination. An edge (s, d) is a *back edge* iff d **dom** s. Each back edge (s, d) defines a *natural loop*, whose header is d. The natural loop of a back edge (s, d) is the union of d and the set of nodes that can reach s without going through d.

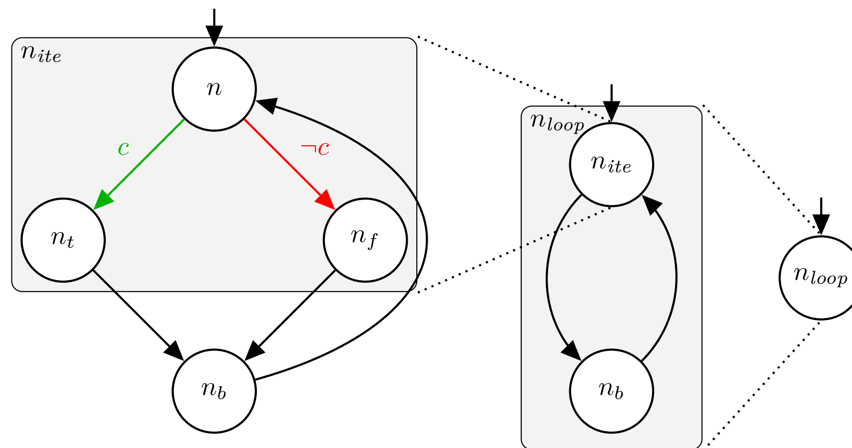


Figure 2.1: Example of structural analysis.

2.2.2 Structural Analysis

Structural analysis is a control flow structuring algorithm for recovering high-level control flow structure such as if-then-else constructs and loops. Intriguingly, such an algorithm has uses in both compilation (during optimization) and decompilation (to recover abstractions). At a high level, structural analysis matches a set of region *schemas* over the CFG by repeatedly visiting its nodes in post-order. Each schema describes the shape of a high-level control structure such as if-then-else. When a match is found, all nodes matched by the schema are *collapsed* or *reduced* into a single node that represents the schema matched. Figure 2.1 shows the progression of structural analysis on a simple example from left to right. In the initial (leftmost) graph, the top three nodes match the shape of an if-then-else. Structural analysis reduces these nodes into a single node that is explicitly labeled as an if-then-else region in the middle graph. This graph is then further reduced into a do-while loop. A decompiler would use this sequence of reductions and infer the control flow structure

```
do { if (c1) then {...} else {...} } while (c2);.
```

Once no further matches can be found, structural analysis reduces acyclic and cyclic subgraphs into *proper regions* and *improper regions*, respectively. Intuitively, both of these regions indicate that no high-level

structure can be identified in that subgraph and thus goto statements will be emitted to encode the control flow. A key topic of this paper is how to build a modern structural analysis algorithm that can *refine* such regions so that more high-level structure can be recovered.

2.2.3 SESS Analysis and Tail Regions

Vanilla structural analysis cannot recognize loops containing common C constructs such as break and continue. For instance, structural analysis would fail to structure the loop

```
while (...) { if (...) { body; break; } }.
```

Engel et al. [52] proposed the SESS (single exit single successor) analysis to identify regions that have multiple exits (using break and continue) but share a unique successor. Such exits can be converted into a *tail region* that represents the equivalent control flow construct. In the above example, body would be reduced to a break tail region. Without tail regions, structural analysis stops making progress when reasoning about loops containing multiple exits.

Although the SESS analysis was proposed to help address this problem, the core part of the algorithm, the detection of tail regions, is left unspecified [52, Algorithm 2, Line 15]. An initial implementation of Phoenix implemented SESS analysis as closely to the paper as possible, but SESS often stopped making progress before identifying a tail region. This can occur when regions have multiple candidate successors, or when loop bodies are too complex. Unfortunately, no structure is recovered for these parts of the program. This problem was the motivation for the iterative refinement of Phoenix's algorithm (Section 2.4).

2.3 Overview

Any end-to-end decompiler such as Phoenix is necessarily a complex system. This section aims to give a high-level overview of the major challenges of building a decompiler, and explain the resulting design choices of Phoenix that address those challenges.

Figure 2.2 shows the high-level overview of how Phoenix decompiles a target binary. Like most previous work, Phoenix uses a number of stages, where the output of stage i is the input to stage $i + 1$. Phoenix can fail to output decompiled source if any of its four stages fails. The first two stages are based on existing implementations. The last two use new techniques and implementations developed specifically for Phoenix. Although this section covers all components of Phoenix, later sections focus on Phoenix's new structural analysis algorithm.

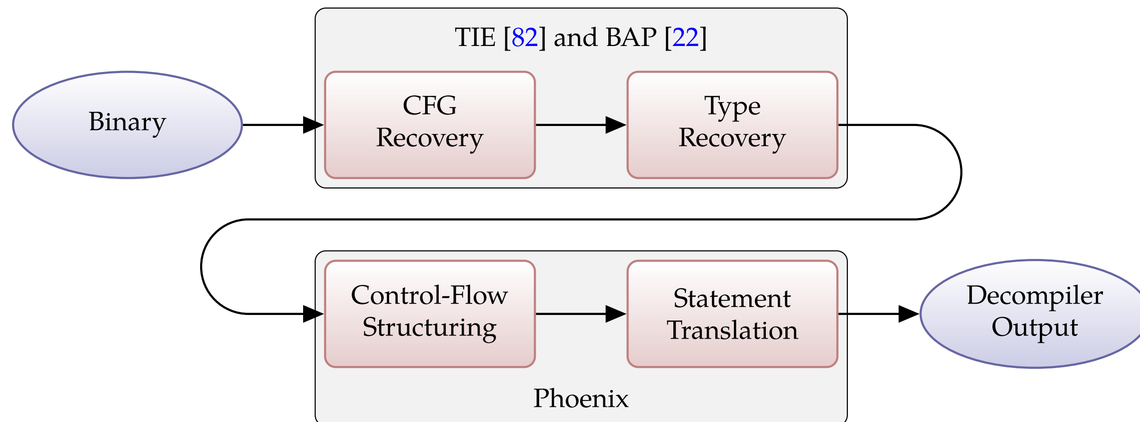


Figure 2.2: Overview of Phoenix's design.

2.3.1 Stages I and II—Existing Work

Control Flow Graph Recovery

The first stage parses the input binary's file format, disassembles the binary, and creates a control flow graph (CFG) for each function (Section 2.2). At a high level, a control flow graph is a program representation in which vertices represent basic blocks of sequential statements, and edges represent possible control flow transitions between blocks. While precisely identifying binary code in an executable is known to be hard in the general case [66], current algorithms have been shown to work well in practice [10,12,74,79].

There are mature platforms that already implement this stage. Phoenix uses the Binary Analysis Platform (BAP) [22]. BAP lifts sequential x86 assembly instructions in the CFG into an intermediate language called BIL, whose syntax is shown in Table 2.1. The end goal of Phoenix is to decompile this language into the high-level language shown in Table 2.2.

Variable and Type Recovery

The second stage recovers individual variables from the binary code, and assigns them types. Phoenix uses the existing TIE [82] system for this stage. TIE runs Value Set Analysis (VSA) [10] to recover variable locations, and then uses a static, constraint-based type inference system similar to the one used in the ML programming language [90]. Roughly speaking, each statement imposes some constraints on the type of variables involved. For example, an argument passed to a function that expects an argument of type T should be of type T, and the denominator in a division must be an integer and not a pointer. The constraints are then solved to assign each variable a type.

<table>
<tr><td>program</td><td>::=</td><td>stmt*</td></tr>
<tr><td>stmt</td><td>::=</td><td>var := exp | jmp(exp) | cjmp(exp, exp, exp)
| halt(exp) | assert(exp) | label(label_kind) | special(string)</td></tr>
<tr><td>exp</td><td>::=</td><td>load(exp, exp, exp, τ_{reg}) | store(exp, exp, exp, exp, τ_{reg}) | exp $\lozenge_b$ exp
| $\lozenge_u$ exp | var | lab(string) | integer | cast(cast_kind, τ_{reg}, exp)
| let var = exp in exp | unknown(string, τ)</td></tr>
<tr><td>label_kind</td><td>::=</td><td>integer | string</td></tr>
<tr><td>cast_kind</td><td>::=</td><td>unsigned | signed | high | low</td></tr>
<tr><td>var</td><td>::=</td><td>(string, id_v, τ)</td></tr>
<tr><td>$\lozenge_b$</td><td>::=</td><td>$+, -, *, /, /_s, \mathrm{mod}, \mathrm{mod}_s, \ll, \gg, \gg_a, \&, |, \oplus, ==, !=, <, \leq, <_s, \leq_s$</td></tr>
<tr><td>$\lozenge_u$</td><td>::=</td><td>$-$ (unary minus), $\sim$ (bit-wise not)</td></tr>
<tr><td>integer</td><td>::=</td><td>$n{:}\tau_{\text{reg}}$</td></tr>
<tr><td>τ</td><td>::=</td><td>$\tau_{\text{reg}} \mid \tau_{\text{mem}}$</td></tr>
<tr><td>τ_{mem}</td><td>::=</td><td>mem_t(τ_{reg}) | array_t(τ_{reg}, τ_{reg})</td></tr>
<tr><td>τ_{reg}</td><td>::=</td><td>reg_t(n)</td></tr>
</table>

Table 2.1: Binary analysis platform intermediate language (BIL).

2.3.2 Stage III—Control-Flow Structure Recovery

The next stage recovers the high-level control flow structure of the program. The input to this stage is a program with variables and types in CFG form. The goal is to recover high-level, structured control flow constructs such as loops, if-then-else and switch constructs from the graph representation. A program or construct is *structured* if it does not utilize gotos. Structured program representations are preferred because they help scale program analysis [91] and make programs easier to understand [46]. The process of recovering a structured representation of the program is sometimes called *control flow structure recovery* or *control flow structuring* in the literature.

Although *control flow structure recovery* is similar in name to *control flow graph recovery* (stage I), the two are very different. Control flow graph recovery starts with a binary program, and produces a control flow graph representation of the program as output. Control flow structure recovery takes a control flow graph representation as input, and outputs the high-level control flow structure of the program, for instance:

```
while (...) { if (...) {...} }.
```

The rest of this chapter will focus on control flow structure recovery and not control flow graph reconstruction.

Structural analysis is a control flow structure recovery algorithm that, roughly speaking, matches predefined graph schemas or patterns to the control flow constructs that create the patterns [91]. For example, if a structural analysis algorithm identifies a diamond-shape in a CFG, it outputs an if-then-else construct, because if-then-else statements create diamond-shaped subgraphs in the CFG.

prog	::=	(varinfo*, func*)
func	::=	(string, varinfo, varinfo, stmt*)
stmt	::=	var := exp | goto(exp) | if exp then stmt else stmt
		| while(exp, stmt) | dowhile(stmt, exp)
		| for(stmt, exp, stmt)
		| sequence(stmt*)
		| switch(exp, stmt*)
		| case(exp, stmt)
		| label(string)
		| nop

Table 2.2: High-level intermediate language (HIL).

However, using structural analysis in a decompiler can cause incorrect decompilation and miss opportunities for recovering structure. These problems motivated the new structural analysis algorithm in Phoenix which avoids these pitfalls. Phoenix's algorithm has two new features: (F1) iterative refinement, which recovers more structure than previous algorithms, and (F2) semantics-preserving schemas, which ensure the control flow of the decompiled program is consistent with the original. Phoenix's structuring algorithm is a major focus of this chapter, and is discussed in more detail in Section 2.4.

2.3.3 Stage IV—Statement Translation and Outputting C

The input to the next stage is a CFG annotated with structural information, which loosely maps each vertex in the CFG to a position in a control construct. What remains is to translate the BIL statements in each vertex of the CFG to a high-level language representation called HIL. Some of HIL's syntax is shown in Table 2.2.

Although most statements are straightforward to translate, some require information gathered in prior stages of the decompiler. For instance, to translate function calls, the translator uses VSA to find the offset of the stack pointer at the call site, and then uses the type signature of the called function to determine how many arguments to include. Phoenix also performs optimizations to make the final source more readable. There are two types of optimizations. First, similar to previous work, Phoenix performs optimizations to remove redundancy such as dead-code elimination [33]. Second, Phoenix contains *untiling* optimizations that improve readability.

During compilation, a compiler uses a transformation called *tiling* to reduce high-level program statements into assembly statements. At a high level, tiling takes as input an abstract syntax tree (AST) of the source language and produces an assembly program by covering the AST with semantically equivalent assembly statements. For example, given:

```
x = (y+z)/w;,
```

tiling would first cover the expression $y + z$ with the `add` instruction, and then the division with the `div` instruction. Tiling will typically produce many assembly instructions for a single high-level statement.

Phoenix uses an *untiling* algorithm to improve readability. Untiling takes several statements and outputs an equivalent high-level source statement. For instance, at a low-level, $\texttt{High}_1\,[a\&b]$ means to extract the most significant bit from bitwise-anding a with b. This may not seem like a common operation used in C, but it is equivalent to the high-level operation of computing $a <_s 0 \;\&\; b <_s 0$ (i.e., both a and b are less than zero when interpreted as signed integers). Phoenix uses about 20 manually crafted untiling patterns to simplify instructions emitted by `gcc`'s code generator. These patterns only improve the readability of the source output, and do not influence correctness or control-flow structure recovery.

The output of the statement translation phase is a HIL program. The final stage in Phoenix is to analyze this HIL program; in the abstraction recovery problem, this corresponds to the analysis function $\llbracket - \rrbracket$. In this chapter, the analysis translates HIL into C to test Phoenix as a traditional binary-to-C decompiler.

2.4 Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring

This section describes Phoenix's control flow structuring algorithm, which builds on the existing structural analysis algorithm by adding iterative refinement and semantics-preserving schemas.

2.4.1 Semantics Preservation

Structural analysis was originally invented to scale data flow analysis by summarizing the reachability properties of a program's CFG. Later, decompiler researchers adapted structural analysis and its predecessor, interval analysis, to recover the control flow structure of decompiled programs [35, 62]. Unfortunately, structural analysis can identify control flow that is consistent with a CFG's reachability, but is inconsistent with the CFG's semantics.

Such an error from structural analysis is demonstrated in Figure 2.3. Structural analysis would identify the loop in the leftmost graph and reduce it to a single node representing the loop, producing the middle diamond-shaped graph. This graph matches the schema for an if-then-else region, which would also be reduced to a single node. Finally, the two remaining nodes would then be reduced to a sequence node (not shown). This analysis would be correct for data flow analysis, which depends on reachability. However, the first reduction was not semantics-preserving. This is easy to see when both $x = 1$ and $y = 2$ hold. In the original graph, the first loop exit would be taken, since $x = 1$ matches the first exit edge's condition. However, in the middle graph, both exit edges can be taken.

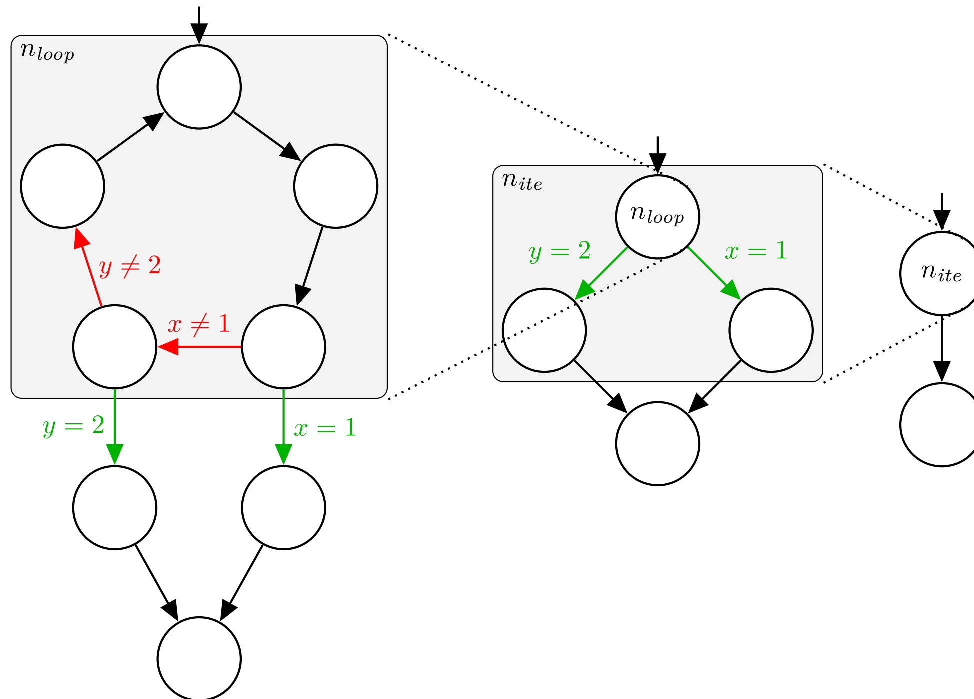


Figure 2.3: Structural analysis failing without semantics-preservation.

Such discrepancies are a problem in abstraction recovery, because they can unintentionally cause unsoundness in analyses. For example, a bug checker could state that a bug is present when applied to the structured program in Figure 2.3, even if the original program had no bugs.

To avoid this, a structural analysis algorithm should preserve the semantics of a CFG during each reduction. Otherwise the recovered control flow structure can become inconsistent with the actual control flow in the binary. Most schemas in structural analysis [91, p. 203] preserve semantics, but one of the schemas used in Figure 2.3, the natural loop schema, does not. Replacing these schemas with semantics-preserving schemas increased the number of utilities Phoenix correctly decompiled by 30% (Section 2.5).

2.4.2 Iterative Refinement

At a high level, *refinement* is the process of removing an edge from a CFG by emitting a goto in its place, and *iterative refinement* refers to the repeated application of refinement until structuring can progress. This may seem counter-intuitive, since adding a goto seems like it would *decrease* the amount of structure recovered. However, the removal of a carefully chosen edge can potentially allow a schema to match the refined CFG, thus enabling the recovery of additional structure. Structural analysis emits 30× more gotos (from 40 to 1,229) than Phoenix's iterative refinement-based algorithm (Section 2.5).

Recovering structure is important for several reasons. Structured code is easier for programmers to un-

derstand [46], and helps scale program analysis in general [91]. In addition, some analyses use syntactic patterns to find facts, which relies on effective structure recovery. For example, a bug checker might conclude that there is no buffer overflow in Figure 2.4 by syntactically discovering the induction variable i and loop invariant $i < 10$. If the structuring algorithm instead represents this loop using gotos, the bug checker might be unable to prove the loop is safe.

```
char b[10];
int i = 0;
while (i < 10) {
  b[i] = 0;
  i++;
}
```

Figure 2.4: Terminating loop.

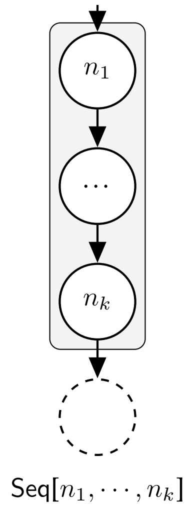
2.4.3 Algorithm Overview

As with vanilla structural analysis [91, p. 203], Phoenix's algorithm visits nodes in post-order in each iteration. Intuitively, this means that all descendants of a node will be visited (and hence had the chance to be reduced) before the node itself. When visiting node n, the algorithm first determines if the region at n is acyclic or cyclic. If n is part of an acyclic region, the algorithm tries to match the subgraph at n to one of the acyclic schemas (Section 2.4.4), and if that fails, attempts to refine the region at n into a switch region (Section 2.4.6). If n is cyclic, the algorithm instead compares the region at n to the cyclic schemas (Section 2.4.7). If this fails, it refines n into a loop (Section 2.4.8). If neither matching or refinement make progress, the current node n is then skipped for the current iteration of the algorithm. If there is an iteration in which every node is skipped, i.e., the algorithm stops making progress, then the algorithm employs a last resort refinement (Section 2.4.9) to ensure that progress can be made.

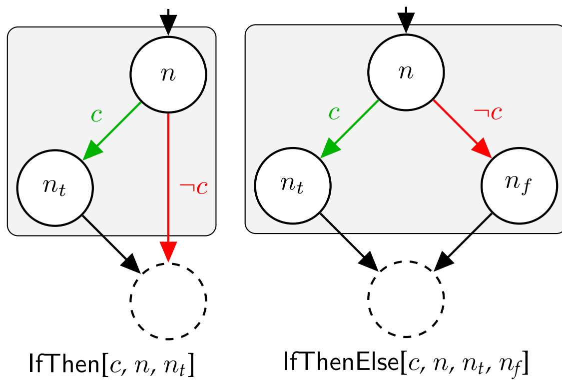
2.4.4 Acyclic Regions

The basic acyclic regions correspond to the acyclic control flow operators in C: sequences, conditionals, and switches. The schemas for these regions are shown in Figure 2.5. For example, the Seq[$n_1, \cdots, n_k$] region contains k regions that always execute in order. IfThenElse[c, n, n_t, n_f] denotes that n_t is executed when condition c holds; otherwise n_f is executed. Node n only contains the conditional branch on c.

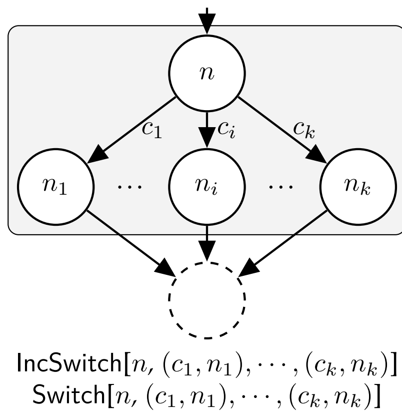
The schemas match both graph shape and edge conditions. Conditions are implicitly described using meta-variables such as c and $\neg c$. The intuition is that shape alone is not enough to distinguish which control structure should be used in decompilation. For instance, a switch for cases $x = 2$ and $x = 3$ can have the diamond shape of an if-then-else, but an if-then-else requires the outgoing conditions to be inverses.



A block of sequential regions that has a single predecessor and a single successor.



Regions corresponding to the if-then and if-then-else branching constructs.



Incomplete and complete switch regions. Outgoing conditions are pairwise disjoint. Complete switches have $\bigvee_{i \in [1,k]} c_i$ = true, and incomplete switches do not.

Figure 2.5: Acyclic region types.

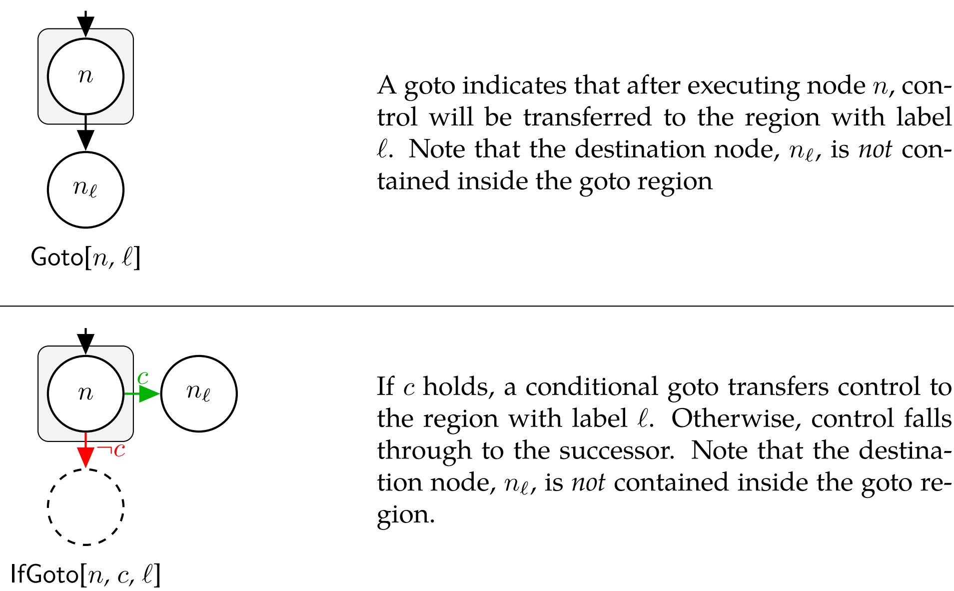


Figure 2.6: Tail regions.

2.4.5 Tail Regions and Edge Virtualization

Even if the subgraph at n does not match any known region type, it may be possible to *refine* the region. The insight behind *refinement* is that removing an edge from the CFG may allow a schema to match, and *iterative refinement* refers to the repeated application of refinement until a match is possible. Of course, each edge in the CFG represents a control flow, and refinement must account for removed edges in some other way to avoid changing the behavior of the program. Removing the edge in a way that preserves semantics is called *virtualizing* the edge, since the refined program behaves as if the edge was present, even though it is not.

Phoenix virtualizes an edge by collapsing the source node of the edge into a tail region, which explicitly denotes there should be a control transfer at the end of the region. Phoenix will select one of the tail regions shown in Figure 2.6 depending on whether the edge to be virtualized is unconditional or not. For instance, to virtualize the unconditional edge (n_1, n_2), Phoenix removes the edge from the CFG, inserts a fresh label ℓ at the start of n_2, and collapses n_1 to Goto[n_1, ℓ], which denotes there should be a `goto` l statement at the end of region n_1. Tail regions can also be translated into `break` or `continue` statements when used inside a switch or loop. Because the tail region explicitly represents the control flow of the virtualized edge, it is safe to remove the edge from the graph and ignore it in future pattern matches.

2.4.6 Switch Refinement

If the subgraph at node n fails to match one of the acyclic region types, it might be a switch candidate. *Switch candidates* are regions that would match a switch schema in Figure 2.5, but contain extra edges. For instance,

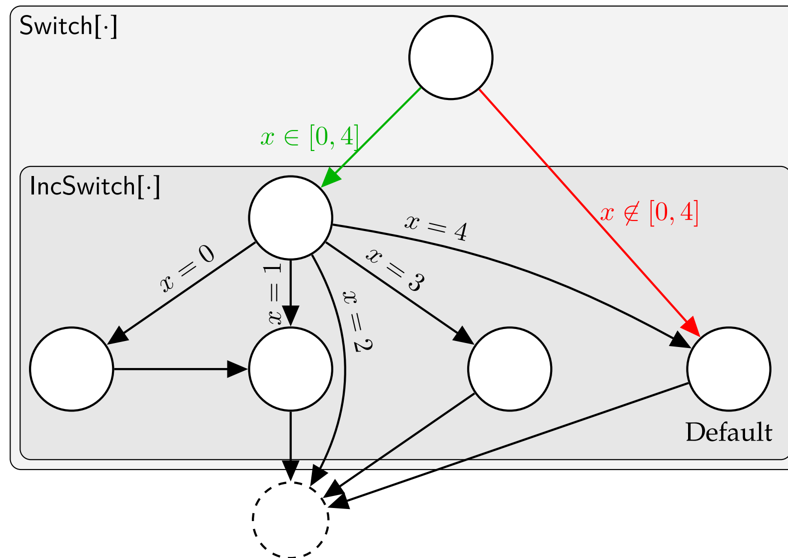


Figure 2.7: Relationship between complete and incomplete switches.

the nodes inside the IncSwitch[·] box in Figure 2.7 would not be identified as an IncSwitch[·] region because there is an extra incoming edge to the default case node.

A switch candidate is first refined by virtualizing incoming edges to any node besides the switch head. The next step is to ensure all switch nodes have the same successor. The immediate post-dominator of the switch head is selected as the successor if it is the successor of any of the case nodes. Otherwise, the node that (1) is a successor of a case node, (2) is not a case node itself, and (3) has the highest number of incoming edges from case nodes is chosen as the successor. After the successor has been identified, any outgoing edge whose destination is not the successor is virtualized.

After refinement, a switch candidate usually matches the IncSwitch[·] schema. For instance, a common implementation strategy for switches is to redirect inputs handled by the default case (e.g., $x > 4$) to a default node, and use a jump table for the remaining cases (e.g., $x \in [0, 4]$). This relationship is depicted in Figure 2.7, along with the corresponding region types. Because the jump table only handles a few cases, it is recognized as an IncSwitch[·]. However, because the default node handles all other cases, together they constitute a Switch[·].

2.4.7 Cyclic Regions

If the subgraph at node n is cyclic, the algorithm tries to match a loop at n to one of the cyclic schemas. It is possible for a node to be the loop header of multiple loops; for instance, nested do-while loops share a common loop header. Distinct loops at node n can be identified by finding back edges pointing to n (Section 2.2). Each back edge (n_b, n) defines a loop body consisting of the nodes that can reach n_b without

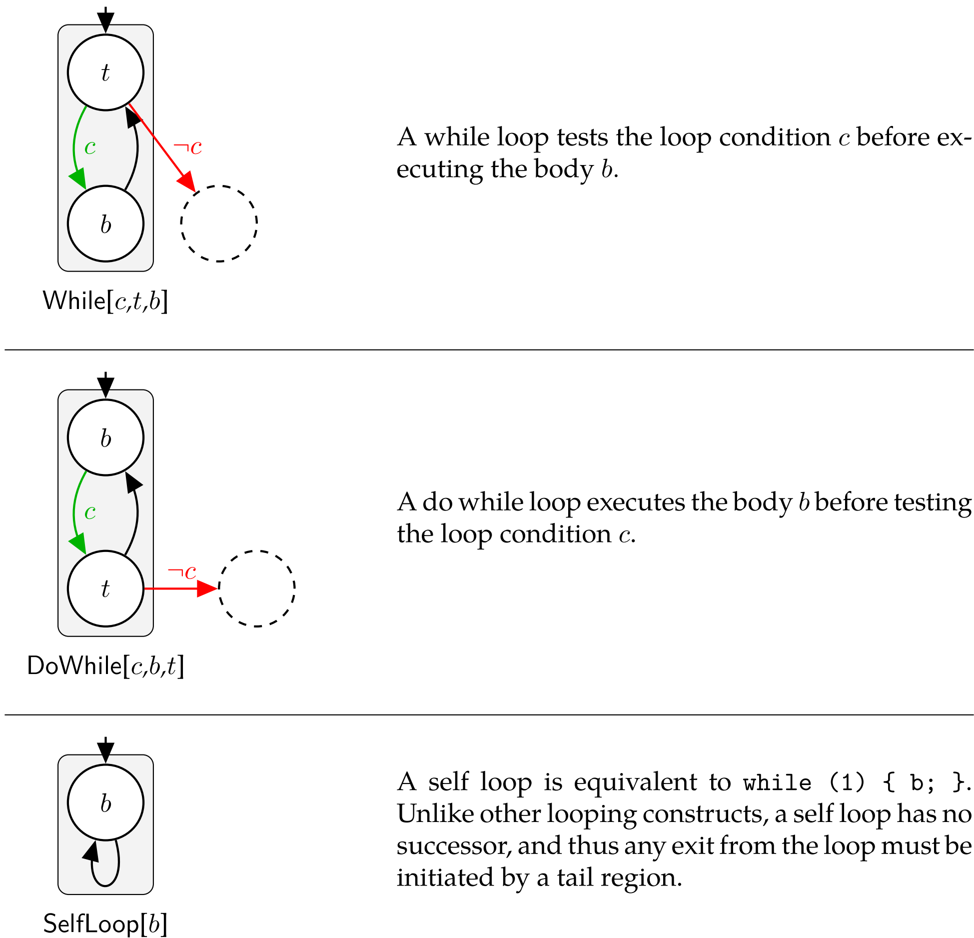


Figure 2.8: Cyclic region types.

going through the loop header, n. The loop with the smallest loop body is reduced first. This must happen before the larger loops can match the cyclic region patterns, because there is no schema for nested loops.

As shown in Figure 2.8, there are three types of loops. While[·] loops test the exit condition before executing the loop body, whereas DoWhile[·] loops test the exit condition after executing the loop body. If the exit condition occurs in the middle of the loop body, the region is a self loop. Self loops do not represent one particular C looping construct, but can be caused by code such as

```
while (1) { body1; if (e) break; body2; }.
```

Notice that the schema for self loops contains no outgoing edges from the loop. This is not a mistake, but is required for semantics-preservation. Because SelfLoop[·] regions are decompiled to

```
while (1) {...}.
```

which has no exits, the body of the loop must trigger any loop exits. In Phoenix, the loop exits are repre-

sented by a tail region, which corresponds to a `goto`, `break`, or `continue` in the decompiled output. These tail regions are added during loop refinement.

2.4.8 Loop Refinement

If any loops are detected with loop header n that do not match a loop schema, loop refinement begins. Cyclic regions may fail to match loop schemas because (1) there are multiple entrances to the loop, (2) there are too many exits from the loop, or (3) the loop body cannot be collapsed (i.e., is a proper region).

The first step of loop refinement is to ensure the loop has a single entrance (nodes with incoming edges from outside the loop). If there are multiple entrances to the loop, the one with the most incoming edges is selected, and incoming edges to the other entrances are virtualized.

The next step is to identify the type of loop. If there is an exit edge from the loop header, the loop is a While[·] candidate. If there is an outgoing edge from the source of the loop's back edge (Section 2.2), it is a DoWhile[·] candidate. Otherwise, any exit edge is selected and the loop is considered a SelfLoop[·] candidate. The exit edge determines the successor of the loop, i.e., the statement that is executed immediately after the loop. The successor in turn determines which nodes are lexically contained in the loop.

Phoenix virtualizes any edge leaving the lexically contained loop nodes other than the exit edge. Edges to the loop header use the `continue` tail regions, while edges to the loop successor use the `break` regions. Any other virtualized edge becomes a goto.

The first implementation of Phoenix used the standard structural analysis definition of the loop body to determine which nodes are lexically contained in the loop body [91]. However, this definition introduced `goto` statements when the original program had `break` statements, as in Figure 2.9(a). The `puts("c")` statement is *not* in the loop body according to the standard definition, because it cannot reach the loop's back edge, but it *is* lexically contained in the loop. A `break` statement must be lexically contained inside the loop body, or there is no loop to break out of.

Phoenix considers the nodes lexically contained in the loop to be the loop body *and* any nodes that execute after the loop body but before the successor. More formally, this corresponds to the loop body, and the nodes that are dominated by the loop header, excluding any nodes reachable from the loop's successor without going through the loop header. For example, `puts("c")` in Figure 2.9(b) is considered as a node that executes between the loop body and the successor, and thus Phoenix places it lexically inside the loop. When Phoenix uses the standard loop membership definition used in structural analysis, Phoenix outputs gotos, as in Figure 2.9(c). Phoenix's evaluation (Section 2.5) shows that the new loop membership definition decreased the numbers of gotos Phoenix emitted by 45% (73 to 40).

```
int f(void) {
  int a = 42;
  int b = 0;
  while (a) {
    if (b) {
      puts("c");
      break;
    } else {
      puts("d");
    }
    a--;
    b++;
  }
  puts("e");
  return 0;
}
```

(a) Original source code

```
t_reg32 f (void) {
  t_reg32 var_20 = 42;
  t_reg32 var_24;
  for (var_24 = 0; var_20 != 0;
  var_24 = var_24 + 1) {
    if (var_24 != 0) {
      puts("c");
      break;
    }
    puts("d");
    var_20 = var_20 - 1;
  }
  puts("e");
  return 0;
}
```

(b) Phoenix decompiled output of 2.9(a) *with* new loop membership definition

```
t_reg32 f (void)
{
  t_reg32 var_20 = 42;
  t_reg32 var_24;
  for (var_24 = 0;
  var_20 != 0; var_24 = var_24 + 1)
  {
    if (var_24 != 0) goto lab_1;
    puts("d");
    var_20 = var_20 - 1;
  }
  lab_2:
  puts("e");
  return 0;
  lab_1:
  puts("c");
  goto lab_2;
}
```

(c) Phoenix decompiled output of 2.9(a) *without* new loop membership definition

Figure 2.9: Loop refinement with and without new loop membership definition.

The last loop refinement step is to remove edges that may prevent the loop body from being collapsed. This can happen, for instance, when a programmer uses a goto in the body of a loop.

2.4.9 Last Resort Refinement

If the algorithm does not collapse any nodes or perform any refinement during an iteration, Phoenix removes an edge in the graph to allow it to make progress. This process is called the last resort refinement because it has the lowest priority, and always allows progress to be made. Last resort refinement prefers to remove

edges whose source does not dominate its target, nor whose target dominates its source. These edges can be thought of as cutting across the dominator tree. By removing them, the edges that remain reflect more structure.

2.5 Evaluation

This section describes a large quantitative evaluation of Phoenix on a suite of real programs, which demonstrates that the techniques employed by Phoenix lead to more correct decompilation and more recovered structure than the *de facto* industry standard Hex-Rays and other techniques. More specifically, Phoenix was able to decompile 114% more utilities that passed the entire coreutils test suite than Hex-Rays (60 vs 28), with Phoenix exhibiting a 30% (from 46 to 60) increase in correctness just by employing semantics-preserving schemas. Most remaining correctness errors in Phoenix have been attributed to the existing type recovery implementation (Section 2.6). Phoenix was also able to structure the control flow for 8,676 functions using only 40 gotos by employing iterative refinement, which corresponds to 30× more structure (40 gotos vs 1,229) than standard structural analysis.

2.5.1 Phoenix Implementation

Phoenix is implemented as an extension to the Binary Analysis Platform (BAP) [22]. Phoenix alone consists of 3,766 new lines of OCaml code which were added to BAP. Together, Phoenix and TIE comprise 8,443 lines of code. For reference, BAP consisted of 29,652 lines of code before any additions. The number of lines of code were measured using David A. Wheeler's `SLOCCount` utility.

2.5.2 Metrics

Phoenix was evaluated on the following two *quantitative* metrics:

Correctness Correctness measures whether the decompiled output is operationally equivalent to the original binary input. If a decompiler produces output that does not actually reflect the behavior of the input binary, it is of little utility in almost all settings. In the following experiments, high-coverage tests are used to approximate correctness.

Structuredness Recovering control flow structure helps program analysis and humans alike. Structured code is easier for programmers to understand [46], and helps scale program analysis in general [91]. Thus, decompiler output with fewer unstructured control flow commands such as `goto` is better.

The benefit of these metrics is that they can be evaluated quantitatively and thus can be automatically measured. These properties makes them suitable for an objective comparison of decompilers.

Existing Metrics

Note that these metrics are vastly different than those appearing in previous decompiler work. Cifuentes proposed using the ratio of the size of the decompiler output to the initial assembly as a "compression ratio" metric [33]:

$$1 - (\text{LOC decompiled}/\text{LOC assembly}).$$

The underlying idea is that humans can better understand compact decompiler output. However, the compression ratio metric side-steps several other important factors, such as whether the decompilation is correct or even compilable. A significant amount of previous work has proposed no metrics, and instead observed that the decompiler produced "reasonable" output, or had a manual qualitative evaluation on a few small examples [29,33,55,56,125].

2.5.3 Coreutils Experiment Overview

The Phoenix experiments were performed on the GNU coreutils 8.17 suite of utilities. coreutils consists of 107 mature, standard programs used on almost every Linux system.[2] The coreutils suite also has a suite of high-coverage tests that can be used to measure correctness. Though prior work has studied individual decompiler components on a large scale (Section 2.7), this evaluation on coreutils is an order of magnitude larger than any other systematic end-to-end decompiler evaluation in which specific metrics were defined and measured.

Tested Decompilers

In addition to Phoenix, the academic decompiler Boomerang [125], and the industrial decompiler Hex-Rays [62] were also tested.[3] Other decompilers such as REC [106], DISC [48], and dcc [33] were considered as well, but these compilers either produced pseudo-code (e.g., REC), did not work on x86 (e.g., dcc), or did not have any documentation that suggested advancements beyond Boomerang (e.g., DISC).

Boomerang and Hex-Rays suffered serious problems in their default configurations. First, Boomerang failed to produce any output for all but a few coreutils programs. Boomerang would get stuck while de-

[2]The number of utilities built depends on the machine that coreutils is compiled on. This is the number applicable to our testing system, which ran Ubuntu 12.04.1 x86-64. We compiled coreutils in 32-bit mode because the current Phoenix implementation only supports 32-bit binaries.

[3]The latest publicly available version of Boomerang, version 0.3, and the latest Hex-Rays version at the original time of writing, 1.7.0.120612, was tested.

compiling one function, and would never move on to other functions. There appeared to be no easy or reasonable fix to enable some type of per-function timeout mechanism. Boomerang is also no longer actively maintained. Second, Hex-Rays did not output compliant C code. In particular, Hex-Rays output non-standard C types and idioms that only Visual Studio recognized, and caused almost every function to fail to compile with `gcc`. The Hex-Rays manual [65] states:

> [...] the produced code is not supposed to be compilable and many compilers will complain about it. This is a deliberate choice of not making the output 100% compilable because the goal is not to recompile the code but to analyze it.

Even if Hex-Rays output is intended to be analyzed rather than compiled, it should still be correct modulo compilation issues. After all, there is little point to pseudo-code if it is semantically incorrect.

Because Hex-Rays was the only decompiler tested that actually produced output for real programs, the experimenter investigated the issue in more detail and noticed that the Hex-Rays output was only uncompilable because of the Visual Studio idioms and types it used. In order to offer a conservative comparison of Phoenix to existing work, the experimenter wrote a post-processor for Hex-Rays that translates the Hex-Rays output to compliant C. The translation is extremely straightforward. For example, one translation transforms types such as `unsigned __intN` to `uintN_t`.[4] All experiments are reported with respect to the post-processed Hex-Rays output. This translation is intended to make the comparison more fair: without post-processing, Hex-Rays does not output valid C.

2.5.4 Coreutils Experiment Procedure

Testing decompilers on real programs is difficult, because even the best decompiler cannot decompile every function. This observation rules out the simple strategy of decompiling every function in a binary, compiling the output source code, and testing the resulting binary. However, it should be possible to test the functions that can be decompiled, and this is the motivation of the substitution method.

Substitution Method

The substitution method produces a binary that is *recompiled* from a combination of original and decompiled source code. The implementation uses CIL [94] to produce a C file for each function in the original source code, and then compiles each C file to a separate object file. Object files for each function emitted by the decompiler are produced in a similar manner. An initial recompiled binary is created by linking all of the

[4] Although it seems like this should be possible to implement using only a C header file containing some `typedef` s, a `typedef` has its qualifiers fixed. For instance, `typedef int t` is equivalent to `typedef signed int t`, and thus the type `unsigned t` is not allowed because `unsigned signed int` is contradictory.

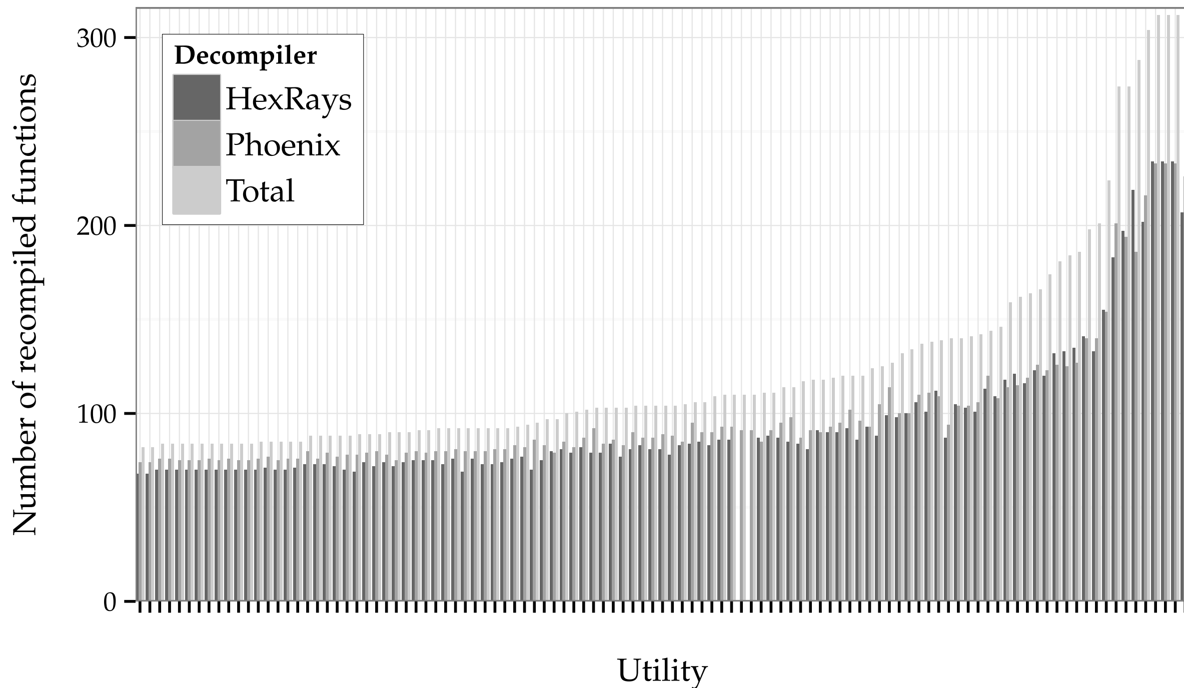


Figure 2.10: Recompilability measurements from the coreutils experiment. The number of functions successfully decompiled and recompiled by each decompiler is shown, organized by utility (names not shown). Hex-Rays failed on two utilities for unknown reasons.

original object files (i.e., object files compiled from the original source code) together to produce a binary. The main idea of the substitution method is to iteratively substitute a decompiler object file (i.e., object files compiled from the decompiler's output) for its corresponding original object file. If linking this new set of object files succeeds without an error, the decompiler object file is used in future iterations; otherwise, the original object file is used. The substitution method was used to produce a recompiled binary was produced for each decompiler and coreutils utility combination.

Of course, for fairness, the recompiled binaries for each decompiler should have approximately the same number of decompiled functions, since non-decompiled functions use the original function definition from the coreutils source code, which presumably passes the test suite and is well-structured. The number of recompilable functions output by each decompiler is depicted by utility in Figure 2.10. Phoenix recompiled 10,756 functions in total, compared to 10,086 functions for Hex-Rays. The Phoenix recompiled binaries consist of 82.2% decompiled functions on average, whereas the Hex-Rays binaries contain 77.5%. This puts Phoenix at a slight disadvantage, since it uses fewer original functions. Hex-Rays did not produce output after running for several hours on the `sha384sum` and `sha512sum` utilities. Phoenix did not completely fail on any utilities, and was able to decompile 91 out of 110 functions (82.7%) for both `sha384sum` and `sha512sum`. (These two utilities are similar). Phoenix's limitations and failure modes are discussed further in Section 2.6.

Decompiler	Passing utilities	Percentage recompiled
Phoenix (with semantics preservation)	60	85.4%
Phoenix (without semantics preservation)	46	—
Hex-Rays	28	73.8%

Table 2.3: Correctness summary for the coreutils experiment. These results includes two utilities for which Hex-Rays recompiled zero functions (thus trivially passing correctness).

Correctness

To measure correctness properly, a recompiled utility is tested by running the coreutils test suite with that recompiled utility and *original* versions of the other utilities. It is important to use the original versions of other utilities because the coreutils test suite is self-hosting, or uses its own utilities to set up the tests. For instance, if a test for `mv` used `mkdir` to setup the test, and if the recompiled version of `mkdir` fails, the test might accidentally blame `mv` for the failure, or worse, incorrectly report that `mv` passed the test when in reality it was not properly set up.

Each tested utility can either pass all tests, or fail. Alternatives such as counting the number of failed tests are not meaningful because many utilities have only one test that exercises them. As an extreme example, one recompiled utility crashed on every execution and yet only failed a single test. It would be misleading to suggest this recompiled program performed well because it failed "only" one test.

Table 2.3 reports the results of the correctness tests. Hex-Rays recompiled 28 utilities that passed the coreutils test suite, but Phoenix was able to recompile 60 passing utilities (114% more). It is important that these utilities are not simply correct because they mostly consist of original coreutils functions. This is not the case for Phoenix: the recompiled utilities that passed all tests consisted of 85.4% decompiled functions on average, which is actually higher than the overall Phoenix average of 82.2%. The correct Hex-Rays utilities consisted of 73.8% decompiled functions, which is less than the overall Hex-Rays average of 77.5%. As can be seen in Figure 2.10, this is partly because Hex-Rays completely failed on two utilities. The recompiled binaries for these utilities consisted completely of the original source code, which unsurprisingly passed all tests. Excluding those two utilities, Hex-Rays only compiled 26 utilities that passed the tests, and these utilities consisted of 79.4% decompiled functions on average.

Phoenix was also evaluated with the standard standard structural analysis schemas, including those that are *not* semantics-preserving, in order to evaluate their effect on correctness. Phoenix produced only 46 correct utilities with these schemas. This 30% reduction in correctness (from 60 down to 46) illustrates the importance of using semantics-preserving schemas.

Decompiler	Total gotos
Phoenix	40
Phoenix (without loop membership)	73
Phoenix (without refinement)	1,229
Hex-Rays	51

Table 2.4: Structuredness summary of the coreutils experiment. The statistics only reflect the 8,676 recompilable functions output by both Phoenix and Hex-Rays.

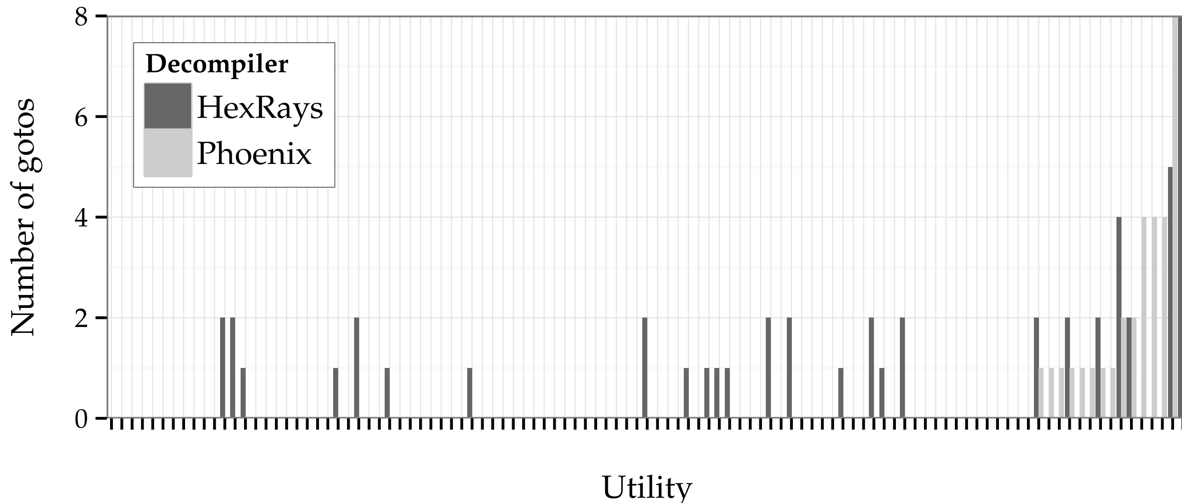


Figure 2.11: Structuredness measurements from the coreutils experiment. The number of gotos emitted by each decompiler is shown, organized by utility (names not shown). Only functions that were decompiled and recompiled by both decompilers are counted.

Structuredness

The amount of structure recovered by each decompiler was measured by simply counting the number of `goto` statements emitted by each decompiler. To ensure a fair comparison, only the intersection of recompilable functions emitted by both decompilers, which consisted of 8,676 functions, was considered. Doing otherwise would penalize a decompiler for outputting a function with `goto` statements, even if the other decompiler could not decompile that function at all.

The overall structuredness results are depicted in Table 2.4, with the results broken down per utility in Figure 2.11. In summary, Phoenix recovered the structure of the 8,676 considered functions using only 40 gotos, but recovered significantly less structure when either refinement (1189 more gotos) or the new loop membership definition (33 more) was disabled. These results suggest that structuring algorithms without iterative refinement [52,91,117] recover significantly less structure, and that Hex-Rays employs a technique similar to iterative refinement.

2.6 Limitations and Future Work

2.6.1 BAP Failures

Phoenix uses BAP [22] to lift instructions to a simple language that is then analyzed. BAP does not have support for floating point and other exotic types of instructions. Phoenix refuses to decompile any function that contains instructions unknown to BAP. Phoenix also relies on some analyses in BAP, such as value set analysis (VSA) and CFG recovery, and if these analyses fail then decompilation cannot proceed. Although these types of failures are rare, they can cascade to affect other functions. For instance, if CFG recovery for function g fails and function f calls g, Phoenix cannot decompile function f because it must know the type of g to emit a function call.

2.6.2 Correctness Failures

Because Phoenix is designed for abstraction recovery and program analysis, it should be correct. The Phoenix experiments show that, although Phoenix significantly improves over prior work with respect to correctness, Phoenix's output is not always correct. Most correctness errors in Phoenix have been manually attributed to the underlying type recovery component, TIE [82]. Many of the problems described below only became apparent when TIE was stress-tested by Phoenix.

Iterative Variable Recovery

TIE does not always identify all local variables. For instance, if function `f` takes a pointer to an integer, and a function calls `f(x)`, then TIE infers that `x` is a subtype of a pointer to an integer, but does *not* automatically infer that `*x`, the locations that `x` can point to, are potentially integer variables. TIE does not recover such variables because it would need to discover variables, generate and solve type constraints iteratively to do so. Unfortunately, undiscovered variables can cause incorrect decompilation. For example, if a struct on the stack is undiscovered, space for the struct is never allocated, which causes the called function to read and overwrite other variables on the stack of the callee. This is the leading cause of correctness errors in Phoenix. It is future work to investigate running type recovery until the set of known variables reaches a fix point.

Calling Conventions

TIE currently assumes that all functions use the `cdecl` calling convention, and does not support multi-register (64-bit) return values. Unfortunately, this can make Phoenix output incorrect or uncompilable code.

Using an interprocedural liveness analysis to automatically detect calling conventions based on the behavior of a function and the functions that call it is future work. The ultimate goal is to detect and understand calling conventions automatically, even when they are non-standard. This is important for analyzing malware, some of which uses unusual calling conventions to try to confuse static analysis.

Recursive Types

TIE has no support for recursive types, which are commonly used in data structures such as linked lists and binary trees. This means that TIE would infer the type

```
struct s {int a; struct s *next;}
```

as

```
struct s {int a; void* next;},
```

which does not specify what type of element `next` points to. Since Phoenix is intended to recover abstractions, it should recover the most specific type possible. Effectively recovering recursive types in a new type inference algorithm is future work.

2.7 Related Work

At a high level, there are three lines of work relevant to Phoenix. First, work in end-to-end decompilers. Second, work in control structure recovery, such as loop identification and structural analysis. Third, literature pertaining to type recovery.

2.7.1 Decompilers

The earliest work in decompilation dates back to the 1960's. For an excellent and thorough review of the literature in decompilation and several related areas up to around 2007, see Van Emmerik's thesis [125, ch. 2]. Another in-depth overview is available online [44].

Modern decompilers typically trace their roots to Cifuentes' 1994 thesis on dcc [33], a decompiler for 80286 to C. The structuring algorithm used in dcc is based on interval analysis [3]. Cifuentes proposed the compression ratio metric (Section 2.5.2), but did not measure correctness on the ten programs that dcc was tested on [34]. Since compression is the target metric, dcc outputs assembly if it encounters code that it cannot handle. Cifuentes et al. have also created a SPARC asm to C decompiler, and measured compressibility and the number of recovered control structures on seven SPEC1995 programs [36]. Again, they did not

test the correctness of the decompilation output. Cifuentes [33] also pioneered the technique of recovering short-circuit evaluations in compound expressions (e.g., `x && (!y || z)` in C).

Chang et al. [29] also use compressibility in their work on cooperating decompilers for the three programs they tested. Their main purpose was to show they can find bugs in the decompiled source that were known to exist in the binary, which is an example of abstraction recovery. However, correctness of the decompilation itself was not verified.

Boomerang is a popular open-source decompiler started by Van Emmerik as part of his Ph.D. [125]. The main idea of Van Emmerik's thesis is that decompilation is easier on the Single Static Assignment (SSA) form of a program. In his thesis, Van Emmerik's experiments are limited to a case study of Boomerang coupled with manual analysis to reverse engineer a single 670KB Windows program. Boomerang was tested in this chapter's evaluation (Section 2.5), but it failed to produce any output on all but a few test cases after running for several hours.

The structuring algorithm used in Boomerang first appeared in Simon [118], who in collaboration with Cifuentes proposed a new algorithm known as "parenthesis theory". Simon's own experiments showed that parenthesis theory is faster and more space efficient than the interval analysis-based algorithm in dcc, but recovers less structure.

Hex-Rays is the *de facto* industry decompiler. The most detailed write-up available on Hex-Rays is a 2008 paper [62] by Hex-Rays' author, Guilfanov, who revealed that Hex-Rays also performs structural analysis. However, Hex-Rays achieves much better structuredness than vanilla structural analysis, which suggests that Hex-Rays is using a heavily modified version. There are many other binary-to-C decompilers such as REC [106] and DISC [48]. However, our experience suggests that they are not as advanced as Hex-Rays.

Phoenix is a binary to C decompiler. Other researchers have investigated decompiling executables from managed languages such as Java [89]. The set of challenges they face are fundamentally different. On the one hand, these managed binaries contain extra information such as types; on the other hand, recovering the control flow itself in the presence of exceptions and synchronization primitives is a difficult problem.

2.7.2 Control Structure Recovery

Control structure recovery is also studied in *compilation*. This is because by the time compilation is in the optimization stage, the input program has already been parsed into a low-level intermediate representation (IR) in which the high-level control structure has been destroyed. Much work in program optimization therefore attempts to recover the control structures.

The most relevant line of work in this direction is the elimination methods in data flow analysis (DFA),

pioneered by Allen [3] and Cooke [38] in the 1970's and commonly known as "interval analysis". Sharir [117] subsequently refined interval analysis into structural analysis. In Sharir's own words, structural analysis can be seen as an "unparser" of the CFG. Besides the potential to speed up DFA even more when compared to interval analysis, structural analysis can also cope with irreducible CFGs.

Engel et al. [52] are the first to extend structural analysis to handle C-specific control statements. Specifically, their Single Entry Single Successor (SESS) analysis adds a new tail region type, which corresponds to the statements that appear before a `break` or `continue`. For example, suppose

```
if (b) { body; break; }
```

appears in a loop, then the statements represented by `body` would belong to a tail region. Engel et al. have extensively tested their implementation of SESS in a source-to-source compiler. However, their SESS analysis does not use iterative refinement, and can get stuck on unstructured code. We show in our evaluation that this leads to a large amount of structure being missed. Their exact algorithm for detecting tail regions is also left unspecified [52, Algorithm 2, Line 15].

Another line of related work lies in the area of program schematology, of which "Go To Statement Considered Harmful" by Dijkstra [46] is the most famous.

2.7.3 Type Recovery

Besides control structure recovery, a high-quality decompiler should also recover the types of variables. Much work has gone into this recently. Phoenix uses TIE [82], which recovers types statically. In contrast, REWARDS [86] and Howard [119] recover types from dynamic traces. Other work has focused on C++-specific issues, such as virtual table recovery [55,56,72].

2.8 Conclusion

This chapter presented Phoenix, a new binary-to-C decompiler designed to accurately and effectively recover abstractions. Phoenix can help leverage the wealth of existing source-based tools and techniques in scenarios where the original source code is unavailable. Phoenix uses a new control flow structuring algorithm that avoids a previously unpublished correctness pitfall in decompilers, and uses iteratively refinement to recover more control flow structure than existing algorithms. Phoenix and the *de facto* industry standard decompiler, Hex-Rays, were evaluated on correctness and amount of control flow structure recovered. Phoenix decompiled twice as many utilities correctly as Hex-Rays, and recovered slightly more structure.

Chapter 3

Forward Verification Conditions

This chapter introduces a versatile program analysis primitive called the verification condition, which can be used in conjunction with abstraction recovery. Researchers have developed many types of analyses using verification conditions, from automatic exploitation [7,8,28,112] and automatic test case generation [26,60] to extended static checking [53,84]. In addition to enabling a multitude of applications, verification condition algorithms accept a variety of program representations, which has allowed researchers to apply them to both abstract [7,26,53] and concrete [8,28,112] program representations.

The ability to analyze both source and binary code is rare among program analyses, and offers a unique opportunity to quantify some of the benefits of abstraction recovery, in support of the thesis that abstraction recovery scales better than equivalent analyses that do not recover abstractions. The high level idea is to build analyses using verification conditions, and then to evaluate these analyses on (1) original source code, (2) compiled binary code, and (3) source code abstractions recovered by Phoenix (Chapter 2). Chapter 5 reports the results of these experiments, which at a high level show that VC-based algorithms can analyze abstract source code more quickly than low-level binary code.

More specifically, this chapter describes a new algorithm for computing verification conditions, FVC, which is specifically designed for use with abstraction recovery. FVC combines techniques from several different approaches to building VCs, including those that utilize abstractions. For example, FVC can simplify computations that do not depend on user input, but this requires variable and type abstractions to be maximally effective.

3.1 Introduction

The ability to automatically generate a logical formula that captures the correctness properties of a program fragment is a fundamental primitive in program verification. Such formulas are called *Verification Condi-*

tions (VCs). VCs reduce properties about programs to logical formulas; by demonstrating validity of these formulas, one can prove facts about the program. VCs can also be solved to enable other applications, such as test-case generation. Regardless of application, previous work in VC generation is largely split into two approaches: forward symbolic execution and weakest preconditions.

3.1.1 Forward Symbolic Execution

Forward Symbolic Execution [75] (FSE) has become a staple technique for generating VCs for individual program paths. FSE enables sound and complete reasoning about all executions on a selected path, and thus is attractive for path-oriented applications [11,59,73,105,114] such as test-case generation. The FSE algorithm "symbolically executes" program statements along a program path to construct a logical formula—the *path predicate* π. Whenever the algorithm encounters a statement that reads user input, it introduces a fresh symbolic variable (instead of a concrete value from the user). Any assignment to input variables that satisfies π will drive execution down the same execution path. Although FSE is path-oriented, it can be used to produce a VC for the entire program by symbolically executing each path and taking the disjunction of the resulting path predicates.

Modern FSE implementations [11, 27, 59, 73, 105, 114] have capitalized on the forward nature of the FSE algorithm by including optimizations such as infeasible path pruning and concrete execution. Infeasible path pruning avoids symbolically executing program paths whose constraints are unrealizable. Concrete execution allows the processor to directly execute program statements that do not involve symbolic user input, instead of adding them to the formula. It also helps cope with unmodeled system behavior in real programs (e.g., system calls or floating point operations), by enabling *concretization*, the process of substituting concrete semantics in place of (unknown) symbolic behavior.

FSE can also dynamically unroll loops until all feasible executions exit the loop. This is useful for programs which are known to terminate, but for which the exact number of loop executions is unknown. For example, consider the program in Figure 3.1, which asserts that the sum of the even numbers in $(0, 10]$ is equal to the user's input. By dynamically unrolling the while loop, and concretely executing the first four statements of the program, FSE will generate the VC: $30 = input$.

```
int c = 10, sum = 0;
while (c > 0) {
  if (c%2 == 0) sum += c;
  c--;
}
assert (sum == input);
```

Figure 3.1: Simple loop.

3.1.2 Weakest preconditions

Weakest preconditions [13, 47, 54, 83] (WP) are an alternative approach to VC generation and traditionally run in the backward direction. As a result, existing WP algorithms do not employ forward optimizations such as concrete evaluation or dynamic loop unrolling. To verify the program in Figure 3.1 using WP, the user could manually unroll the loop n times, and WP would produce a VC that described 2^n paths, even though only one of those paths is feasible. Even a modern WP algorithm [54] would produce a VC that is over 70KB in size, which is huge compared to the FSE VC, which is simply $30 = \mathit{input}$. These extra constraints can slow down a theorem prover or constraint solver, especially when user input only affects a small portion of the program.

The backwards nature of WP is also an obstacle for verifying programs with loops. WP implementations cannot dynamically unroll loops. Instead, loops can be unrolled a fixed number of times before the WP algorithm runs. However, verification can fail when loops are not unrolled enough times, e.g., when the loop is unrolled fewer than 10 times in the above example. Alternatively, a loop approximation can be used if a loop invariant is known [54, 61].

Despite these shortcomings, modern WP algorithms can produce VCs that are $O(n^2)$ in size for programs with n statements [13, 54]. In contrast, every branch can double the number of program paths encoded in a formula generated by FSE, so FSE produces a $O(2^n)$-sized VC in the worst case. FSE often performs better than the worst case, as the example in Figure 3.1 demonstrates, which explains why it is still used in practice.

WP algorithms are defined on a language called the Guarded Command Language (GCL), which was proposed by Dijkstra for specifying and verifying programs [47]. The GCL can model both non-deterministic and partial programs, which enables programs to be modularly verified one function at a time. Although the GCL is a structured language (i.e., does not represent goto statements), recent work has extended the WP algorithm to operate on an unstructured graph representation of the GCL [13]. FSE is defined on a more restrictive deterministic, total language.

3.1.3 The best of WP and FSE

WP and FSE offer very different trade-offs: compact WP algorithms offer better theoretical size guarantees and can enable modular verification, but FSE enables forward optimizations and analyses that are effective in practice. This chapter introduces the Forward Verification Condition (FVC), a new algorithm which combines the best features from both WP *and* FSE. FVC produces formulas that are provably compact, but runs in the forward direction. FVC is defined on the GCL (and the unstructured GCL [13]), which allows modular verification, and verification of non-deterministic programs. This chapter also demonstrates how

common FSE optimizations such as concrete evaluation and dynamic loop unrolling can be integrated with FVC, and that this leads to faster verification times than either FSE or WP alone.

Contributions The main contribution of this chapter is a new VC algorithm, FVC, that combines the best features of existing FSE and WP algorithms. FVC is defined on a language allowing non-determinism and modular verification, produces provably compact formulas, and runs in the forward direction. FVC is compatible with forward optimizations and analyses, such as concrete path pruning, feasibility testing and dynamic loop unrolling. Appendix B includes a formal proof in Isabelle/HOL [98] showing that FVC produces VCs which are logically equivalent to those produced by existing WP algorithms. Last, an evaluation of FVC on a series on benchmarks shows that generating and solving VCs using FVC is 6× faster than existing algorithms.

3.2 Background

$$\begin{array}{lll} \text{stmt } S & ::= & \mathsf{skip} \mid x := e \mid \mathsf{assert}\ e \mid \mathsf{assume}\ e \mid S_1 \ ; S_2 \mid S_1 \ \square \ S_2 \\ \text{program} & ::= & S \end{array}$$

Figure 3.2: Guarded command language (GCL).

The language considered in this chapter is a variant of the Guarded Command Language (GCL) [47,95], and is shown in Figure 3.2. The language includes skip, assert and assume, assignment (:=), sequence (;) and non-deterministic choice (□) statements. Only the assume and choice statements are explained here, as the rest are self-explanatory. assume e is equivalent to skip when e is true, and simply does not execute otherwise (i.e., the program is only defined when e holds). It can be used to mark e as a precondition for running the program. $S_1 \ \square \ S_2$ means to non-deterministically execute either S_1 or S_2. This allows a GCL program to have multiple outcomes for a single input. The language is acyclic, but adding loops is discussed in the next section (Section 3.3.6).

The GCL might appear to be missing an if-then-else statement. However, the if-then-else statement $\mathsf{ite}(e, S_1, S_2)$ can be represented in GCL as

$$\mathsf{ite}(e, S_1, S_2) \equiv (\mathsf{assume}\ e \ ; S_1) \ \square \ (\mathsf{assume}\ \neg e \ ; S_2). \tag{ITE}$$

Because deterministic branches are relatively common, $\mathsf{ite}(e, S_1, S_2)$ will be used as a short-hand for the equivalent GCL program.

S	when	FSE σ π S next
skip	next = skip	π
skip	–	FSE σ π next skip
$x := e$	–	FSE $\sigma[x \mapsto \sigma\, e]$ π next skip
assert e	–	FSE σ $(\pi \wedge \sigma\, e)$ next skip
ite(e, S_1, S_2)	–	FSE σ $(\pi \wedge \sigma\, e)$ S_1 next $\vee$ FSE σ $(\pi \wedge \sigma\, \neg e)$ S_2 next
S_1 ; S_2	–	FSE σ π S_1 $(S_2$; next$)$

Figure 3.3: Forward symbolic execution (FSE).

3.2.1 Forward Symbolic Execution

Forward Symbolic Execution (FSE) is a path-based algorithm for computing VCs in the forward direction. FSE executes the program as normal, but with symbolic variables introduced in place of concrete input. Subsequent execution builds symbolic expressions for each program variable, which are stored in a variable context/mapping, σ. FSE also builds a path predicate π; the invariant maintained is that any assignment to symbolic input variables which satisfies π will cause the program to proceed down the current path without failing an assertion.

The basic FSE algorithm is shown in Figure 3.3. FSE takes four arguments: the mapping of variables to their symbolic expressions σ (initially empty), the path predicate π (initially true), the program to be symbolically executed S, and the next statement to be executed (initially skip).

Figure 3.3 and other descriptions of algorithms in this chapter use a pattern matching notation. The basic idea is to match the argument values (e.g., S in Figure 3.3) to the patterns in the respective columns, starting from the first row and proceeding downwards. The first match in which the *when* constraint is either satisfied or unspecified (–) lists the code to be evaluated in the rightmost column. The FSE algorithm also uses some notation for the σ mapping. $\sigma\, e$ denotes evaluating the expression e in the variable mapping σ. For example, $x * x$ evaluates to $5 * 5$ in the mapping $\{x \mapsto 5\}$, and to $y * y$ in the mapping $\{x \mapsto y\}$. $\sigma[x \mapsto e]$ returns the mapping σ with an additional binding of x to expression e.

FSE is known to have disadvantages:

- FSE cannot be used to generate VCs for the full GCL. In particular, $S_1 \;\square\; S_2$ and assume e statements may only be used to form an ite(e, S_1, S_2) statement (see Equation ITE).

- FSE processes any statements following an ite(e, S_1, S_2) statement on *both* branches, which is equivalent to forking at each branch. This allows FSE to create a formula one path at a time, but causes exponential blowup when producing a formula for the entire program.

S	$\mathsf{WP}\ S\ Q$	$\mathsf{WLP}\ S\ Q$
skip	Q	$\mathsf{WP}\ S\ Q$
$x := e$	$Q[x \mapsto e]$	$\mathsf{WP}\ S\ Q$
$\mathsf{assert}\ e$	$e \wedge Q$	$e \implies Q$
$\mathsf{assume}\ e$	$e \implies Q$	$\mathsf{WP}\ S\ Q$
$S_1 \square S_2$	$\mathsf{WP}\ S_1\ Q\ \wedge\ \mathsf{WP}\ S_2\ Q$	$\mathsf{WP}\ S\ Q$
$S_1\ ;\ S_2$	$\mathsf{WP}\ S_1\ (\mathsf{WP}\ S_2\ Q)$	$\mathsf{WP}\ S\ Q$

Figure 3.4: Weakest preconditions (WP) and weakest liberal preconditions (WLP).

3.2.2 Dijkstra's Weakest Preconditions

The weakest precondition (WP) [47] of a program S and postcondition Q is a predicate that is true for all pre-states that, after executing S, lead to successful termination in a post-state satisfying Q. Dijkstra was the first to give an algorithm for computing the weakest precondition of a GCL program.

Dijkstra's WP algorithm is shown in Figure 3.4. $Q[x \mapsto e]$ denotes Q with free instances of x substituted with e. The WP algorithm has a few known disadvantages:

- Duplication of the postcondition Q causes exponential blowup when it is used on both branches of $S_1 \square S_2$ statements. For example, Q is clearly duplicated in

$$\mathsf{WP}\ (x := 4 \square x := 5)\ Q = Q[x \mapsto 4] \wedge Q[x \mapsto 5],$$

  but it is not obvious how to merge the parts of Q that are unrelated to x.

- The sequence (;) rule for WP shows it is a backwards algorithm. When processing $S_1\ ;\ S_2$ statements, the algorithm first recurses on $\mathsf{WP}\ S_2\ Q$, and uses this as the input postcondition for S_1.

Normal and Abnormal Termination Sometimes it is useful to characterize the states from which a program will (1) terminate normally, or (2) terminate in error. $\mathsf{WP}\ S\ \mathsf{true}$ by definition characterizes the states from which S will terminate normally. However, to describe the states from which S will terminate *abnormally*, a variant of WP known as the *weakest liberal precondition* (WLP) is needed. The WLP of a program S and postcondition Q is a predicate that is true for all pre-states that, after executing S, (1) terminate in error (i.e., fail an assertion) or (2) terminate successfully in a post-state satisfying Q. Thus, $\mathsf{WLP}\ S\ \mathsf{false}$ describes the program executions that always terminate abnormally. Functionally, the algorithm for computing the WLP only differs from WP on $\mathsf{assert}\ e$, as shown in Figure 3.4.

3.2.3 Verification Conditions and the Connection to Hoare Logic

Correctness properties have traditionally been described in terms of Hoare tuples [61]. A Hoare tuple {P}S{Q} states that Q will hold after executing S from a state where P holds, and is valid if and only if $P \implies \mathsf{WP}\, S\, Q$.

Modern VC algorithms take a different approach to specifying correctness properties. A VC of a GCL program S is a logical formula that is valid if and only if all defined executions of S terminate without failing an $\mathsf{assert}\, e$ statement. Instead of specifying preconditions P and postconditions Q for the entire program, VC algorithms expect correctness conditions to be denoted inside the program via $\mathsf{assume}\, P$ and $\mathsf{assert}\, Q$ statements. Unlike Hoare tuples, these statements can exploit locality to concisely describe local properties. A Hoare tuple {P}S{Q} can still be expressed as

$$S' = \mathsf{assume}\, P \;;\; S \;;\; \mathsf{assert}\, Q.$$

Because Q is encoded in S', WP can be used to generate a verification condition with $\mathsf{WP}\, S'\, \mathsf{true}$.

3.2.4 Passification

Modern WP algorithms [13, 54] use a two stage algorithm for computing provably compact VCs. The first step is to remove all side-effects from the program (e.g., assignments) in a process called *passification*. The second step is to use the passified program to compute the VC. Passification is helpful because it makes it possible to merge state at confluence points without duplication. For example, one of the weaknesses of WP is that Q is duplicated at branches:

$$\mathsf{WP}\, (x := 4 \,\square\, x := 5)\, Q = Q[x \mapsto 4] \wedge Q[x \mapsto 5].$$

Passification solves this problem by eliminating assignments from the program, which allows the same definition of Q to be shared by both branches.

Flanagan and Saxe proposed the first algorithm [54] for passifying an acyclic GCL program, called passify. Conceptually, passify consists of two steps: (1) variable renaming, and (2) assignment rewriting. During variable renaming the program is rewritten to ensure that each variable is assigned to at most once. This is equivalent to putting the program in static single assignment (SSA) form [40]. This translation produces a quadratic increase in program size in the worst case, though in practice the increase is close to linear [54]. The second step is to remove assignments by converting each $v := e$ statement to an $\mathsf{assume}\, v = e$ statement.[1] The FVC algorithm introduced in the next section assumes that variable renaming has already been

[1] Note that this is only correct when querying for validity. For more details, see Section 3.5.

performed, but explicitly performs the assignment rewriting step. This allows optimizations such as concrete evaluation (Section 3.3.4) to distinguish between assignments and assumptions.

Leino summarized the advantages of passified programs in a theorem [83]:

Theorem 3.1 (Leino)**.** For all passified GCL programs P:

$$\mathsf{WP}\ P\ Q = \mathsf{WP}\ P\ \mathsf{true} \wedge (\mathsf{WLP}\ P\ \mathsf{false} \vee Q).$$

At a high-level, Leino's theorem states that for passified programs, the bulk of the WP computation can be done independently of the postcondition Q. Flanagan and Saxe first proposed an efficient WP algorithm [54] based on this observation.[2]

3.3 Forward Verification Conditions

S	MS S	MF S
skip	true	false
$x := e$	MS (assume $x = e$)	MF (assume $x = e$)
assert e	true	$\neg e$
assume e	e	false
$S_1 \square S_2$	MS $S_1 \vee$ MS S_2	MF $S_1 \vee$ MF S_2
$S_1 ; S_2$	MS $S_1 \wedge$ (MF $S_1 \vee$ MS S_2)	MS $S_1 \wedge$ (MF $S_1 \vee$ MF S_2)

Figure 3.5: Predicates of the forward verification conditions (FVC) algorithm.

This section presents Forward Verification Conditions (FVC). FVC is motivated in part by existing modern WP algorithms [13, 54, 83], which produce provably compact VCs. Unfortunately, these existing algorithms run in the backwards direction, which prevents them from applying standard FSE optimizations, and makes them difficult to apply in practice. FVC is a new formulation of WP that avoids these problems by running in the forward direction.

The FVC algorithm consists of three steps. The first step is passification (Section 3.2.4). The second step is to compute two predicates, MS and MF, that jointly summarize the behavior of the program. The intuition for these predicates is that MS S describes the condition from which S *may start* (i.e., meet all assumptions on at least one path), and MF S computes the states from which S *may fail* an assertion. MF corresponds to the negation of the path predicate π of FSE. In fact, FVC only needs to compute MS when processing partial programs (Section 3.3.3). Figure 3.5 provides the definitions for MS and MF. Note that the rule for $S_1 ; S_2$

[2]Leino actually published his paper [83] after Flanagan and Saxe's paper [54] to explain the high-level theorem their algorithm is based on.

recurses on S_1 before S_2, which makes the algorithm a forward one.[3] The final step is to combine MS S and MF S with the postcondition Q to compute the weakest precondition (using Leino's theorem):

$$\mathsf{FVC}\; S\; Q \equiv \Big(\mathsf{MS}\; S \implies (\neg \mathsf{MF}\; S \wedge Q)\Big) \tag{FVC}$$

3.3.1 Correctness

The intuition for why FVC is correct is given in the following theorem, which connects MS and MF to WP and WLP (Section 3.2.2):

Theorem 3.2 (Connection)**.** For all passified GCL statements S,

$$\mathsf{MS}\; S = \neg(\mathsf{WP}\; S\; \mathsf{true} \wedge \mathsf{WLP}\; S\; \mathsf{false})$$

and

$$\mathsf{MF}\; S = \neg \mathsf{WP}\; S\; \mathsf{true}.$$

The full correctness and size proofs are given in Appendix B.

Using the connection theorem, it is then possible to show that Equation FVC is invoking Leino's theorem to compute the weakest precondition:

Theorem 3.3 (Correctness)**.** For all passified GCL statements S and postconditions Q, $\mathsf{FVC}\; S\; Q = \mathsf{WP}\; S\; Q$.

3.3.2 Formula Size

The presentation of the FVC algorithm in Figure 3.5 and Equation FVC is foremost intended to be understandable. Unfortunately, the presented version duplicates expressions when MS and MF recurse on the same values. For example, MS $(S_1 \; ; \; S_2)$ and MF $(S_1 \; ; \; S_2)$ both recurse on MS S_1 and MF S_1.

This duplication can be eliminated by implementing MS and MF as a single recursive function that returns a tuple. This allows each expression e (e.g., $e = \mathsf{MS}\; S_1$) that is duplicated in both MS and MF to be deduplicated by replacing e with a fresh variable v, and then adding an equality binding $v = e$ to the start of the formula VC:[4]

$$\forall v.\; (v = e) \implies VC.$$

With this final amount of duplication removed, FVC $S\; Q$ is $O(|S| + |Q|)$ in size for all passified programs S. Because passification adds a quadratic increase in code size in the worst case [54], the final size theorem is:

[3] Nothing requires the algorithm to proceed in the forward direction. However, as shown in this chapter, there are many advantages to doing so.

[4] The equivalent form $\exists v.\; (v = e) \wedge VC$ is also possible for checking satisfiability.

S	TOTAL S
skip	true
assert e	true
assume e	false
ite(e, S_1, S_2)	TOTAL $S_1 \wedge$ TOTAL S_2
$S_1 \ \square \ S_2$	false
$S_1 \ ; \ S_2$	TOTAL $S_1 \wedge$ TOTAL S_2

Figure 3.6: Definition of total GCL programs.

Theorem 3.4 (Size)**.** For all GCL statements S and postconditions Q, FVC $S\,Q$ is $O(|S|^2 + |Q|)$ in size.

3.3.3 Total Programs

The GCL is a rich language that can model partial programs, which enables functions to be verified modularly. However, some programs are total, i.e., they are defined for every possible input. FVC reduces to a simpler algorithm when applied to total programs. The high-level idea is that total programs are always defined, and thus always start, making MS always true. FVC does not need to compute MS for such programs, which cuts the formula size in half. More specifically, the following FVC definition applies:

$$\mathsf{FVC}^{\mathsf{T}}\, S\, Q \equiv (\neg \mathsf{MF}\ S \wedge Q). \qquad \text{(FVC-TOTAL)}$$

Figure 3.6 gives the precise definition of the GCL programs for which this optimization is correct. At a high level, these programs are restricted and may only use assume e to form an ite(e, S_1, S_2) statement, which guarantees that at least one branch will execute in every program execution.

Theorem 3.5 (Correctness 2)**.** For all passified GCL statements S and postconditions Q,

$$\mathsf{TOTAL}\ S \implies \left(\mathsf{FVC}^{\mathsf{T}}\, S\, Q = \mathsf{WP}\ S\, Q\right).$$

3.3.4 Concrete Evaluation and Concrete Path Pruning

Concrete evaluation allows FVC to execute program fragments not involving symbolic input directly on the processor. This has two benefits. First, it simplifies some expressions all the way to concrete values. These simplified values can be used in the output formula in place of the original expressions. This frees the theorem prover from reasoning about the concrete behavior of the program. The second advantage of concrete evaluation is that it enables concrete path pruning. Concrete path pruning is the process of short-circuiting VC computation on program fragments that can never execute. The idea behind path pruning is simple: if e concretely evaluates to false, then any statement following assume e or assert e will never be executed, and these statements need not be considered.

S	when	$\mathsf{MS}^{\mathsf{C}}\ \sigma\ S$
skip	–	σ, true
$x := e$	$\sigma\ e$ = Concrete v	$\mathsf{MS}^{\mathsf{C}}\ \sigma[x \mapsto v]$ (assume $x = v$)
$x := e$	$\sigma\ e$ = Symbolic _	$\mathsf{MS}^{\mathsf{C}}\ \sigma[x \mapsto \bot]$ (assume $x = e$)
assert e	–	σ, true
assume e	$\sigma\ e$ = Concrete false	σ, false
assume e	$\sigma\ e$ = Symbolic _	σ, e
$S_1 \square S_2$	–	let σ_1, MS_1 = $\mathsf{MS}^{\mathsf{C}}\ \sigma\ S_1$ in let σ_2, MS_2 = $\mathsf{MS}^{\mathsf{C}}\ \sigma\ S_2$ in $\sigma_1 \cap \sigma_2$, $\mathsf{MS}_1 \vee \mathsf{MS}_2$
$S_1\ ;\ S_2$	$\left(\mathsf{MS}^{\mathsf{C}}\ \sigma\ S_1\right) = (\sigma_1,\ \text{false})$	σ_1, false
$S_1\ ;\ S_2$	$\left(\mathsf{MF}^{\mathsf{C}}\ \sigma\ S_1\right) = (\sigma_1,\ \text{true})$	σ_1, true
$S_1\ ;\ S_2$	–	let σ_1, MS_1 = $\mathsf{MS}^{\mathsf{C}}\ \sigma\ S_1$ in let _, MF_1 = $\mathsf{MF}^{\mathsf{C}}\ \sigma\ S_1$ in let σ_2, MS_2 = $\mathsf{MS}^{\mathsf{C}}\ \sigma_1\ S_2$ in σ_2, $\mathsf{MS}_1 \wedge (\mathsf{MF}_1 \vee \mathsf{MS}_2)$

Figure 3.7: MS^{C}: MS predicate with concrete evaluation and concrete path pruning.

FVC can be extended with concrete evaluation by adding a concrete variable mapping σ that maps variables to their current concrete values. Figure 3.7 shows MS^{C}, the MS algorithm extended with concrete evaluation. The algorithm for MF^{C} is omitted to save trees, but the idea is similar.

The MS^{C} algorithm takes a statement S and a concrete context σ as input, and outputs an updated context σ' and the MS predicate. Each concrete assignment is stored in the context σ to help simplify expressions in subsequent statements. The notation for describing operations on the context σ is reused (Section 3.2.1). At assume e statements, the algorithm checks if e concretely evaluates to false, and simplifies the result accordingly. At $S_1 \square S_2$ statements, the algorithm must merge the concrete mappings σ_1 and σ_2 from both branches by taking the intersection. For instance, if $x \mapsto 4$ in one branch and $x \mapsto 5$ in the other, x will have no concrete mapping in the merged context, and will be included in the formula.

Path pruning occurs at $S_1\ ;\ S_2$ statements. S_1 is executed first, but if it does not start (MS $S_1 = \text{false}$), or if it may fail (MF $S_1 = \text{true}$),[5] then the value of MS $S_1\ ;\ S_2$ and MF $S_1\ ;\ S_2$ do not depend on S_2, and thus S_2 does not need to be executed. In all other cases, S_2 is executed and the result is computed normally.

3.3.5 Feasibility testing

Feasibility testing is a more powerful version of concrete path pruning. Instead of using concrete evaluation to determine whether a path is feasible, online calls to an automated theorem prover are used. This requires the FVC implementation to be written such that intermediate values of MS and MF are available at branch

[5] Because MS $S \wedge$ MF $S =$ MF S.

points, which is straightforward. The same pruning opportunities for concrete path pruning are then applied: if the theorem prover determines that S may not start or may fail, then any subsequent statement does not need to be executed.

3.3.6 Loops

WP and FSE algorithms have traditionally handled loops in two different ways. WP algorithms usually approximate loops using loop invariants. A $\mathsf{while}\ e\ \mathsf{do}\ \{I\}\ S$ statement, which executes the statement S as long as the condition e is true and maintains the loop invariant I, can be approximated in GCL as [54]:

$$\mathsf{assert}\, I\ ;\ x_1 := y_1\ ;\ \ldots\ ;\ x_n := y_n\ ;\ \mathsf{assume}\, I\ ;$$
$$\Big((\mathsf{assume}\, e\ ;\ S\ ;\ \mathsf{assert}\, I\ ;\ \mathsf{assume\ false}) \,\Box\, \mathsf{assume}\, \neg e \Big),$$

where x_i are the variables assigned in S, and y_i are fresh variables. The encoding asserts that I holds before the loop, and then sets the assigned variables in S to arbitrary values that satisfy the loop invariant. If the loop exits, control is passed to the next statement. Otherwise S is executed and the loop invariant I is again checked. The effectiveness of this approach largely depends on the strength of the given loop invariants; a weak loop invariant can prevent a correct program from being verified.

In contrast, FSE implementations often unroll loops until all feasible executions exit the loop. This can be done by applying the following rule:

$$\mathsf{while}\ e\ \mathsf{do}\ \{I\}\ S \equiv (\mathsf{assume}\, e\ ;\ S\ ;\ \mathsf{while}\ e\ \mathsf{do}\ \{I\}\ S) \,\Box\, (\mathsf{assume}\, \neg e).$$

If the branch that stays in the loop is infeasible, concrete pruning or feasibility testing can prevent the loop from being unrolled further. A major disadvantage of dynamic loop unrolling is that it may not terminate. An additional complication is that the unrolled program must remain passified. In practice, this means that the variable renaming step of passification (Section 3.2.4) must be called each time loops are unrolled.

3.3.7 Unstructured FVC algorithm

Converting programs to the GCL is a challenge for using WP algorithms in practice, because the GCL is a structured language. Programmers can use unstructured programming constructs, such as goto, break, and continue statements, to create programs that cannot be converted to structured languages like GCL without duplication. An alternative program representation is the *control flow graph* (CFG), which can represent unstructured programs without duplication.

Barnett and Leino proposed an unstructured weakest preconditions (UWP) algorithm [13] to allow verification of programs that are in an unstructured form similar to a CFG. Specifically, UWP extends the GCL

so that (1) programs are split into labeled blocks of GCL statements, and (2) goto *block** statements are added. The goto *block** statement non-deterministically transfers control to one of the listed blocks, effectively replacing $S_1 \square S_2$ statements as the mechanism for non-determinism and branching. Unlike $S_1 \square S_2$ statements, the goto *block** statement does *not* require program execution to merge after branching, which is what enables verification of unstructured programs.

The basic UWP algorithm is to compute an $\mathcal{A}_{OK}$ variable for each block $\mathcal{A}$:

$$\mathcal{A}_{OK} = \mathsf{WP}\ S \bigwedge_{\mathcal{B} \in Succ(\mathcal{A})} \mathcal{B}_{OK}$$

where $Succ(\mathcal{A})$ denotes the successors of $\mathcal{A}$ in the control flow graph. Intuitively, $\mathcal{A}_{OK}$ describes the program states from which executions *starting from* block $\mathcal{A}$ are correct. This is an inherently backwards algorithm.

The idea behind UWP can be applied to construct an unstructured FVC algorithm, $\mathsf{FVC}^{\mathsf{U}}$. The basic idea is to define $\mathcal{A}_{\mathsf{MS}}$ and $\mathcal{A}_{\mathsf{MF}}$ variables for each block $\mathcal{A}$, intuitively corresponding to the program states from which executions *up to and including* $\mathcal{A}$ may start, or may fail, respectively. These variables depend on the corresponding values of their predecessors. More specifically,

$$\mathcal{A}_{\mathsf{MS}} = \mathcal{P}_{\mathsf{MS}} \wedge (\mathcal{P}_{\mathsf{MF}} \vee \mathsf{MS}\ \mathcal{A})$$

$$\mathcal{A}_{\mathsf{MF}} = \mathcal{P}_{\mathsf{MS}} \wedge (\mathcal{P}_{\mathsf{MF}} \vee \mathsf{MF}\ \mathcal{A})$$

where $\mathcal{P}_{\mathsf{MS}} = \bigvee_{P' \in Pred(\mathcal{A})} P'_{\mathsf{MS}}$, $\mathcal{P}_{\mathsf{MF}} = \bigvee_{P' \in Pred(\mathcal{A})} P'_{\mathsf{MF}}$ and $Pred(\mathcal{A})$ denotes the predecessors of $\mathcal{A}$. Computation is performed in the forward direction. Note that the $\mathcal{A}_{\mathsf{MS}}$ and $\mathcal{A}_{\mathsf{MF}}$ equations are derived from the $S_1 ; S_2$ rule of FVC, and $\mathcal{P}_{\mathsf{MS}}$ and $\mathcal{P}_{\mathsf{MF}}$ are derived from the $S_1 \square S_2$ rule.

Adding additional extensions such as concrete evaluation and path pruning to the $\mathsf{FVC}^{\mathsf{U}}$ algorithm is straightforward, and our implementation, which is benchmarked in the next section, contains these optimizations.

3.4 Evaluation

This section evaluates the performance of FVC compared to existing VC algorithms.

3.4.1 Experiment Setup

Five VC generation algorithms were implemented in the Binary Analysis Platform (BAP) [22]: (1) Forward Symbolic Execution (FSE) [75], (2) Dijkstra's Weakest Preconditions (WP) [47], (3) Flanagan and Saxe's Weakest Preconditions (FS) [54], (4) Unstructured Weakest Preconditions (UWP) [13], and (5) Unstructured Forward Verification Conditions (with concrete path pruning) ($\mathsf{FVC}^{\mathsf{U}}$) (Section 3.3.7). FSE is implemented di-

rectly on the BAP intermediate language (BIL). For all other VCs, conversion to GCL is necessary, which is counted towards that algorithm's formula generation time.

All experiments were performed on an Amazon EC2 `m3.2xlarge` instance, which features 8 hardware threads on an Intel Xeon E5-2670 processor with 30GB of RAM. All queries were limited to 60 minutes CPU time (for formula generation and solving), 2GB of disk space and 2GB of memory usage. VCs were tested for validity using the `yices` [51] SMT solver.

3.4.2 Benchmarks

The benchmark suite includes: (1) standard algorithms (`bubble` sort, `binary` search, `fibonacci`, `prime` testing, `summation`), (2) programs inspired from NECLA Static Analysis Benchmarks [93] (`ex18`, `ex30`, `ex39`, and `inf3`), and (3) benchmarks inspired from the Synergy project [63] (`barber`, `berkeley`, `cars`, and `efm`). Table 3.1 lists the correctness condition tested in each benchmark. Additionally, the source code of each benchmark can be found in Appendix C. Benchmarks were selected to cover a variety of behaviors, in order to stress test the weaknesses of existing VC algorithms. For instance, programs with many execution paths are difficult for FSE, whereas programs with large amounts of concrete computation are challenging for WP algorithms.

Each benchmark was manually translated to a subset of C, which was then compiled to the BAP language BIL using a custom CIL [94] module. The conversion between CIL and BIL is mostly straightforward, except for pointers. Each pointer reference is statically resolved to a concrete object using the alias analysis built into CIL. This approach was sufficient to convert each program in the benchmark suite, although it may not work for more complex programs. Loops in the benchmarks were unrolled prior to formula generation (the exact number of loop unrolls is specified in the following sections), and this time is not counted in any following results. Executions that execute a loop more times than it was unrolled are assumed to be correct, since otherwise almost all benchmarks would be trivially incorrect.

Program	Postcondition	Expected Result
berkeley	The error condition specified in the original benchmark did not occur.	Valid
barber	The error condition specified in the original benchmark did not occur.	Valid
binary	The output is the index of the specified element.	Valid
bubble	The output is sorted.	Invalid
cars	The error condition specified in the original benchmark did not occur.	Valid
efm	The error condition specified in the original benchmark did not occur.	Valid
ex18	Array accesses are within bounds.	Valid
ex30	The x and w arrays are equal.	Invalid
ex39	The error condition specified in the original benchmark did not occur.	Valid
fib	The computed value is the correct element in the fibonacci sequence.	Valid
inf3	The error condition specified in the original benchmark did not occur.	Valid
prime	The chosen value is correctly identified as a prime or composite number.	Valid
sum	The computed sum is always less than 200.	Invalid

Table 3.1: Benchmarks programs tested in correctness experiments.

Program	#U	OLOC	ULOC	FSE		WP		FS		UWP		FVC^U	
				Time	Ω×	Time	Ω×	Time	Ω×	Time	Ω×	Time	Ω×
berkeley	31	5,105	136,591	∞	∞	∞	∞	**17.7**	**0**×	33.1	0.9×	18	0.01×
barber	8	15,634	113,254	∞	∞	∞	∞	18.9	0.2×	22.5	0.5×	**15.5**	**0**×
binary	11	3,896	28,571	∞	∞	∞	∞	24.8	0.3×	33.8	0.8×	**18.7**	**0**×
bubble	4	5,235	80,852	∞	∞	∞	∞	∞	∞	**11.2**	**0**×	16.4	0.5×
cars	11	6,054	25,943	∞	∞	∞	∞	60.5	2×	47	1×	**23.4**	**0**×
efm	16	8,631	95,466	∞	∞	∞	∞	22.2	0.5×	20.5	0.4×	**15.1**	**0**×
ex18	56	4,354	195,543	∞	∞	∞	∞	117	9×	116	9×	**11.7**	**0**×
ex30	16	5,928	84,873	∞	∞	∞	∞	203	3×	205	3×	**47.5**	**0**×
ex39	236	2,216	210,897	**14.4**	**0**×	60.6	3×	∞	∞	254	17×	26.4	0.8×
fib	51	4,514	79,699	17.9	0.5×	**12.3**	**0**×	∞	∞	∞	∞	407	32×
inf3	136	4,086	333,212	∞	∞	∞	∞	∞	∞	94.2	8×	**10.3**	**0**×
prime	11	4,932	132,836	∞	∞	∞	∞	∞	∞	85.3	4×	**16.8**	**0**×
sum	101	1,738	116,243	∞	∞	∞	∞	19.3	0.7×	15.7	0.4×	**11.6**	**0**×
Total				39632	61×	39673	61×	18484	28×	4539	6×	**638**	**0**×

Table 3.2: Verification times (in seconds) for each VC algorithm. #U shows the number of loop unrolls; OLOC and ULOC show the lines of code of the program before and after unrolling respectively; the Ω× column indicates how many times slower that program is than the best VC for that program. Timeouts are denoted by ∞, and best times are shown in bold.

3.4.3 Generation & Solving Time

This section compares the performance of each VC algorithm on a fixed program size. For each benchmark, loops were unrolled until *all* algorithms for that benchmark required at least ten seconds to verify. Table 3.2 shows the verification times for the tested programs. The columns include: (1) the name of the benchmark, (2) the number of times loops were unrolled, (3) the size in characters of the original and unrolled program in BIL, and (4) total verification times for each of the tested VC algorithms. All verification times are accompanied by an overhead metric (Ω), which shows how many times slower the algorithm performed relative to the best algorithm for that benchmark. For example, for the ex18 benchmark, verification with FS has an

Ω of $9\times$, which means that verification takes nine times longer than the fastest algorithm ($\mathsf{FVC}^{\mathsf{U}}$). The fastest algorithm always has an Ω of $0\times$.

The table shows that different benchmarks favor different algorithms. For instance, FSE and WP perform very well on `ex39` and `fib`, but timeout for all others. These two programs have few realizable paths and perform concrete computations. In contrast, FS and UWP struggle because they cannot exploit the concrete computations and reason about many infeasible paths. $\mathsf{FVC}^{\mathsf{U}}$ outperforms FS and UWP on these benchmarks because it can simplify the concrete computations. The other benchmarks have many feasible paths, which benefits VC algorithms that generate compact formulas (FS, UWP, and $\mathsf{FVC}^{\mathsf{U}}$).

$\mathsf{FVC}^{\mathsf{U}}$ achieves the best performance on 9 out of 13 benchmarks. However, even when $\mathsf{FVC}^{\mathsf{U}}$ is not the best, it achieves performance consistently close to the best—$\mathsf{FVC}^{\mathsf{U}}$ only takes more than twice as long as the best algorithm on one benchmark (`fib`). This consistency is also reflected in the fact $\mathsf{FVC}^{\mathsf{U}}$ has no timeouts in the table. This observation highlights one of the main strengths of FVC: it is able to combine benefits from both forward (concrete pruning) and backward (compact formulas) running algorithms, thus allowing it to achieve good performance on a variety of benchmarks.

In total, $\mathsf{FVC}^{\mathsf{U}}$ took 638 s to verify all of the benchmarks, compared to UWP and FS which required 4,539 s (6× slower) and 18,484 s (28×) respectively. This is true even though the verification time for timeouts was conservatively counted as equal to the timeout period (60 minutes).

3.4.4 Scalability

The previous section showed that $\mathsf{FVC}^{\mathsf{U}}$ took the least amount of time to verify the benchmarks for a fixed program size. The immediate follow-up is: how does $\mathsf{FVC}^{\mathsf{U}}$ scale as the size of the program increases, and how does this compare with other algorithms? This question is addressed by measuring each algorithm's performance as the number of loop unrolls increases. Figure 3.8 summarizes the results of this experiment.

A common pattern observed across all benchmarks is that $\mathsf{FVC}^{\mathsf{U}}$ always generates the smallest formulas. $\mathsf{FVC}^{\mathsf{U}}$ also scales well on verification times, performing consistently close to the best algorithm for each benchmark, with two exceptions: `ex39` and `fib`. $\mathsf{FVC}^{\mathsf{U}}$ is faster than the other compact VC algorithms on these two benchmarks, but is outperformed by the path-based algorithms. These two programs have only a few feasible paths, and each feasible path can be concretely executed. This allows the path-based algorithms to generate formulas that are easier to solve. In contrast, compact VC algorithms must merge paths at confluence points, causing values to become symbolic, and thus making it harder for the solver to reason about each path. The obvious downside is that the path-based VCs create much larger formulas, and eventually hit the 2GB limit for the experiments.

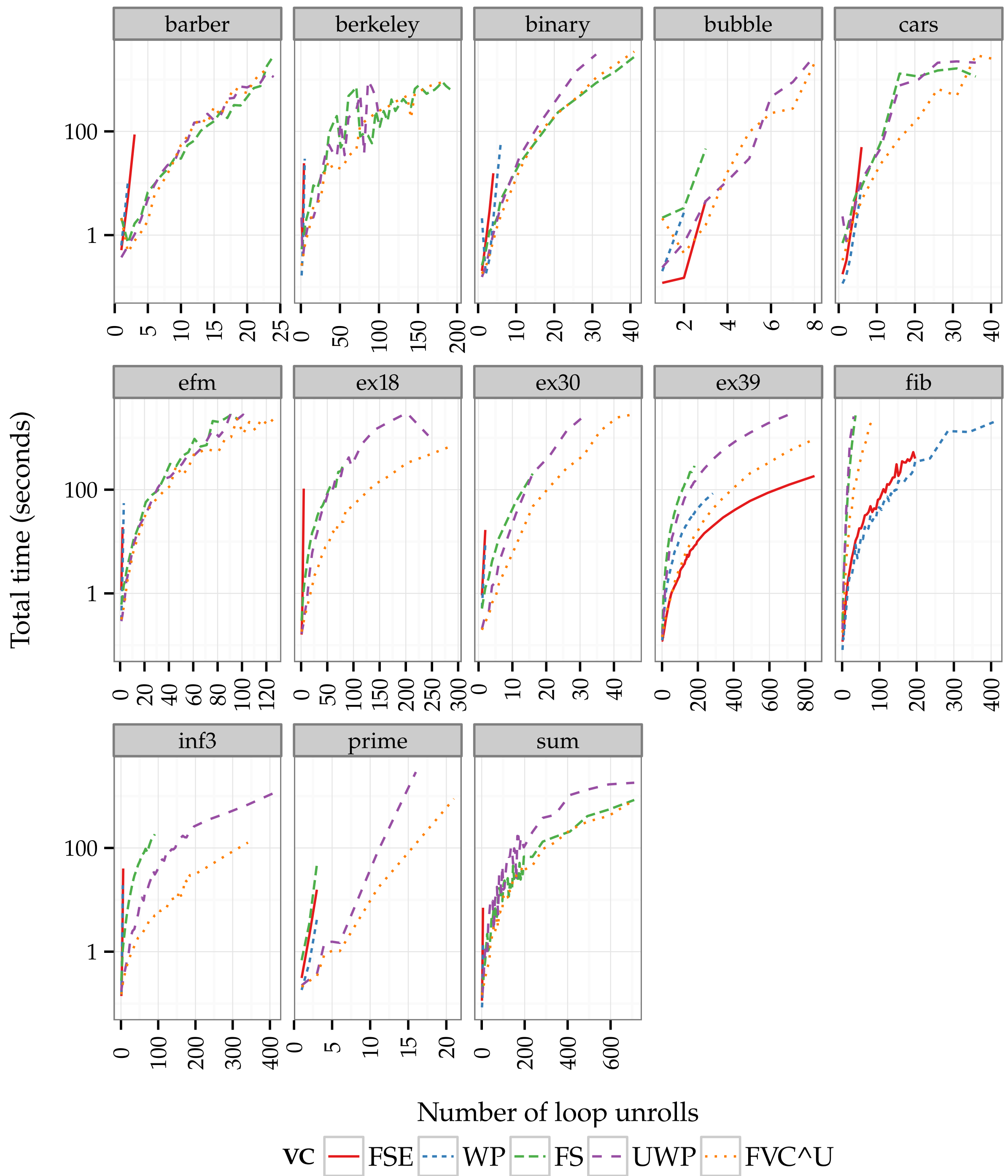


Figure 3.8: Total verification time compared to number of loop unrolls.

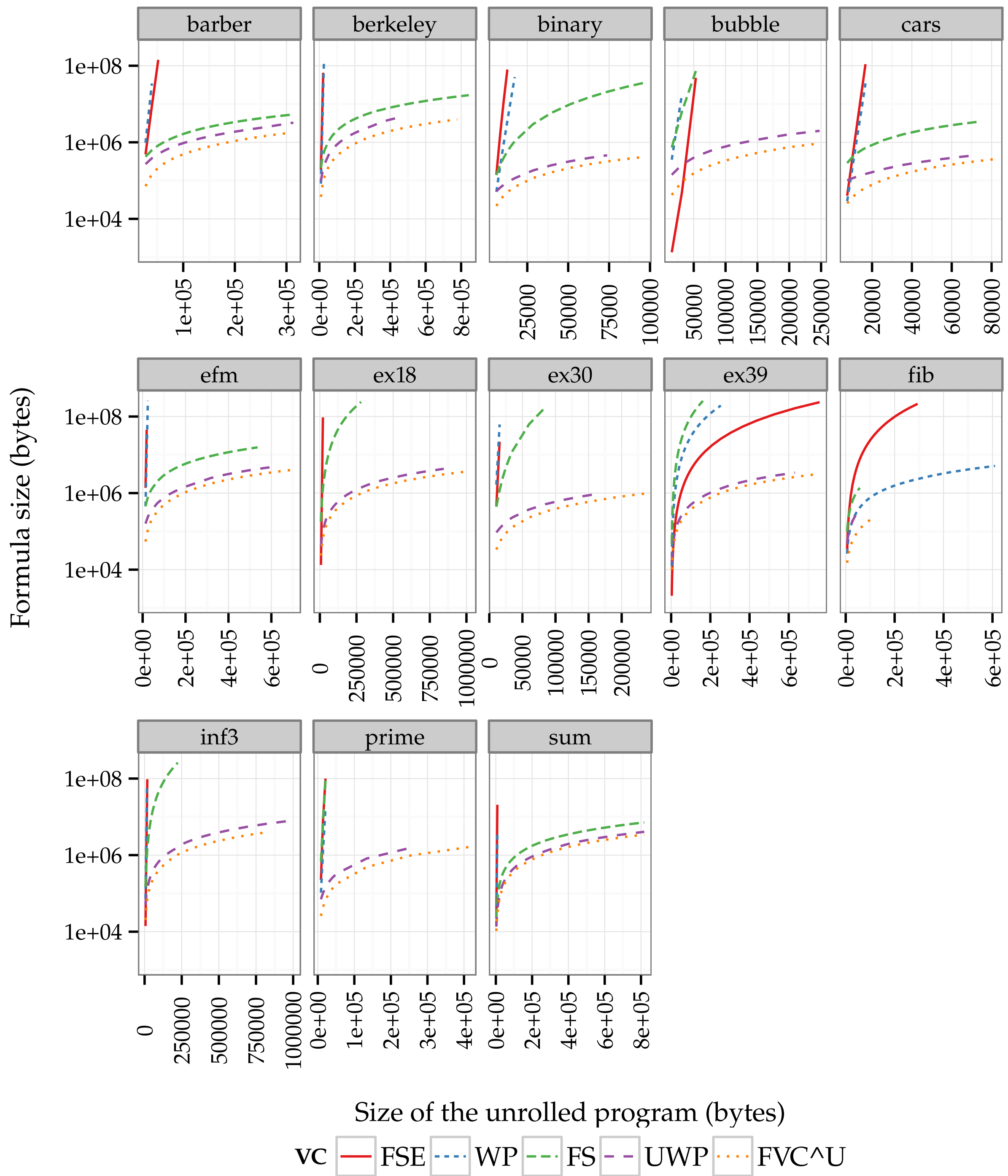


Figure 3.9: Formula size compared to program size.

3.5 Discussion

The passify algorithm [54] preserves the meaning of the program as long as a validity checker is being used.[6] In particular, it has the following property:

$$\forall \vec{v'}.\mathsf{WP}\,(\mathsf{passify}(P))\,Q \equiv \mathsf{WP}\,P\,Q,$$

where $\vec{v'}$ are the variables introduced by passification. Informally, this property asserts that the weakest precondition of the passified program is equal to the weakest precondition of the original program, as long as the new variables are quantified over all possible values, which is the case for validity testing.

Such a VC will have nonsensical satisfying inputs, however. For example, $\mathsf{WP}\,(\mathsf{passify}(v := 5))\,\mathsf{true}$ is satisfied by $v \mapsto 6$, but this is not a feasible execution of the original program. When querying VCs for satisfiability, such meaningless executions should make the VC false. The solution is to modify the assignment rewriting stage of the passify algorithm to convert assignment statements $v := e$ to assertion statements, instead of assumption statements. This produces a passification algorithm such that:

$$\exists \vec{v'}.\mathsf{WP}\,(\mathsf{passify}(P))\,Q \equiv \mathsf{WP}\,P\,Q,$$

which is appropriate for satisfiability.

3.6 Related Work

King initially proposed symbolic execution (FSE) [75] as a technique for reasoning about multiple inputs on a program path. In recent years, there has been a renewed interest in FSE in many different research areas [11, 59, 73, 105, 114], despite not producing compact VCs. FSE applications usually produce VCs for smaller program fragments (e.g., a few program paths) to avoid blowup. Because they cover smaller program fragments, FSE VCs are often solved for a satisfying input rather than being verified. For instance, FSE has been used to generate inputs to drive execution to unexplored parts of the program [59, 114], summarize execution paths for model checking of Java programs [73], and discover new predicates for predicate abstraction-based verification tools [11]. Many other recent applications are described in a recent survey of symbolic execution techniques and applications [105].

Several techniques have been proposed for scaling FSE. One approach is to identify heuristics for when symbolic executors should be merged [5, 19, 80]. For example, a simple strategy for merging is to stop exploring a path if its suffix is identical to a previously executed path [19]. However, Kuznetsov et al. [80] note that merging too often can result in small formulas that are difficult to prove. Thus, these heuristics

[6] Or any checker that supports quantifiers.

may also be of interest in the context of FVC to ensure that formulas do not become too difficult to solve, by splitting the program.

Koelbl and Pixley [76] and Xie et al. [129] were the first to propose using static symbolic execution (SSE) to encode multiple execution paths in a single formula. Babic [9] extended SSE further by leveraging Gated Single Assignment [122] and sharing at the CFG and expression level (maximally-shared graphs) to produce more compact VCs. The proposed SSE-based algorithms perform VC generation in the forward direction, but they do not provide a guaranteed $O(n^2)$ bound for the size of the generated formulas.

Flanagan et al. [54] created the first algorithm for calculating the weakest precondition that is $O(n^2)$ in size. Barnett and Leino have also proposed an efficient weakest precondition algorithm for unstructured code [13]. Existing weakest preconditions algorithms are primarily used in program verification frameworks such as WHY3 [18], Boogie [84], and extended static checkers such as ESC/Java [53]. These frameworks generate VCs for program fragments (e.g., functions), and then perform modular verification of the program by proving the validity of the VCs.

3.7 Conclusion

The ability to produce verification conditions (VCs) is a fundamental primitive in program verification, program analysis, and bug finding. This chapter showed that FVC can generate compact VCs by visiting the program in the forward direction. These compact VCs are quadratically-sized in the worst case compared to the original program. FVC enables optimizations such as concrete evaluation and path pruning that are not possible with backwards algorithms. With these optimizations, FVC can verify programs more quickly than existing VC algorithms.

Part II

General Abstractions

Chapter 4

Gadget Abstractions

Although Part I of this thesis describes techniques that recover source code abstractions from low-level binary code, it is important to realize that these techniques have limitations. This chapter explores the utility of recovering abstractions besides source code abstractions. The primary disadvantages of source code abstractions, and thus the motivation for studying other types of abstractions, are that (1) source code abstractions cannot represent the semantics of all binaries, and that (2) source code abstractions cannot answer every question about the behavior of binary code.

Some binary code cannot be represented in abstract form. One example of a function that cannot be represented abstractly is the `get_pc_thunk` function in the Embedded GNU C Library (EGLIBC), which returns the value of the program counter so that a program or library can detect where it is loaded into memory. There is no way to represent `get_pc_thunk` using C abstractions because the program counter is not a concept that can be expressed with C abstractions. More generally, source code abstractions for a given language are specific to binaries compiled from that language; other binaries may have behavior that cannot be represented using that language's source code abstractions.

Even when a binary can be represented with source code abstractions, it may be difficult (or even undecidable) to create an analysis that recovers those abstractions in practice. Some post-compilation transformations can exacerbate this problem. One example of this is program obfuscation, in which attackers intentionally modify a program to aggravate program analysis and reverse engineering. Unfortunately, obfuscation generally makes abstraction recovery difficult too, since abstraction recovery is usually based on program analysis. Benign transformations can be problematic too. For example, a link-time optimizer that inlines function calls can remove the evidence of function boundaries. Counter-intuitively, optimizing a program to run faster can actually make analysis slower [126].

Abstraction reduces complexity by removing details, but it is also possible to abstract away too much

detail. Source code abstractions describe high-level program behavior, and abstract away low-level details about how the abstract behavior is implemented. As a result, source code abstractions are useful for reasoning about high-level program behavior, but do not capture the concrete aspects of a binary's behavior, such as how it implements undefined behaviors. For instance, they cannot explain how a program will behave during a buffer overflow. Thus, source code abstractions are undesirable for answering questions about low-level program behavior.

General abstractions are those abstractions that can be found from any binary. This chapter introduces a general abstraction called the *gadget* abstraction. Just as a gadget is a small machine or invention that accomplishes a specific task, the gadget abstraction records the presence of a specific behavior or computation in a small part of the binary code, but abstracts away the details of how the computation is implemented. Gadgets are useful because they allow an analysis to determine to what extent an attacker can control a program with a vulnerability in the presence of ASLR and DEP, two widely deployed software defenses.

4.1 Introduction

Control flow hijack vulnerabilities are extremely dangerous. In essence, they allow the attacker to hijack the intended control flow of a program and instead execute whatever actions the attacker chooses. These actions could be to spawn a remote shell to control the program, to install malware, or to exfiltrate sensitive information stored by the program.

Luckily, modern OSes now employ DEP and ASLR, two defenses intended to thwart control flow hijacks. Data execution prevention (DEP, also known as W$\oplus$X) prevents an attacker's payload itself from being directly executed. Address space layout randomization (ASLR) prevents an attacker from utilizing structures within the application itself as a payload by randomizing the addresses of program segments. Microsoft said the following about ASLR and DEP in their 2009 Security Intelligence Report [6, p. 44]:

> The combination of ASLR and DEP creates a fairly formidable barrier for attackers to overcome in order to achieve reliable code execution when exploiting vulnerabilities.

Unfortunately, modern operating systems such as OS X, Linux, and Windows do not completely enforce ASLR and DEP. A complete ASLR implementation ensures that no portion of code is unrandomized, and a complete DEP implementation always prevents injected code from being executed. Some examples of incompleteness are that Linux does not randomize the program image, and third-party Windows applications must opt-in to ASLR and DEP for the defenses to be enabled, but many do not [103]. These implementations are incomplete because of the high costs of completely enabling the defenses, which include breaking existing applications and introducing a performance penalty.

Previous work [115] has shown that systems that do not randomize large libraries like libc are vulnerable to return-oriented programming (ROP) attacks. At a high level, ROP reuses instruction sequences already present in memory that end with `ret` instructions, and these sequences are called *gadgets*. Shacham showed that it was possible to build a Turing-complete set of gadgets using the program code of libc. Finding ROP gadgets has since been, to a large extent, automated when large amounts of code are left unrandomized [50, 68, 107]. However, it has been left as an open question whether current defenses, which randomize large libraries like libc, but leave small amounts of code unrandomized, are sufficient for all practical purposes, or permit such attacks.

This chapter develops automated ROP techniques that require only small amounts of unrandomized code to bypass current defenses, and uses these techniques to demonstrate that current implementations of defenses are often vulnerable. Some of the new insights for analyzing small amounts of unrandomized code include:

- A ROP system can find more gadgets using *semantic* definitions instead of *syntactic* definitions. For instance, a syntactic approach might declare any assembly code that matches `movl *, *; ret` to be a move gadget [68, 107]. However, some gadgets, such as those that use mathematical identities, are difficult to characterize using syntactic patterns. By using the identity property of multiplication, for example, `imul $1, %eax, %ebx; ret` moves a value from `%eax` to `%ebx`. Q can detect such gadgets because it uses the semantic definition OutReg $\leftarrow$ InReg to define move gadgets. Such a semantic approach describes the desired *behavior* rather than how that behavior is implemented.

- The *every munch* algorithm allows a ROP compiler to function even when missing some types of gadgets. This is comparable to writing a compiler for an instruction set architecture that is missing key instructions; for instance, the compiler must still be able to add two numbers even when the `add` instruction is missing. The every munch algorithm searches over many combinations of gadget types to synthesize a working payload even when the most natural gadget type is unavailable. Prior work on ROP compilers [50, 68, 107] focuses on finding gadgets for all gadget types, such that the compiler can then create a program using these gadget types. This direct approach does not work if some gadget types are missing. The use of every munch is critical when compiling ROP from small code bases, since most programs will be missing at least some gadget types.

While it has long been known that ASLR and DEP offer important protection in theory, the main conclusion of this chapter is that current defense implementations make compatibility and performance tradeoffs, and as a result it is possible to *automatically harden existing exploits* to bypass these defenses. Our results

build on existing ROP research, which demonstrated that allowing large amounts of unrandomized code undermines the protection of DEP. The initial ROP research was performed by hand [25,31,115]. Later, researchers developed automated ROP techniques for libc [107] (1,300KB), a kernel [68] (5,910KB) and mobile libraries [50,77] (size varies; on order of 1,000KB). In contrast, the techniques described in this chapter can build ROP payloads using much smaller amounts of unrandomized code (20KB). In fact, these techniques can build ROP payloads for 85% of Linux programs larger than 20 kB. This chapter also presents techniques for transplanting a ROP payload into an existing exploit that does not bypass defenses, effectively hardening the original exploit to bypass DEP and ASLR. Recent work in automatic exploit generation [7, 23] has demonstrated that automated techniques can generate such unhardened exploits automatically. The techniques developed in this chapter were shown to automatically harden exploits to bypass defenses on nine exploits for real binary programs on Linux and Windows. Since these defenses can automatically be bypassed, it suggests that they provide insufficient security.

Contributions The main contribution of this chapter is demonstrating that existing ASLR and DEP implementations do not provide adequate protection by developing automated techniques to bypass them. A survey of modern defense implementations and their weaknesses motivates the problem setting. Second, automatic ROP techniques for small, unrandomized code bases as found in most practical exploit settings are introduced. These techniques can automatically compile programs written in a high-level language down to ROP payloads. Third, these techniques are evaluated in an end-to-end system, and the evaluation shows that existing defenses can be automatically bypassed for nine real-life vulnerabilities on Windows and Linux.

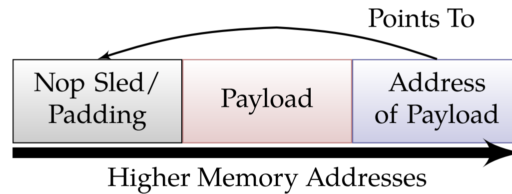


Figure 4.1: Traditional code injection exploit.

4.2 Background

4.2.1 Control Flow Hijack Vulnerabilities

This chapter is about defenses against control hijack exploits. A successful control flow exploit will usually allow an attacker to execute arbitrary code in the context of a vulnerable program. Regardless of the exact attack mechanisms (stack overflow, format string vulnerabilities, etc.) for control hijacks, many traditional control flow hijack exploits have a similar structure to the one shown in Figure 4.1.

This exploit demonstrates two critical properties that all control flow exploits must satisfy.

Computation Every control flow hijack exploit contains a computation that the attacker wishes to perform in place of the original program logic. Traditional exploits accomplished this by including *shellcode*, or machine code, that an attacker injects into an executable buffer. Recent research has shown that it is also possible to use ROP payloads to specify computations.

Control Flow A control flow exploit must disrupt the regular control flow of the program, or the attacker's code will never execute. In a stack buffer overflow, the attacker often overwrites the saved return address for a function. When the function returns to the saved return address, control is diverted to whatever address the attacker wrote instead of the original saved value.

4.2.2 Modern Defense Overview

This section introduces two defenses against control flow hijack exploits, DEP and ASLR. These defenses are intended to thwart attackers who attempt to gain reliable code execution. One of the central themes of this chapter is that DEP and ASLR are much less effective when the defenses are only applied to part of the program in memory. Thus, a main focus in this overview is to explain the DEP and ASLR implementations in common operating systems with an emphasis on their limitations.

Table 4.1 lists some limitations of common DEP and ASLR implementations. The major limitation exploited in this chapter is that, on Linux, program images are unrandomized by default. It may also be surprising to some that some systems silently disable DEP on older hardware.

Operating System	DEP	ASLR		
		stack, heap	libraries	program image
Ubuntu 10.04–13.10	Yes	Yes	Yes	Opt-In
Debian 3.1–7	HW	Yes	Yes	Opt-In
Windows Vista, 7, 8	HW/Transition	Yes	Transition	Transition
Mac OS X 10.6	Yes	No	Yes	No
Mac OS X 10.8	Yes	Yes	Yes	Yes

Table 4.1: Comparison of defenses on modern operating systems for the x86 architecture with default settings. Opt-In means that a non-default compiler flag must be set for the protection to be enabled. Transition means that recent compilers enable the protection by default, but many legacy binaries are compiled with the protection disabled. HW denotes that the level of protection depends on hardware.

DEP Data Execution Prevention (DEP) prevents attackers from injecting their own payload and executing it by ensuring that protected program segments are not writable and executable at the same time.[1] Attackers have traditionally included shellcode (executable machine code) in their exploits as payloads. Since shellcode must be written to memory at runtime, the DEP property ensures it must also be non-executable.

DEP is often implemented [45, 67, 100] using a NX (no execute) bit that the hardware platform enforces: if execution moves to a page with the NX bit enabled, the hardware raises a fault. On x86, this bit can be set using the PAE addressing mode [69]. Some distributions of Linux, such as Debian, silently disable DEP when PAE support is unavailable. Other distributions handle the lack of hardware support differently. For example, versions of Ubuntu prior to 12.10 included ExecShield [124], a patch for the Linux kernel that emulates DEP on processors without PAE by using x86 segments. Newer versions of Ubuntu require PAE to install.

Windows Vista and 7 both enable DEP by default on processors supporting the NX bit.[2] In contrast, Windows 8 requires support for the NX bit to install. However, all three versions of Windows only enforce DEP for binaries and libraries marked as DEP compatible. Many notable third-party software programs such as Oracle's Java JRE, Apple Quicktime, VLC Media Player and others do not opt-in to DEP [103].

A fundamental limitation of DEP is that it only prevents an attacker from utilizing *new* payload code. The attacker can still reuse existing code in memory. For instance, an attacker can call `system` by launching a return-to-libc attack, in which the attacker creates an exploit that will call a function in libc without injecting any shellcode. DEP does not prevent return-to-libc attacks because the executed code is in libc and is intended to be executable at compile time. Return-Oriented Programming (ROP) is more advanced attack on DEP, and is the focus of this chapter. It is discussed further in Section 4.2.3.

[1] DEP is sometimes referred to as W$\oplus$X, but this is a misnomer, since memory is allowed to be unwritable and non-executable, but $0 \oplus 0 = 0$.

[2] Windows also contains *software DEP*, but this is unrelated to DEP [45].

ASLR Address Space Layout Randomization (ASLR) prevents an attacker from directly referring to objects in memory by randomizing their locations. This stops an attacker from reliably transferring control to her shellcode by hard-coding its address in her exploit. Likewise, it makes return-to-libc and ROP using libc difficult, because the attacker will not know where libc is located in memory.

ASLR implementations often randomize the stack, heap, shared libraries (e.g., libc), and program image, or some subset. Linux [99,124] randomizes the stack, heap, and shared libraries, but not the program image by default. To randomize the program image, the user must compile the program into a position independent executable (PIE), which is typically not a default. Modern distributions [43,123] only compile a select group of programs as PIEs. This is probably because PIE code can introduce a 10% performance overhead at runtime for computationally expensive tasks [101].

Windows Vista and 7 [67,120] can randomize the locations of the program image, stack, heap, and libraries, but only when the program and all of its libraries opt-in to ASLR. If they do not, some code is left unrandomized. Many third-party applications including Oracle's Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked as ASLR compatible [103]. Ultimately, this means most Windows binaries have unrandomized code.

Some attacks on ASLR implementations take advantage of the low entropy available for randomization. For instance, Shacham, et al. [116] show that brute forcing ASLR on a 32-bit platform takes about 200 seconds on average. Other attacks, such as `ret2reg` attacks, allow an attacker to transfer control to her payload by utilizing pointers leaked in registers or memory [92]. For instance, the `strcpy` function returns such a pointer to the destination string in the `%eax` register. The applicability of these attacks are heavily dependent on the vulnerable program.

4.2.3 Return-Oriented Programming

Return-Oriented Programming (ROP) is an attack in which the attacker uses instruction sequences found in memory, called gadgets, and chains them together to encode a larger computation. ROP is a generalization of the return-to-libc attack, which the attacker reuses entire functions from libc. ROP attacks are desirable because they allow the attacker to perform computations beyond the functions of libc (or whatever code is unrandomized). This is especially important in the context of modern systems, because the unrandomized code may not contain functions that are directly useful for the attacker. Researchers [50,68,115] have shown that it is possible to find a Turing-complete set of gadgets in libc, the windows kernel, and mobile phone libraries.

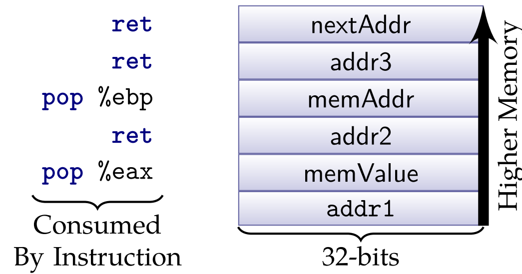


Figure 4.2: Example return-oriented payload that stores memValue to memAddr and then transfers controls to address nextAddr for the scenario described in the text.

Example

Assume that the following instruction sequences are in memory at addr1: `pop %eax; ret`; at addr2: `pop %ebp; ret`; and at addr3: `movl %eax, (%ebp); ret`. The first two sequences pop a 32-bit value from the stack, store it into a register, and then jump to the address stored on the stack. If the attacker controls the stack and can cause one of these instruction sequences to execute, then the attacker can put values in `%eax` and `%ebp` and transfer control to another address. By chaining together all three instruction sequences, the attacker can write to memory (and still transfer control to the next gadget). The attacker's payload for writing memValue to memAddr is shown in Figure 4.2.

4.3 System Overview

Q, a system for automatic exploit hardening, is described in the next two sections.[3] Figure 4.3 shows the end-to-end workflow of Q, which is divided into two phases. The first phase automatically generates ROP payloads (Section 4.4). The second phase is exploit hardening (Section 4.5), during which Q transplants ROP payloads generated in the first stage into existing exploits which do not bypass defenses. The resulting hardened exploit can then bypass DEP and ASLR.

[3] Q was named after the character of the same name in the James Bond movies, who creates, modifies, and combines gadgets to help Bond meet his objectives.

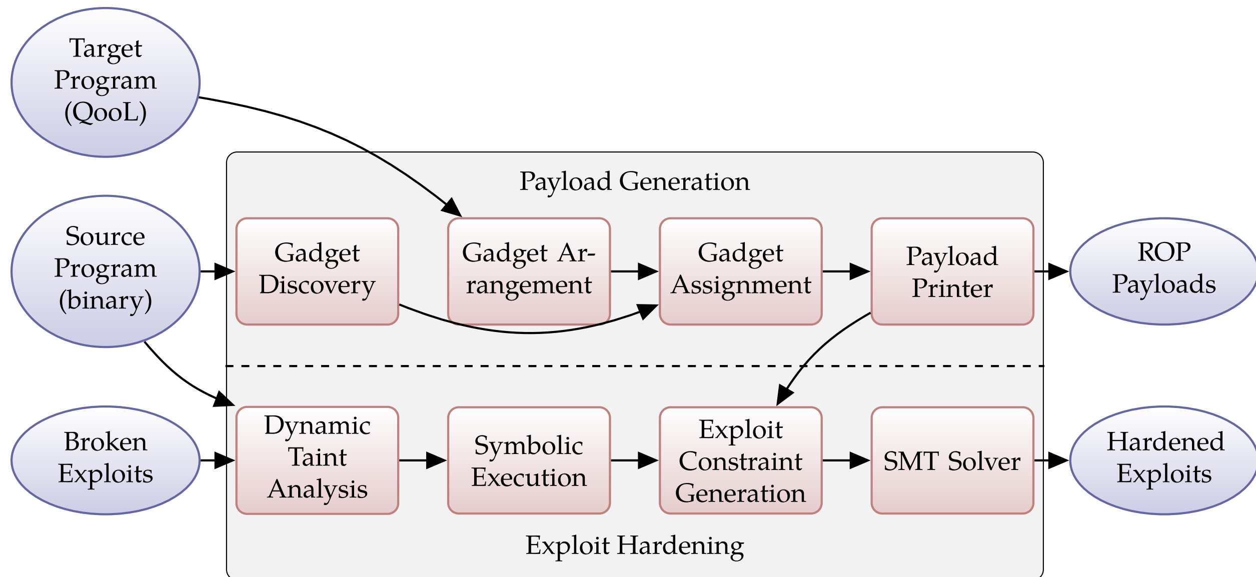


Figure 4.3: Overview of Q's design.

4.4 Automatically Generating Return-Oriented Payloads

Q's end-to-end return-oriented programming system consists of a number of different stages. Previous research on automated ROP has typically focused on one specific stage; for instance, gadget discovery [50, 77, 107] or compilation [25]. Since Q is an end-to-end ROP system, it has multiple stages. Each stage is described in the context of a user's potential interaction with the system below.

Assume that Alice wants to create a ROP payload that calls `system` (her target program) using instructions from `rsync`'s unrandomized code (her source program). Here, source program means the program from which Q takes instruction sequences to construct gadgets (e.g., the program with a vulnerability), and target program means the program Alice wants to run using ROP. Alice would use the following stages of Q, which are depicted in the top half of Figure 4.3:

Gadget Discovery The first stage of Q is to find gadgets in the source program that Alice provides — in this case, `rsync`. The gadgets will be the building blocks for the ROP payloads that are ultimately created, and thus it is important to find as many as possible. Q finds gadgets of various types (specified in Table 4.2) by using semantic program verification techniques on the instruction sequences found in `rsync`.

Q's semantic engine allows it to find gadgets that humans might miss. For instance, Q can automatically determine that `lea (%ebx,%ecx,1), %eax; ret` adds `%ebx` with `%ecx` and stores the result in `%eax`. Likewise it discovers that `sbb %eax, %eax; neg %eax; ret` moves the carry flag to `%eax`.

Input Alice writes the target program that she wants to execute in Q's high level language, QooL (shown in Table 4.3). The target program calls `system` with the desired arguments (e.g., `/bin/sh`).

Gadget Arrangement Q builds a list of *gadget arrangements.* Each gadget arrangement is an implementation of the target program using different types of gadgets. For example, a gadget arrangement for writing to memory is shown in Figure 4.8; this arrangement is the most natural way of storing to memory, but will not work if Q cannot find a StoreMemG gadget. Gadget arrangement is somewhat analogous to instruction selection in a compiler. A major difference is that a regular compiler can use whichever instructions it chooses, but Q is limited to the gadget types that were found during gadget discovery.

Gadget arrangement allows Q to cope with missing gadgets. If the most natural choice of gadget is not available, Q tries to synthesize a combination of other gadgets that will have the same semantics.

Gadget Assignment Gadget assignment takes actual gadgets found during discovery, and assigns them to the arrangements that Q generated during gadget arrangement. The primary challenge is that assignments must be compatible. This means that the output register of one gadget must match the input register on the receiving gadget. Likewise, gadgets cannot clobber a register if that value is waiting to be used by a future gadget. This phase is roughly analogous to register allocation in a traditional compiler. Unlike a traditional compiler, Q cannot spill registers to memory, since this usually increases register pressure instead of decreasing it. As an example, Figure 4.4 shows the gadgets from `rsync` that Q assigned to implement the gadget arrangement in Figure 4.8.

```
# Load value into %eax
pop %ebp; ret
xchg %eax, %ebp; ret
# Load address-0x14 into %ebx
pop %ebx; pop %ebp; ret
# Store memory
mov %eax, 0x14(%ebx); ret
```

Figure 4.4: Gadgets assigned from `rsync` to implement the gadget arrangement in Figure 4.8.

Output If at least one gadget arrangement has been assigned compatible gadgets, Q prints out payload bytes that Alice can use in her exploit. If Alice already has an exploit that DEP and ASLR stopped from working, she can feed in the generated ROP payload along with her old exploit to the second phase of Q (see Section 4.5) to harden her exploit against these defenses.

4.4.1 Gadget Discovery

Not every instruction sequence can be used as a gadget. Q requires each gadget to satisfy four properties:

Name	Input	Parameters	Semantic Definition
NoOpG	—	—	No side-effects
MoveRegG	InReg, OutReg	—	OutReg $\leftarrow$ InReg
LoadConstG	OutReg, Value	—	OutReg $\leftarrow$ Value
ArithmeticG	InReg1, InReg2, OutReg	$\Diamond_b$	OutReg $\leftarrow$ InReg1 $\Diamond_b$ InReg2
LoadMemG	AddrReg, OutReg	#Bytes, Offset	OutReg $\leftarrow$ M[AddrReg + Offset]
StoreMemG	AddrReg, InReg	#Bytes, Offset	M[AddrReg + Offset] $\leftarrow$ InReg
ArithLoadG	OutReg, AddrReg	#Bytes, Offset, $\Diamond_b$	OutReg $\Diamond_b\leftarrow$ M[AddrReg + Offset]
ArithStoreG	InReg, AddrReg	#Bytes, Offset, $\Diamond_b$	M[AddrReg + Offset] $\Diamond_b\leftarrow$ InReg

Table 4.2: Gadget types. M[addr] means accessing memory at address addr. $\Diamond_b$ means an arbitrary binary operation. a $\leftarrow$ b denotes that final value of a equals the initial value of b. X $\Diamond_b\leftarrow$ Y is short for X $\leftarrow$ X $\Diamond_b$ Y.

Functional Each gadget has a *type* (from Table 4.2) that defines its function. A gadget's type is specified *semantically* by a boolean predicate that must always be true after executing the gadget.

Control Preserving Each gadget must be capable of transferring control to another gadget. In Q, this means that the gadget must end with `ret` or some semantically equivalent instruction sequence (e.g., `pop %eax; jmp *%eax`).

Known Side-effects The gadget must not have unknown side-effects. For instance, the gadget must not write to any undesired memory locations.

Constant Stack Offset Most gadget types require the stack pointer to increase by a constant offset after each execution.

Some alternatives are discussed in Section 4.8.

Gadget Types

The set of gadget types in Q defines a new instruction set architecture (ISA) in which each gadget type acts as an instruction. At a high-level, each gadget type is specified by a postcondition Q. An instruction sequence S satisfies a postcondition Q if and only if the postcondition is true after running S from any starting state. Q can recognize any of the gadget types listed in Table 4.2.

Semantic Analysis

Given an instruction sequence S and a semantic definition Q, Q must decide if the properties in Q will always occur after executing S. One way to determine this is to use a *weakest precondition* algorithm such as the one presented in Chapter 3. At a high level, the weakest precondition WP $S\,Q$ of a program S and postcondition Q is a boolean formula that describes when S will terminate in a state satisfying Q.

Q verifies whether the semantic definition Q always holds after executing the instruction sequence S by checking if the formula WP S Q is valid, or true for any input. If it is, then the semantic definition Q is guaranteed to hold after every execution of S, and thus S is a gadget with the semantic type Q.

Although in theory using weakest preconditions by itself is sufficient to implement gadget discovery, doing so is too slow in practice. Thus, Q also performs a number of random concrete executions to improve performance. Each concrete execution evaluates each possible Q concretely, and when any concrete execution finds a Q to be false, the corresponding gadget type is ruled out, since Q must hold after every execution. Thus, the weakest precondition process is only invoked when Q is true for all random concrete executions.

Q also infers potential parameter values (shown in Table 4.2) by performing dynamic analysis during random concrete execution. For instance, by looking at the values of all registers, and the addresses that memory was read from, Q can compute a set of possible offsets for the LoadMemG gadget type.

As an example of how a gadget type is tested, consider the LoadMemG gadget type in Table 4.2. LoadMemG gadgets operate on two registers: the output register and the address register. Each LoadMemG gadget has two parameters, #Bytes and Offset, which are specific to a particular instruction sequence S. These will be found using dynamic analysis as described above. For instance, the instruction sequence `movl 0xc(%eax), %ebx; ret` is a LoadMemG gadget with parameters $\{\#\text{Bytes} \mapsto 4, \text{Offset} \mapsto 12\}$ and registers $\{\text{OutReg} \mapsto \texttt{\%ebx}, \text{AddrReg} \mapsto \texttt{\%eax}\}$. The semantic definition for this instruction sequence would be `%ebx` ← M[`%eax` + 12]. Q converts this to $\text{final}(\texttt{\%ebx}) = \text{initial}(\text{M}[\texttt{\%eax} + 12])$, the postcondition Q that is checked for validity.

Gadget Discovery Algorithm

Gadget discovery consists of two algorithms. The `discover` algorithm, shown in Algorithm 1, tests whether or not the semantics of an instruction sequence S match those of any gadget type using randomized concrete testing and validity checking of the weakest precondition. In the actual implementation, `discover` also outputs some metadata (not shown) about each discovered gadget for use in other Q algorithms; this metadata includes the gadget's address, stack offset, and any registers that the gadget clobbers. The second algorithm iterates over the executable bytes of the source program, disassembles them, and calls the first algorithm as a subroutine. This is similar to the Galileo [115] algorithm, and so it is not replicated here.

```
1  Function discover(S, numRuns, gadgetTypes[])
       // Execute S from numRuns random input states
2      for i ← 1 to numRuns do
3          inState[i] ← getRandomInputState();
4          outState[i] ← S(inState[i]);            // Concretely execute S from state inState[i]
5      end
       // Test if the result of each randomized execution is consistent with the behavior
          of gadget type gtype
6      foreach gtype ∈ gadgetTypes do
7          consistent ← true;
8          Q ← gtype.postcondition;                    // Select the postcondition Q for gtype
9          for i ← 1 to numRuns do                           // Test if Q holds in execution i
10             if Q(outState[i]) = false then consistent ← false ;
11         end
12         if consistent then                              // S might be a gadget of type gtype
13             precondition ← WP(S, Q);
14             if precondition is valid then           // Check precondition with validity checker
15                 output S as gadget of type gtype
16             end
17         end
18     end
19 end
```

Algorithm 1: Automatically test an instruction sequence S for gadgets

4.4.2 Gadget Arrangement

Q is similar to a compiler: it reads in programs written in QooL (discussed below) searches for an implementation using the gadgets types shown in Table 4.2. These gadgets define an instruction set architecture, and thus Q can benefit from techniques from compiler theory. However, Q must deal with several hard problems not faced by most compilers:

- Only a few registers can be used for moving, accessing memory, and performing arithmetic operations.
- Most instructions will clobber (modify) the majority of available registers.
- Some instruction types may not be available at all.

Q's Language: QooL

Users write the target program in Q's high level language, QooL, which is displayed in Table 4.3. QooL enables the user to easily interact with the exploited program's environment. For instance, the attacker can call a function (e.g., `system`), overwrite values in memory to violate program invariants, or copy and run a binary payload (when DEP is not present or has been disabled by first calling `mprotect` or a similar function). QooL is not Turing-complete; this is discussed further in Section 4.8.

program	::=	stmt*
stmt	::=	var := exp | store(exp, exp, τ_{reg}) | call-extern(exp, exp*) | syscall
exp	::=	load(exp, τ_{reg}) | exp $\lozenge_b$ exp | $\lozenge_u$ exp | var | integer
var	::=	(string, id$_v$, τ)
$\lozenge_b$	::=	$+, -, *, /, /_s, \text{mod}, \text{mod}_s, \ll, \gg, \gg_a, \&, \|, \oplus, ==, !=, <, \leq, <_s, \leq_s$
$\lozenge_u$	::=	$-$ (unary minus), $\sim$ (bit-wise not)
integer	::=	$n{:}\tau_{\text{reg}}$
τ_{reg}	::=	reg_t(n)

Table 4.3: Q's high level language, QooL.

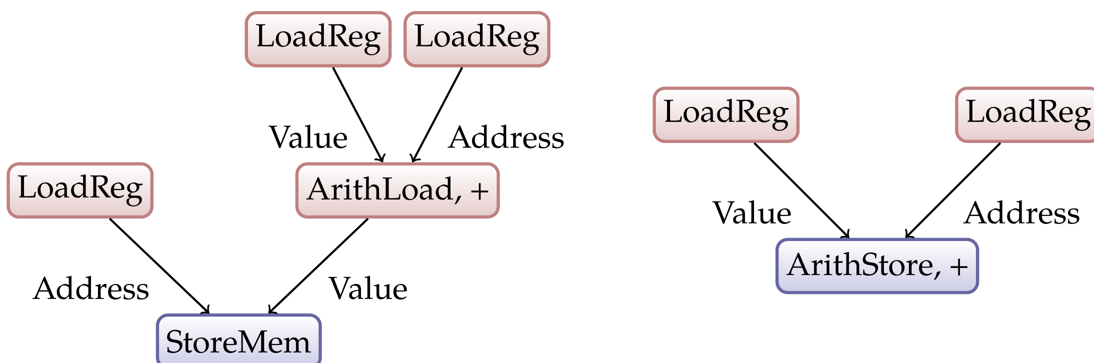


Figure 4.5: Two arrangements that increment a value in memory. It is important to consider multiple arrangements, since only a few of them are typically satisfiable.

Arrangements

One of the essential tasks of a compiler is to perform *instruction selection*, since there are many combinations of instructions that can implement a given computation. The gadget architecture is no exception; there are many ways of combining gadget types to produce a particular computation. Each combination of gadgets is specified by a *gadget arrangement*. A gadget arrangement is a tree in which the vertices represent gadget types,[4] and an edge labeled $type$ from a to b means that the output of gadget a is used for the $type$ input in gadget b.

Two gadget arrangements that increment a value in memory are shown in Figure 4.5. The arrangement on the right increments M[Address] in one step. The other arrangement computes M[Address] + 1 in a register, and then copies the value of that register to memory. Although the difference seems trivial, one arrangement may be satisfiable in a given source program, while the other is not. Our experiments (Section 4.7) show that StoreMemG gadgets are less common than ArithStoreG gadgets, for example.

One simple algorithm for performing instruction selection (or selecting a gadget arrangement, in this case) is the maximal munch algorithm [4]. Maximal munch assumes that any instruction selected as the

[4]Vertices also include parameters that are relevant to the computation, such as binary operator type and number of bytes for memory operations.

best will always be available for use. This assumption makes sense in a traditional compiler, since on a normal architecture there are few restrictions on when instructions can be used.

A gadget arrangement algorithm cannot make such assumptions, however. Any particular gadget type chosen by maximal munch might not be available at that point in the program because Q did not find any. Even if such a gadget was found, it might be incompatible with earlier gadgets, e.g., because they operate on different registers.

Instead of using maximal munch, Q employs *every munch.* Rather than selecting only one arrangement of gadget types as a traditional instruction selection algorithm would, every munch lazily builds a tree representing many possible combinations of gadget types that perform the desired computation. These combinations are generated by recursively evaluating *munch rules* to the program being compiled.

Munch Rules

Each QooL language construct has at least one munch rule that can implement the construct in terms of its subexpressions. For instance, the obvious munch rule for the `StoreMem` statement is to use a StoreMemG gadget, which is shown in Figure 4.6.

```
let munch = function
| StoreMem(e1, e2, t) ->
  let e1l = munch e1 in
  let e2l = munch e2 in
  (* For each e1g,e2g in Cartesian product of e1l and e2l do: *)
  add_output (StoreMemG(addr=e1g, value=e2g, typ=t));
```

Figure 4.6: ML pseudo-code implementing a munch rule.

The initial implementation of Q only contained straightforward rules, and often failed on smaller binaries because they do not contain StoreMemG gadgets (Section 4.7). However, there are alternate methods for writing values to memory, such as writing 0 or -1 before executing an ArithStoreG gadget. For example, Q can write x to memory by bitwise anding the target location with zero, and adding x.

Q discovered the relatively complicated return-oriented program shown in Figure 4.7, which writes a single byte to memory using bitwise or and addition gadgets from /usr/bin/apt − get. More straightforward options were not available.

```
# Load %eax: -1
pop %ebp; ret
xchg %eax, %ebp; ret
# Load %ebx: address-0x5e5b3cc4
pop %ebx; pop %ebp; ret
# Write -1 to M[address]
or %al, 0x5e5b3cc4(%ebx); pop %edi; pop %ebp; ret
# Load %eax: value + 1
pop %ebp; ret
xchg %eax, %ebp; ret
# Load %ebp: address-0xf3774ff
pop %ebp; ret
# Add value + 1 to M[address]
add %al,0xf3774ff(%ebp); movl $0x85, %dh; ret
```

Figure 4.7: Return-oriented payload for `apt-get`.

4.4.3 Gadget Assignment

During gadget assignment, Q determines if a gadget arrangement can be satisfied using the gadgets it discovered in the source program. The goal of gadget assignment is to output an assignment of concrete gadgets to the vertices of arrangements that is *compatible*. It is straightforward to print a ROP payload with this assignment.

A gadget assignment must also select a schedule that assigns each gadget to a particular time slot. It is important to consider multiple schedules because the order in which gadgets execute can influence the data dependencies between different gadgets. For example, if the gadget at T_2 clobbers (overwrites) the Value register in Figure 4.8, the gadget at T_3 will not receive the correct input. To resolve such dependencies between gadgets, a gadget assignment and corresponding schedule must satisfy the following properties:

Matching Registers Whenever the result of gadget a is used as input *type* to gadget b, then the two registers should match, i.e., $\mathsf{OutReg}(a) = \mathsf{InReg}(b, type)$.

No Register Clobbering If the output of gadget a is used by gadget b, then a's output register should not be clobbered by any gadget scheduled to execute between a and b.

A gadget assignment and schedule are *compatible* when the above properties hold, and a gadget arrangement that has a compatible assignment and schedule is *satisfiable*.

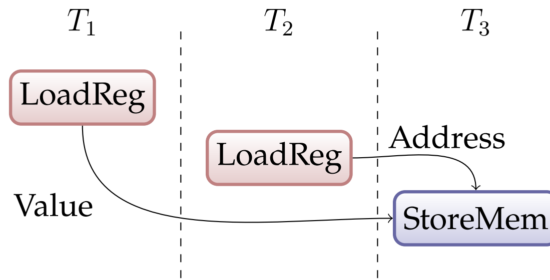


Figure 4.8: Gadget arrangement and schedule for storing a constant value to a constant address.

Although deciding whether a given gadget schedule and assignment are compatible is straightforward (i.e., just ensure the above properties are satisfied), creating a practical algorithm to search for satisfiable arrangements is more complicated. The most straightforward approach is to iterate over all possible arrangements, schedules, and assignments, but this is too inefficient in practice.

Instead, a key observation is that if a gadget arrangement GA is unsatisfiable, then any GA' that contains GA as a subtree is unsatisfiable as well. Thus, one strategy is to satisfy iteratively larger subtrees until one is unsatisfiable, or the entire arrangement has has been satisfied. If a subtree cannot be satisfied, the entire arrangement can be aborted. Since most arrangements are unsatisfiable, this saves considerable time. (If most arrangements are satisfiable, the search will not take very long anyway.)

The assignment algorithms `findAssign` and `findAssignCache` are based on this idea, and are shown in Algorithms 2 and 3 respectively. `findAssign` is a naive search over a schedule for all possible gadget assignments. `findAssignCache` is a caching wrapper that caches results and calls `findAssign` on iteratively larger subtrees. It stops as soon as it finds a subtree which cannot be satisfied. Q calls `findAssignCache` on each possible gadget arrangement until one is satisfiable or none remain.

The algorithms make use of several data structures:

- C: $\mathbb{V} \rightarrow \{0, 1, ?\}$ is a cache that maps a gadget arrangement vertex to one of true, false, or unknown.
- O: $\mathbb{V} \rightarrow \mathbb{N}$ represents the current schedule (i.e., an ordering of vertices) as a one-to-one mapping between each vertex and its position in the schedule.
- G: $\mathbb{V} \rightarrow \mathbb{G}$ is the current assignment of each vertex to its assigned gadget.

Q can also search for assignments that meet other constraints, such as searching for a small payload. This option is useful in practice because ROP payloads are typically larger than conventional payloads, and vulnerabilities usually limit the number of payload bytes that can be written.

```
1  Function findAssign(GA, O, G, nodeNum)
2  |  V ← O^{-1}(nodeNum);                    // Find vertex in GA representing nodeNum
3  |  if V = ⊥ then return true;                // Base case: All time slots assigned
4  |  gadgets ← getGadgetsOfType(getGadgetType(V));
5  |  foreach g ∈ gadgets do
   |  |  // Check if g is compatible with all gadgets before time slot nodeNum
6  |  |  if isCompatible(G, nodeNum, g) then
   |  |  |  // Attempt to schedule later schedule slots
7  |  |  |  findAssign(GA, O, G[V ← g], nodeNum + 1);
8  |  |  end
9  |  end
10 |  return false                              // Unable to schedule time slot nodeNum
11 end
```

Algorithm 2: Find a satisfying schedule and gadget assignment for GA

```
1  Function findAssignCache(GA, C)
2  |  foreach subtree GA' of GA, from shortest to tallest do
3  |  |  if C(GA') = ? then
   |  |  |  // Search for a schedule O that satisfies GA'
4  |  |  |  C(GA') ← ∃O ∈ Schedules(GA'). findAssign(O, Empty, 0) = true;
5  |  |  |  if C(GA') = false then
   |  |  |  |  // Stop early when a subtree is unsatisfiable
6  |  |  |  |  return false
7  |  |  |  end
8  |  |  end
9  |  end
   |  // Return the final value from the cache
10 |  return C(GA)
11 end
```

Algorithm 3: Iteratively try to satisfy larger subtrees of GA, caching results over all arrangements.

4.5 Creating Exploits that Bypass ASLR and DEP

The previous section described how Q generates return-oriented *payloads*. If an attacker can change the control flow of the program so that the payload executes, then the attacker's computation will occur instead of the intended program logic. This section describes how Q can automatically generate an *exploit* that executes a ROP payload to bypass ASLR and DEP, when given as input an exploit that is broken by ASLR and DEP.

This is called the exploit hardening problem. Specifically, the exploit hardening problem is to take a program P and an existing exploit that triggers a vulnerability as input. The existing exploit can be an exploit that does not bypass defenses, or can even be a proof of concept crashing input. The goal is to output an exploit for P that bypasses DEP and ASLR.

Q uses information in the input exploit to reason about other inputs that would trigger the same vulnerability. Specifically, Q reasons about all inputs that follow the same execution path as the original exploit

(i.e., the sequence of conditional branches and jumps taken by an execution of the input). It then attempts to find a new input that takes the same execution path, but uses a return-oriented payload (Section 4.4).

4.5.1 Background: Generating Formulas from a Concrete Run

There can be many inputs along the vulnerable path. Rather than trying to reason about each input individually, it is possible to reason about all of them by constructing a constraint formula using the process of symbolic execution. Such formulas have been used in many research areas, including automatic test case generation, automatic signature creation, and others [23,26,111].

Generating formulas from an input involves two steps:

- The first step is to generate an *execution trace.* Q's recording tool incorporates dynamic taint analysis [37,97,111] to keep track of which instructions deal with tainted (or input-derived) data. Only the tainted instructions are stored in the execution trace. Taint information is also used to decide when to stop recording, which happens when the instruction pointer becomes tainted.

- The second step is to symbolically execute the trace [26,111]. Symbolic execution is similar to normal execution, except that each input byte is replaced with a symbol (e.g., s_i for input byte i). Any computation involving a symbolic input is replaced with a symbolic expression. Computations that do not involve a symbolic input are computed as normal using the processor. Any constraints on the inputs to ensure that execution would be guided down the same path as the execution trace are stored in the constraint formula Π.

4.5.2 Exploit Constraint Generation

The constraint formula Π describes all inputs that follow the vulnerable path. However, exploit hardening is concerned with exploits that hijack control to the desired computation. The α (control flow) and Σ (computation) constraints exclude any inputs that do not have these properties. α is true when a program's control flow has been diverted, and Σ holds when the payload for some desired computation is in the exploit.

Control Flow Constraints

α takes the form $\mathsf{jumpExp} = \mathsf{targetExp}$. $\mathsf{jumpExp}$ is the symbolic expression representing the target of the jump that tainted the instruction pointer, and can be obtained from the execution trace. Since the trace halts when the program jumps to a user-derived address, $\mathsf{jumpExp}$ is the symbolic expression representing the target of the last jump in the trace.

For a typical stack exploit, targetExp is set to the address of the shellcode. In a ROP exploit, targetExp instead points to a `ret` instruction when the ROP payload is located in memory at `%esp`. When `%esp` does not point at the payload, Q can use a stack pivot in place of a `ret`. For instance, `xchg %eax, %esp; ret` would transfer control to the ROP payload pointed to by `%eax`.

Computation Constraints

Computation constraints ensure that the computation payload is available in memory at the proper address at the time of exploitation. For instance, computation constraints for a `strcpy` buffer overflow would be unsatisfiable for a payload containing a null byte, since this would result in only part of the payload being copied.

Computation constraints have the form

$$\Sigma = (\mathsf{mem}[\mathsf{payloadBase}] = \mathsf{payload}[0] \wedge \ldots \wedge \mathsf{mem}[\mathsf{payloadBase} + n] = \mathsf{payload}[n]), \tag{4.1}$$

where payloadBase denotes the starting address of the payload in memory, and payload denotes the bytes in the payload (e.g., the ROP payload from Section 4.4). For basic ROP payloads, payloadBase will be set to `%esp`, since that is where a `ret` will start executing. When using a pivot, payloadBase will point to the destination of the pivot.

Finding an Exploit

Combining these constraints with Π, which only holds for inputs following the vulnerable path, results in a constraint formula that only describes exploits along the vulnerable path:

$$\Pi \wedge \alpha \wedge \Sigma. \tag{4.2}$$

Any assignment to the initial program state that satisfies this constraint formula is an exploit for the program semantics recorded in the trace.

4.6 Implementation

The ROP component (Section 4.4) of Q is built on top of the Binary Analysis Platform (BAP) [22]. The implementation of the gadget discovery, arrangement, and assignment phases comprises 4,585 lines of ML code. The current implementation has been extensively tested with x86 input programs, but there is preliminary work to support x86-64 programs. `stp` [58] is used to determine the validity of generated weakest preconditions.

Q's exploit hardening component (Section 4.5) has been incorporated into the BAP [22] project. There are two sub-components. The tracing component is built in the Pin [88] dynamic binary instrumentation framework. Users can mark input from files, network sockets, environment variables, or program arguments as being tainted. The tracing component is written in C++, and includes 2,102 lines of code written for this thesis.

The analysis component of exploit hardening lifts the recorded assembly instructions into BAP's intermediate language, symbolically execute the trace, obtaining the constraint formula Π, and computes the constraints α and Σ. A solver such as `stp` [58] is then used to find a satisfying answer to the resulting constraint formula, and the result is used to build the exploit. The analysis understands ROP pivots and the Windows structured exception handler and can use them to produce its exploits. The analysis implementation is written in OCaml, and includes 1,090 new lines of code for this thesis.

All components of Q are fully capable of reasoning about Windows and Linux binaries.

4.7 Evaluation

4.7.1 How much unrandomized code is sufficient to enable return-oriented programming?

The most important unanswered question about ROP is how much unrandomized code is needed before ROP becomes possible. To answer this question, Q analyzed all 1,429 ELF programs in the `/usr/bin` directory of the author's Ubuntu 9.10 desktop machine. Results for 75 programs marked as position independent (PIE) were discarded unless otherwise noted. For each program P, Q attempted to generate three types of payloads:

Call-Local External library functions called by P have an entry in P's Procedure Linkage Table (PLT). This payload calls the PLT entries directly.

Call-External Calling external functions that do not have a PLT entry is more complicated. For this, Q uses a technique for calculating the address of functions in libc even when libc is randomized [108]. This

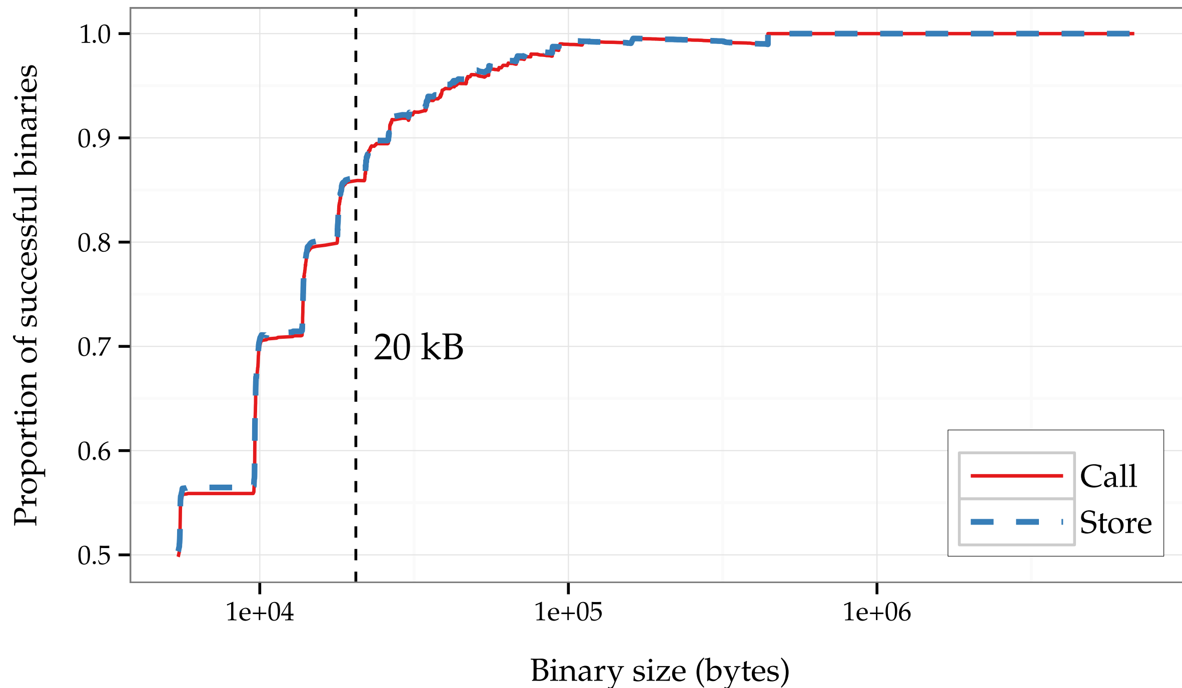


Figure 4.9: Probability that Q can generate payloads as a function of source file size. For example, the point (20 kB, 0.85) signifies that Q created a payload in 85% of the programs that were at least 20 kB in size. The results for the Call-Local and Call-External payloads are identical, and are thus both shown as the Call line.

type of payload involves more computation than a Call-Local payload.

Store This payload writes four bytes to an arbitrary address.

Figure 4.9 shows the probability of successfully generating each type of payload as a function of the smallest program size considered. Several important observations are communicated by the results.

- Q is more likely to generate ROP payloads for larger binaries. This is expected because larger binaries often have more executable code which can be mined for gadgets.
- Only a small amount of unrandomized code is needed before it is possible to create ROP payloads with high probability. In particular, Q was able to generate all three types of payloads for *85% of programs 20 kB or larger.* To put this into context, 20 kB is approximately the size of the `true` command, which always returns true. Ultimately, this means that even a small amount of unrandomized code is dangerous. This is particularly troubling on Linux, because program images are unrandomized by default.
- There is little difference between the Call and Store payloads. There were only 7 programs for which Q generated a Store but was unable to generate a Call payload. This implies that the ability to generate

a payload is not highly correlated with the type of payload, which is somewhat unexpected.

4.7.2 How long does it take to generate ROP payloads?

The time required to run Q can be broken up into (1) the time needed to discover gadgets, and (2) the time required to arrange and assign gadgets for a particular payload type.

Discovery The mean and median times for gadget discovery among the 1,429 tested programs are 158 s and 5.77 s respectively, indicating that most of the tested programs have a relatively short discovery time, but also that there are some long-running outliers. Figure 4.10 displays the distribution of gadget discovery times, and confirms that there is a large variation in the time needed to discover gadgets, ranging from 0.12 s to 2.3 hr. Figure 4.11 suggests a strong correlation ($r = 0.894$) between discovery time and the size of the binary. Intuitively, this is because larger binaries have more instruction sequences to consider.

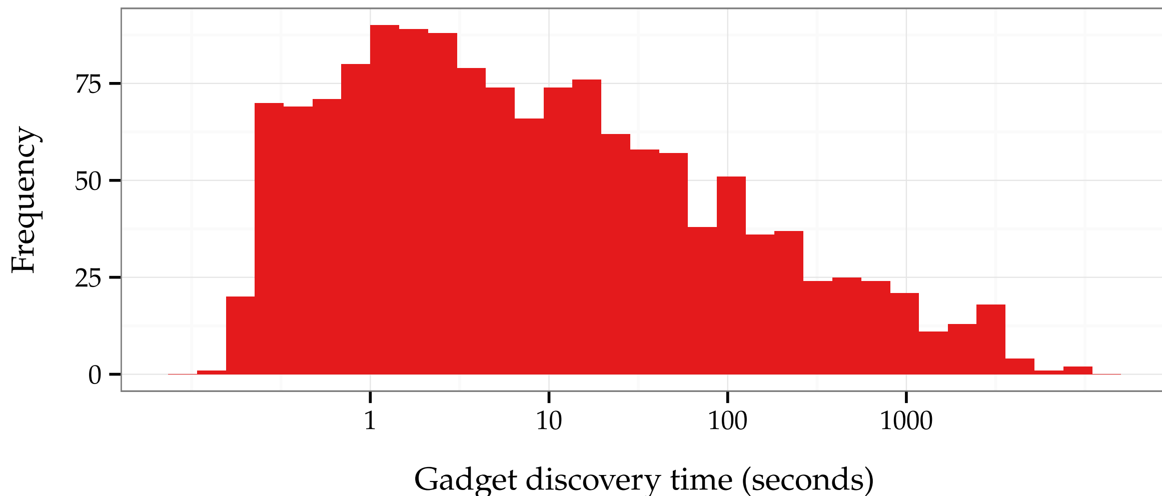


Figure 4.10: Distribution of gadget discovery times.

Arrangement and Assignment As can be seen in Figure 4.12, the time required to arrange and assign gadgets depends both on the payload being generated, and whether or not the payload was successfully generated. On average, attempting to create a Call payload takes more time than for a Store (4.21 s vs. 2.51 s). This is expected, since a Call payload contains a superset of the Store functionality. The variation of failed payloads is also significantly lower than those of successful ones ($\sigma = 6.23$ s vs. 27.9 s). One possible explanation for this difference is that Q tries all possible assignments before failing, whereas when a satisfying assignment is found the search may stop earlier. Unlike gadget discovery, arrangement and assignment times show no clear correlation with the program size ($r = 0.00702$), as can be seen in Figure 4.13.

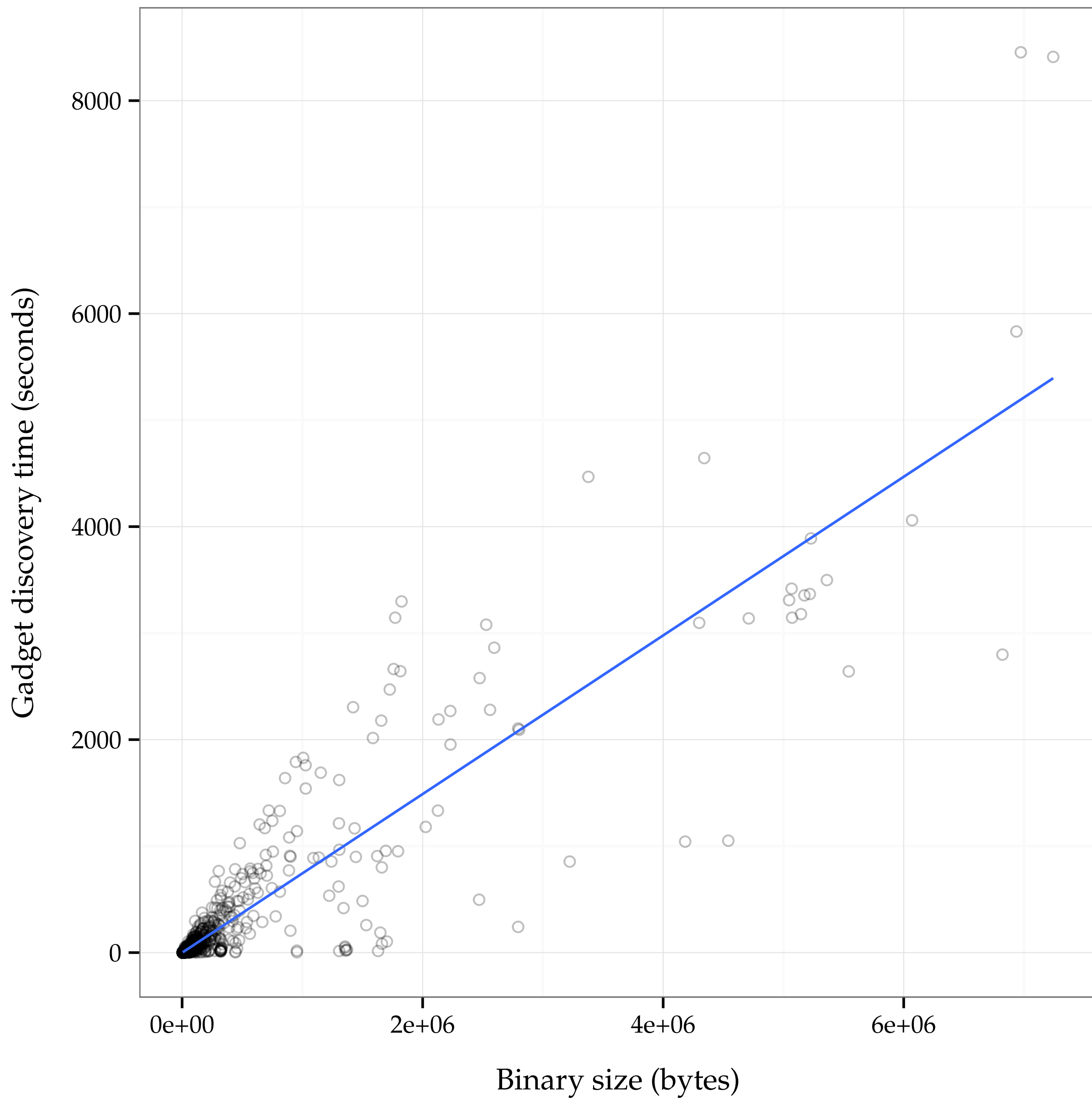


Figure 4.11: Gadget discovery time compared to binary size. $r = 0.894$.

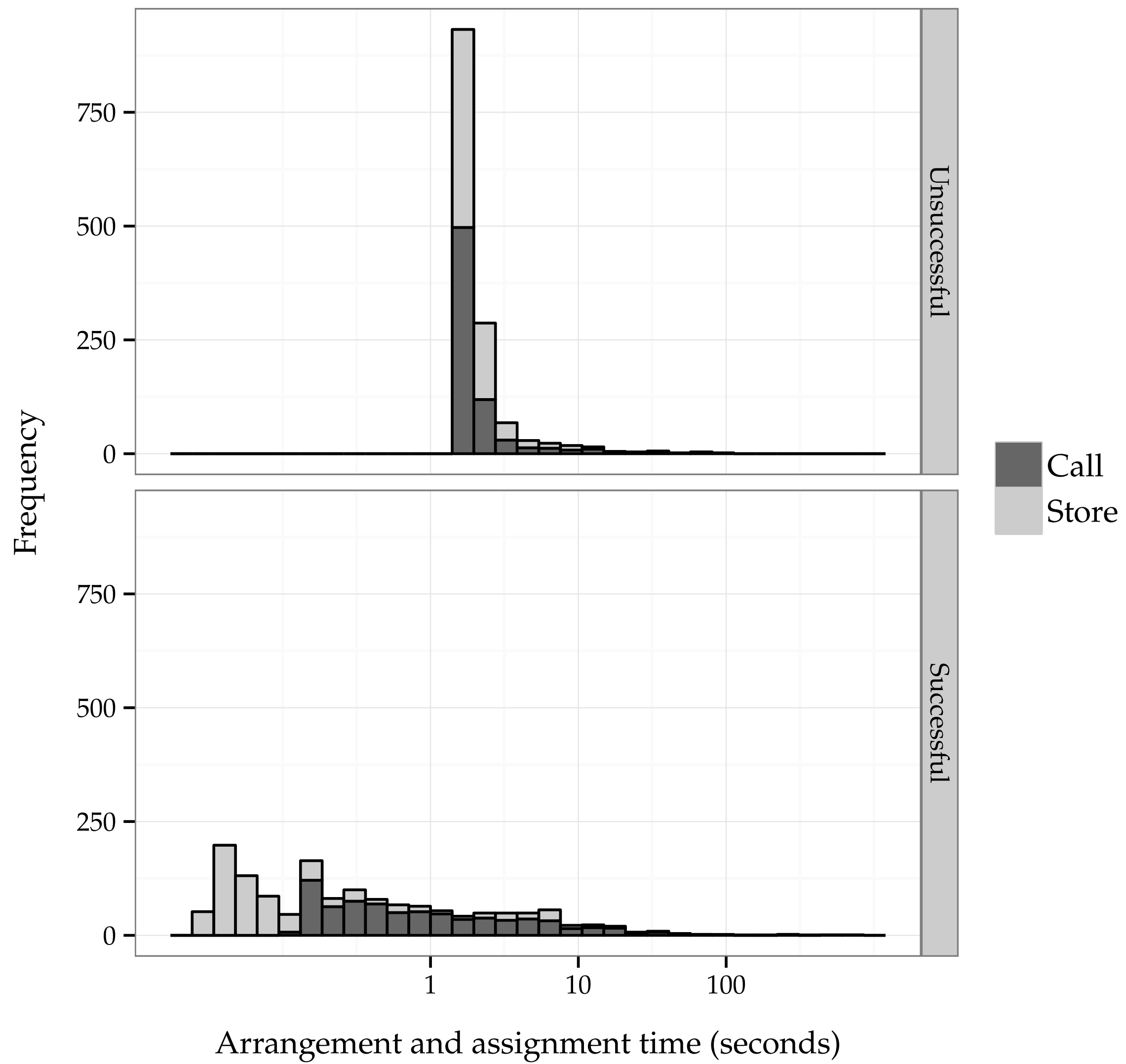


Figure 4.12: Distribution of time elapsed during gadget arrangement and assignment, organized by payload type and whether a payload was successfully produced.

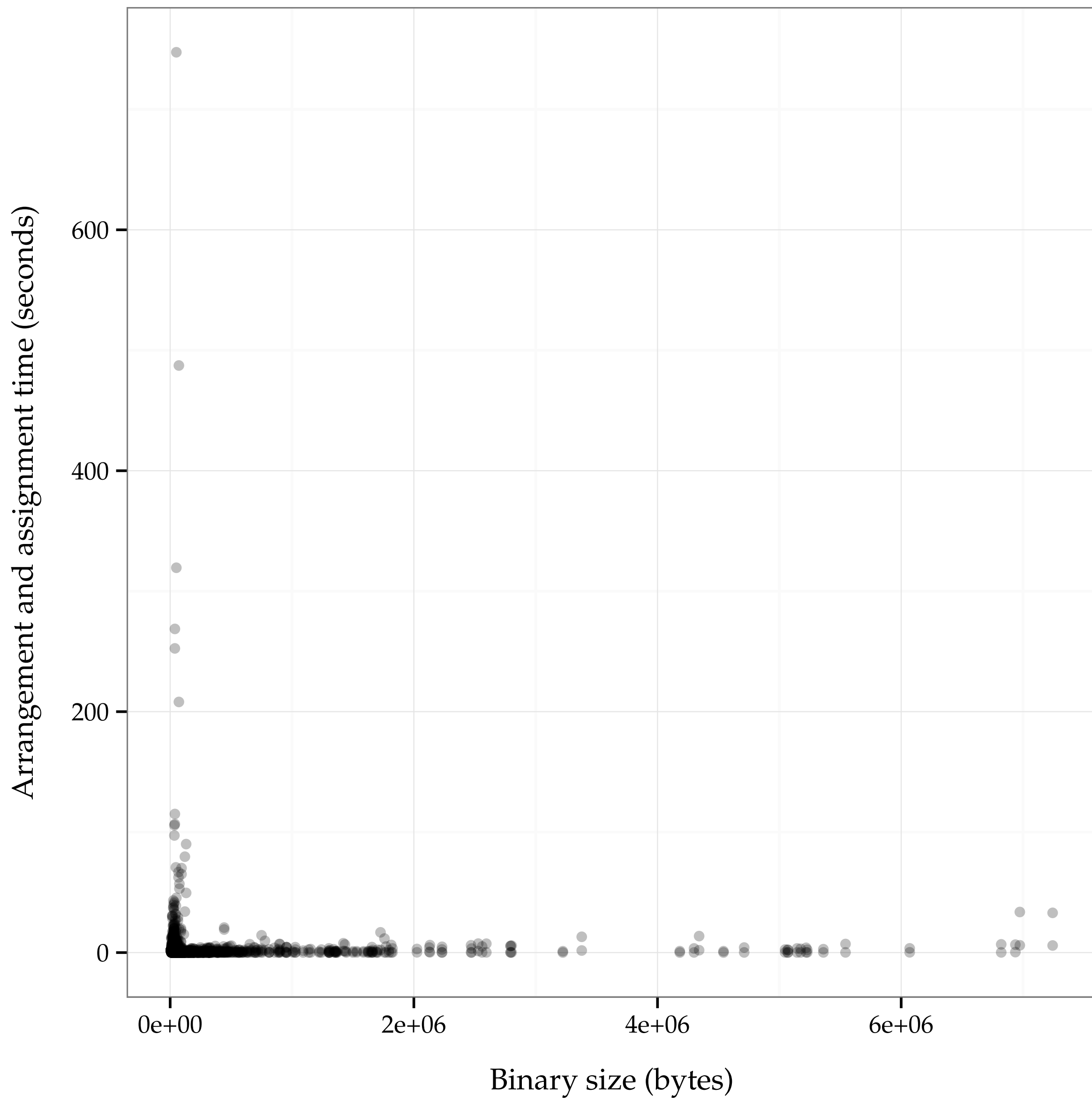


Figure 4.13: Time elapsed during gadget arrangement and assignment as a function of program size. ($r = 0.00702$).

4.7.3 How large are Linux programs?

The results from Section 4.7.1 suggest that the likelihood of generating a ROP payload is highly correlated with the amount of unrandomized code. The unrandomized code on Linux includes the program image by default, and thus the distribution of program sizes influences how many programs are susceptible to ROP. If most programs are small, ROP may not be possible. Figure 4.14 shows the distribution of file sizes in the corpus of tested programs.

- 55% (793 of 1,429) of the programs analyzed are 20 kB or larger. 20 kB is the threshold at which Q can often generate ROP payloads.
- Only 5% (69 of 1,429) of the programs are at least the size of unrandomized code considered by prior automated ROP research (1.1 mB or larger). 1.1 mB corresponds to the size of the iPhone libsystem library, which is the smallest amount of code that prior automated ROP researchers have targeted [50]. Other automated ROP targets include libc [107], and the windows kernel [68]. These binaries are significantly larger than most `/usr/bin` programs.

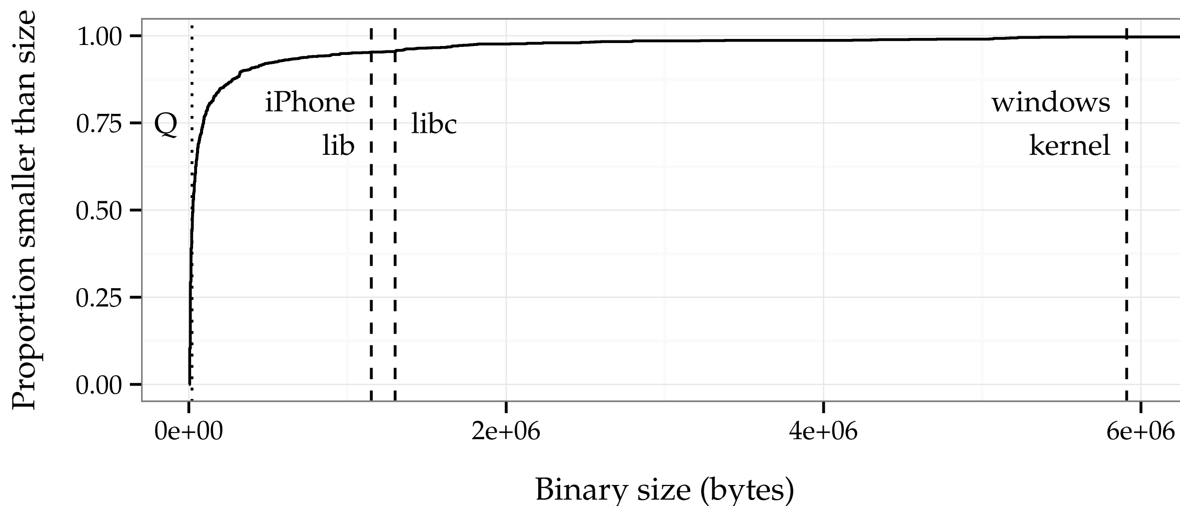


Figure 4.14: Empirical cumulative distribution function of the file sizes in `/usr/bin`. In this graph, a point at (x, y) signifies that $100y$ percent of the files in `/usr/bin` have a size less than or equal to x bytes. The sizes of the iPhone libsystem library [50], libc [107] and the windows kernel [68], which prior work has targeted, are all shown as vertical dashed lines. libc and the iPhone library are both larger than 95% of the programs in our corpus, while the windows kernel is larger than 99%. A dotted line also marks 20 kB, which is the size at which Q can generate payloads for 85% of programs.

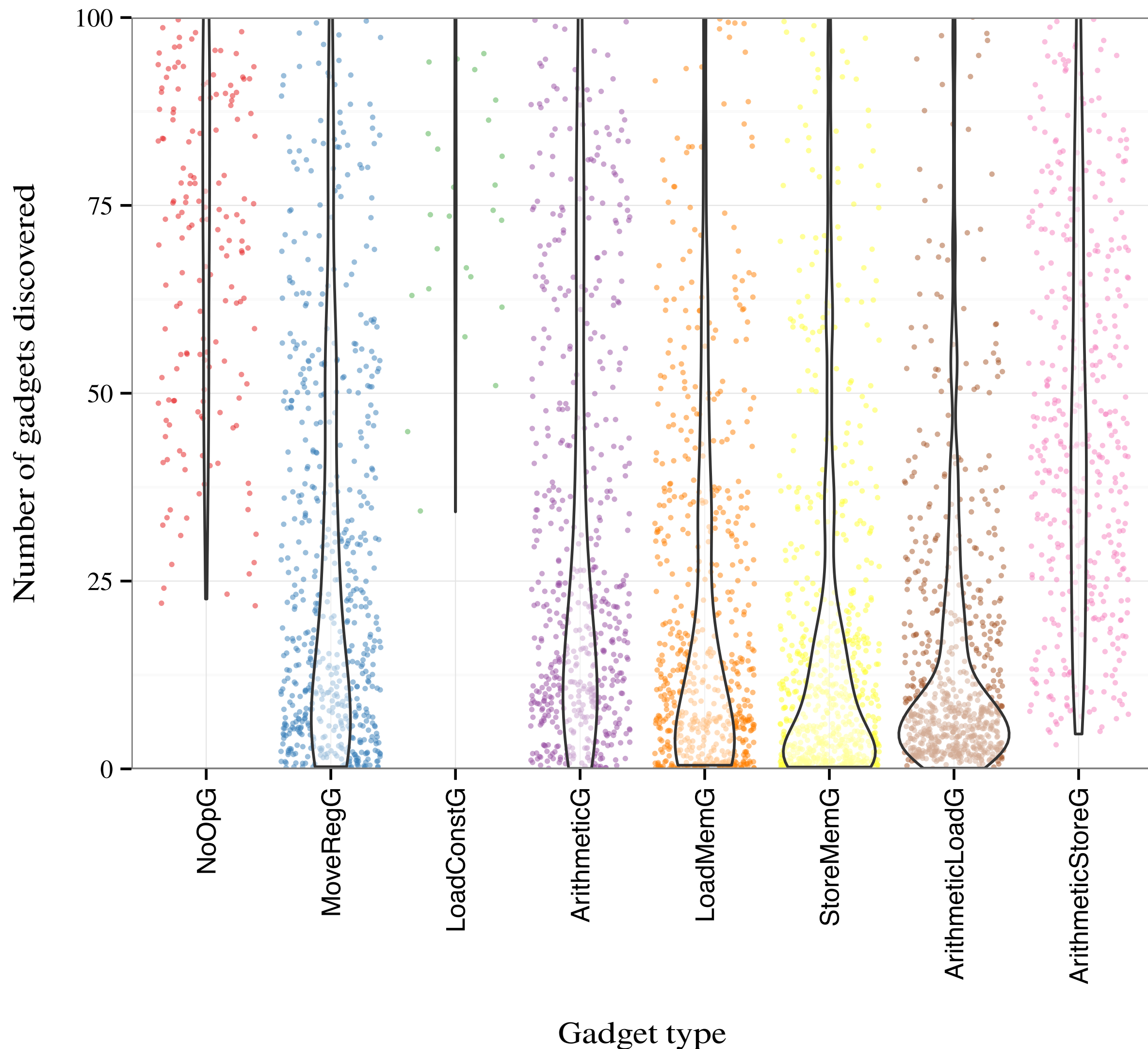


Figure 4.15: Number of gadgets discovered in each `/usr/bin` program larger than 20 kB, categorized by gadget type. Results for programs containing over 100 gadgets of the same type are truncated from the figure.

4.7.4 How frequent is each gadget type?

Figure 4.15 shows the frequency of various types of gadgets in programs larger than 20 kB. The same gadget located at multiple addresses is counted more than once.

- Many programs have no StoreMemG gadgets. However, every program larger than 20 kB had at least one ArithStoreG gadget. This highlights the importance of considering alternate gadget arrangement rules (Section 4.4.2).
- MoveRegG, ArithmeticG, LoadMemG, and ArithmeticLoadG gadgets are sometimes not found. Thus it is important not to depend on these gadget types.

- NoOpG and LoadConstG gadgets are extremely prevalent, although the results are cutoff in the figure.

4.7.5 Is Exploit Hardening Possible for Real Exploits?

Q's exploit hardening capabilities were tested on a variety of publicly available exploits for Linux and Windows. Each experiment is considered a success if Q can harden the public exploit by producing a working exploit that correctly functioned on a test machine with DEP and ASLR enabled.

The experimenter compiled each vulnerable program from source code (when possible), disabled all defenses (including ASLR and DEP), and then verified that the exploit at least crashed the vulnerable program. He then ran the exploit through the exploit hardening component of Q, and created Call-Local and Call-External payloads (Section 4.7.1). These payloads 1) call a linked function and 2) call `system("w")` on Linux or `WinExec("calc.exe")` on Windows. The experimenter then verified each exploit operated correctly with ASLR and DEP enabled. The results of these experiments are shown in Table 4.4.

In summary:

- Q was able to harden nine exploits automatically.
- Some exploits were hardened for Windows 7, and others for Linux.
- Exploit hardening is fast. The longest exploit to harden took slightly over six minutes.

Program	Vulnerability	Tracing	Analysis	Call Linked	Call System	OS	SEH
Free CD to MP3 Converter	OSVDB-69116	89s	41s	Yes	Yes	Win	No
FatPlayer	CVE-2009-4962	90s	43s	Yes	Yes	Win	Yes
A-PDF Converter	OSVDB-67241	238s	140s	Yes	Yes	Win	No
A-PDF Converter	OSVDB-68132	215s	142s	Yes	Yes	Win	Yes
MP3 CD Converter Pro	OSVDB-69951	103s	55s	Yes	Yes	Win	Yes
rsync	CVE-2004-2093	60s	5s	Yes	Yes	Lin	NA
opendchub	CVE-2010-1147	195s	30s	Yes	No	Lin	NA
gv	CVE-2004-1717	113s	124s	Yes	Yes	Lin	NA
proftpd	CVE-2006-6563	30s	10s	Yes	Yes	Lin	NA

Table 4.4: Public exploits hardened by Q. For each exploit, the time elapsed trace and analysis components took to run, and report if Q produced hardened exploits that call 1) a linked function, and 2) `system` or `WinExec`.

4.8 Discussion

4.8.1 Ret-less ROP

Before the design of Q, no one had shown that ROP was possible without using `ret`-like instructions. Since then, Checkoway, et al. have shown [30] that it is possible to create a Turing-complete gadget set that does not

use `ret` instructions. Their gadgets have control flow preservation preconditions. For example, the gadget `pop %eax; jmp *%edx` only preserves control flow if `%edx` is preset to the next gadget address. Q's model of gadgets does not allow for these types of preconditions, which prevents it from finding such gadgets. In theory, the automatic code reuse system described in Section 5.2.2 can find gadgets with preconditions, but is too inefficient to be practical.

4.8.2 Side Effects

Q conservatively handles side effects by discarding any instruction sequence that might cause the program to crash, such as a pointer dereference. As one example, `pushl %eax; popl %ebx; ret` will move the value in `%eax` to `%ebx`. Since a MoveRegG gadget does not intentionally use memory, however, Q would discard this gadget. Proper memory modeling was never added, because Q was successful as is. The automatic code reuse system in Section 5.2.2 precisely reasons about side effects. However, it scales so poorly that it is not a practical tool.

4.8.3 Turing-completeness

Q's language for describing target programs, QooL, is not Turing-complete. Our early tests revealed that the ArithmeticG gadgets needed for conditional jumps, such as equality tests, were often unavailable in small programs. As a result, work on Q focused on the gadgets needed for practical exploitation, rather than striving for Turing-completeness. In most cases, it is possible to call libc functions to disable DEP and execute arbitrary binary code, which is obviously Turing-complete.

4.9 Related Work

Return-Oriented Programming Krahmer was the first to propose using borrowed code chunks [78] from the program text to perform meaningful actions. Later, Shacham showed in his seminal paper [115] on ROP that a set of Turing-complete gadgets can be created using the program text of libc. Shacham developed an algorithm that put instruction sequences into trie form to help a human manually select useful instruction sequences.

Since then, several researchers have investigated how to automate ROP [50, 68, 107]. Dullien and Kornau [50,77] automatically found gadgets in mobile support libraries (on order of 1,000KB), and Roemer [107] demonstrated it was possible to automatically discover gadgets in libc (1,300KB). Hund [68] used gadgets from `ntoskrnl.exe` (3,700KB) and `win32k.sys` (2,200KB). In contrast, Q often produces payloads with only 20 kB of binary code to create gadgets from. This is important in practice because small code modules

are often the only unrandomized code in modern exploitation contexts. Previous work focusing on such small code bases was mostly or entirely manual; for instance, Checkoway, et al. manually crafted a Turing-complete set of gadgets from 16 kB of Z80 BIOS [31].

Automatic Exploitation Exploit hardening (Section 4.5) is related to existing automatic exploitation research [7, 23, 64, 85]. In automatic exploitation, the goal is to automatically find an exploit for a bug when given some starting information (such as a patch [23], guiding input [64, 85], or program precondition [7]). Some automatic exploitation research focuses on creating an input that triggers a particular vulnerability [23, 57, 85], but does not focus on control flow exploitation, which is a focus of exploit hardening. Q can use the inputs produced by these projects as an input exploit, and harden them so that they bypass DEP and ASLR.

Heelan [64] also considered the problem of creating an exploit when given another exploit; in his case the input exploit only causes a crash. Q uses symbolic execution to reason about other inputs that take the same path as the input exploit. In contrast, Heelan tracks data dependencies between the desired payload bytes and the input bytes, but does not ensure that control flow will stay the same and preserve the observed data dependencies. As a result, his approach is heuristic in nature, but is likely to be faster.

Related Attacks Other researchers have previously used simple ROP gadgets in the `.text` section of binaries to calculate the address of functions in libc [108]. Unfortunately, this is insufficient to make arbitrary function calls when ASLR is enabled, because many functions require pointers to data. Recall from Section 4.2 that all modern operating systems randomize the stack and heap, thus making it difficult for an attacker to introduce argument data and know a pointer to its address. QooL (Section 4.4.2) allows target programs to write payloads to known addresses, typically in the `.data` segment, which eliminates this problem.

A recent attack developed concurrently with Q [87] can also write data to known constant memory locations, and thus can also make arbitrary function calls in the DEP and ASLR setting. This attack uses repeated `strcpy` return-to-libc calls to copy data from the binary itself to a specified location. In contrast, Q uses ROP gadgets.

There are also specialized attacks against DEP and ASLR that are only applicable inside of a browser, such as JIT spraying [17, 120]. The downside is that they are not applicable to all programs.

Related Defenses The most natural way of defeating ROP is to randomize all executable code. For instance, Q is unable to deterministically attack position independent executables in Linux, because it does

not know where the instruction sequences will be in memory. Operating systems have chosen not to randomize all code in the past because of performance and compatibility issues; these reasons should now be reevaluated considering the new evidence that allowing even small amounts of unrandomized code can enable an attacker to use ROP payloads [110].

ROP attacks can also be limited by enforcing control flow integrity [1]. Control flow integrity ensures that control transfers must respect the intended control flow of the program. ROP gadgets are generally not part of the intended control flow, and thus will be disallowed.

Other defenses that are more specific to ROP have also been proposed. One defense is to dynamically instrument running programs and look for sequences of instructions that contain returns with few instructions spaced between [32,41]. The assumption is that normal code will generally execute non-trivial amounts of code in between `ret` instructions, whereas ROP code will not.

A similar defense is to ensure that the call chain of a program respects the stack semantics, i.e., that a `ret` will only transfer control to a program location that previously executed a `call` instruction. Such techniques [42, 104] are implemented using a shadow stack that is maintained outside of normal memory space.

Unfortunately for defenders, researchers [30] have recently shown that it is possible to perform ROP on x86 without using `ret` instructions at all, which is enough to bypass the last two classes of defenses [32, 41, 42, 104]. However, the proof of concept techniques [30] required access to large libraries, which are randomized in modern operating systems. It remains an open question whether such attacks are possible in modern user-mode exploitation contexts, where little unrandomized code is available.

4.10 Conclusion

This chapter developed return-oriented programming (ROP) techniques that recover gadget abstractions from small, unrandomized code bases as found in modern systems. Using the recovered gadget abstractions, it is possible to synthesize ROP payloads for 85% of programs larger than 20 kB, implying that even a small amount of unrandomized code is harmful. It is also possible to automatically read as input an exploit that does not bypass defenses, and automatically harden it to one that bypasses ASLR and DEP. These results demonstrate that current ASLR and DEP implementations, which allow small amounts of code to be unrandomized, continue to allow ROP attacks. Operating system designers should carefully weigh the dangers of such attacks against the performance and compatibility penalties imposed by randomizing all code by default.

Chapter 5

Scalability

This chapter compares the scalability of abstraction recovery-based analyses to low-level static binary analyses that do not employ abstraction. Advances in static binary analysis have been promising, but scalability remains one of the primary challenges. In general, abstraction improves efficiency and scalability by eliminating details that are irrelevant to the property being considered; the right abstraction can make analysis faster and use fewer resources. Unfortunately, the converse also explains why static binary analysis is so difficult to scale: binary code has little abstraction, even when it implements an abstract program. Instead, each operation is described in detail, which is bad for efficiency. The high-level motivation for abstraction recovery is that it is faster to analyze the recovered abstractions than the low-level program, which should improve scalability. It is not surprising that analyzing the abstract representation of the program is more efficient, but what is surprising is that, for at least the abstractions explored in this dissertation, the time required to recover abstractions is minor compared to the total analysis time. Thus, from a scalability point of view, abstraction recovery is preferable to low-level analysis for those abstractions.

The impact that source code abstractions have on scalability is measured by building analyses with the verification conditions (VC) algorithm developed in Chapter 3. VCs are a unique program analysis primitive in that they can operate on both low-level concrete program representations and abstract source code representations. Thus, VCs are generated for C programs, their compiled binary form, and the decompiled C program that Phoenix (Chapter 2) recovers from the compiled binary, and the performance of each is compared.

The scalability of gadget abstractions is evaluated differently. The Q system from Chapter 4 uses gadget abstractions, but unlike VCs, cannot be used directly without abstractions. Instead, Q is compared to a system for finding code reuse attacks, which are a generalization of gadget-based attacks. A dynamic symbolic executor, Mayhem, is used to search the symbolic state space of several simple vulnerable programs for

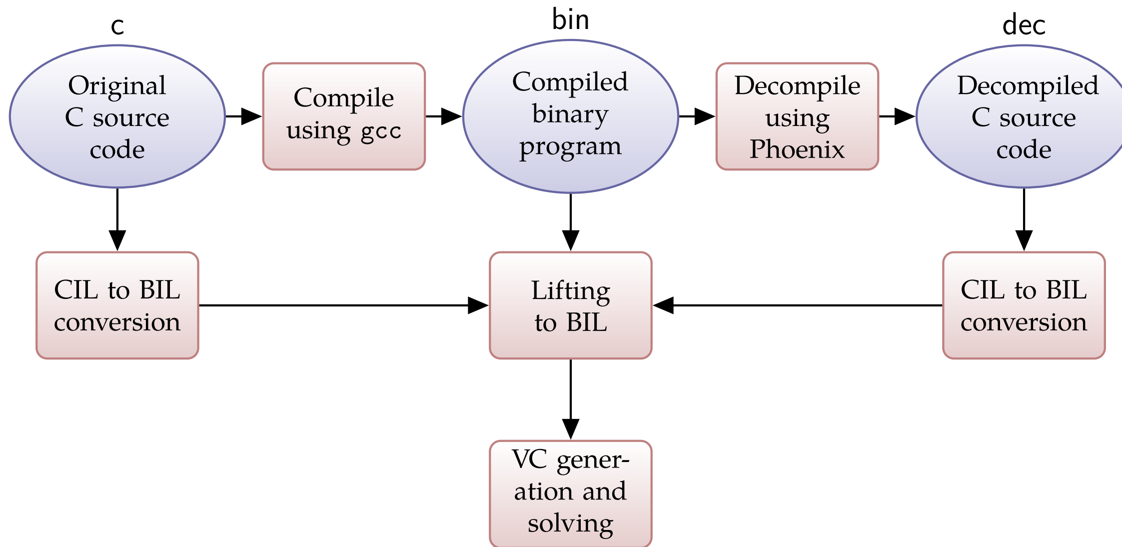


Figure 5.1: Overview of source-code scalability experiment.

specified goal states. Interestingly, Mayhem and Q often find very different attacks for the same program.

5.1 Source-Code Abstractions

This section investigates the impact of source-code abstractions on program analyses. Verifications conditions (VCs) (Chapter 3) are a natural choice for evaluating the performance impact of abstractions because VCs can analyze both abstract and low-level program representations, whereas many source analyses are defined over high-level abstractions like variables and types and cannot be used directly on binary programs. VC algorithms can accept both abstract and low-level program representations because they are defined over a simple language which both program representations can be converted to. Since VCs operate on the general structure of the program, the amount of abstraction in the program naturally effects the complexity of the formulas that are produced and the time required to create them. For example, one C statement may be implemented with several complicated assembly instructions, and the additional complexity at the assembly level produces larger and more complex formulas than the C program does.

VCs can also be used for a variety of applications, which makes the results widely applicable. This section specifically investigates correctness testing and buffer overflow detection, but there are many other applications of VCs as well, from automatic exploitation [7,8,28,112], automatic test case generation [26,60], to extended static checking [53,84]. These experiments are also representative of how these applications would likely perform.

As can be seen in Figure 5.1, the experiment operates on three different versions of the same program:

c, dec, and bin. The experiment begins with programs in the c representation, which is simply the C source code for each program. The programs evaluated are the same benchmarks described in Section 3.4.2 and listed in Table 3.1. The c representation for each program is compiled to a x86 binary using gcc 4.6.3, without optimization, to produce the bin representation. The bin program is then decompiled using Phoenix (Chapter 2) to produce the dec representation. A VC is then produced for each one of these three representations.The FVC implementation in the Binary Analysis Platform (BAP) [22] was used to produce VCs for all three representations from the BAP intermediate language (BIL). BAP was used to lift the bin representation to BIL, and the CIL-based C to BIL converter (Section 3.4.2) was used to convert the c and dec programs to BIL using variable and type abstractions. The time required to decompile the dec representation is counted towards the dec VC generation time. The generated VCs were solved using cvc4. cvc4 was selected because it consumed predictable amounts of time while solving formulas, which resulted in figures that demonstrated relationships more clearly than those for the other solvers considered (z3, yices, and boolector).

All experiments were performed on an Amazon EC2 m3.2xlarge instance, which features 8 hardware threads on an Intel Xeon E5-2670 processor with 30GB of ram. Our experiment script computed at most 8 VC instances at once (one per core). Each instance was limited to 60 minutes CPU time (for both formula generation and solving), 2GB of disk space and 2GB of memory usage.

Two types of VCs were generated: those for correctness testing and buffer overflow detection. When testing correctness, the postconditions specified in Table 3.4.2 were checked for validity. Checking for buffer overflows is more complicated, because binary programs do not have buffer abstractions, which requires binaries to be checked differently than the C-based representations. To check the bin representations for overflow, the saved return address on the stack was verified to be unmodified at the epilogue of each function. Since all buffers in the benchmarks are on the stack, this tests for the existence of a buffer overflow long enough to overwrite the return address. For the c and dec representations, CIL was used to statically instrument the programs so that each operation that reads or writes from a compound object does a bounds check, and updates an auxiliary has_overflow variable accordingly. This variable was then verified to always be false with the VC.

In both applications, loops are unrolled a variable number of times (reported below) and any remaining cycles are removed before FVC is called. Any execution that would execute a loop more iterations than loops are unrolled are assumed to be correct, or to not contain buffer overflows.

The goal of the experiment is to answer the questions in the following subsections.

5.1.1 Is recovering and analyzing abstract source code faster than analyzing the binary?

Figure 5.2 shows the total time required to generate and solve a VC that establishes the correctness of each benchmark program, up to the number of loop unrolls on the x-axis, for both bin and dec program representations. Unrolling loops allows more program executions to be considered, and also artificially increases the complexity of the program. The figure overwhelmingly shows that:

- When a program could be verified from both the bin and dec representations, verifying the dec representation was significantly faster on all but trivially small instances.
- Programs could be verified to a larger number of unrolls using the dec representation than the bin representation. In many cases, programs could not be verified for even a single unroll from the bin representation.

Figure 5.3 shows the same type of figure, but for VCs that test for buffer overflows instead of correctness. The results are largely the same, suggesting that application is unlikely to significantly effect the behavior of VCs.

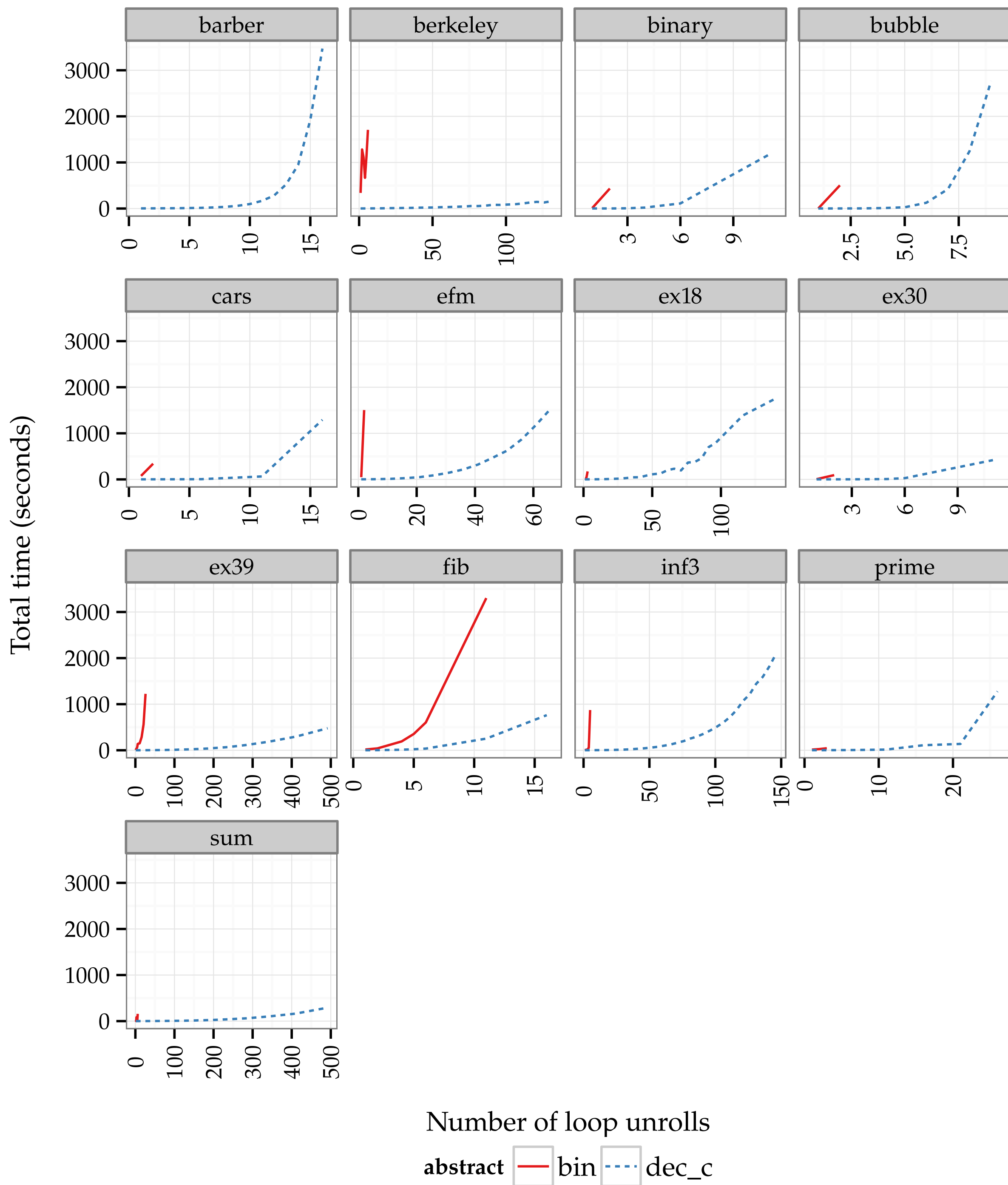


Figure 5.2: Total time required to verify correctness of benchmark programs compared to number of loop unrolls.

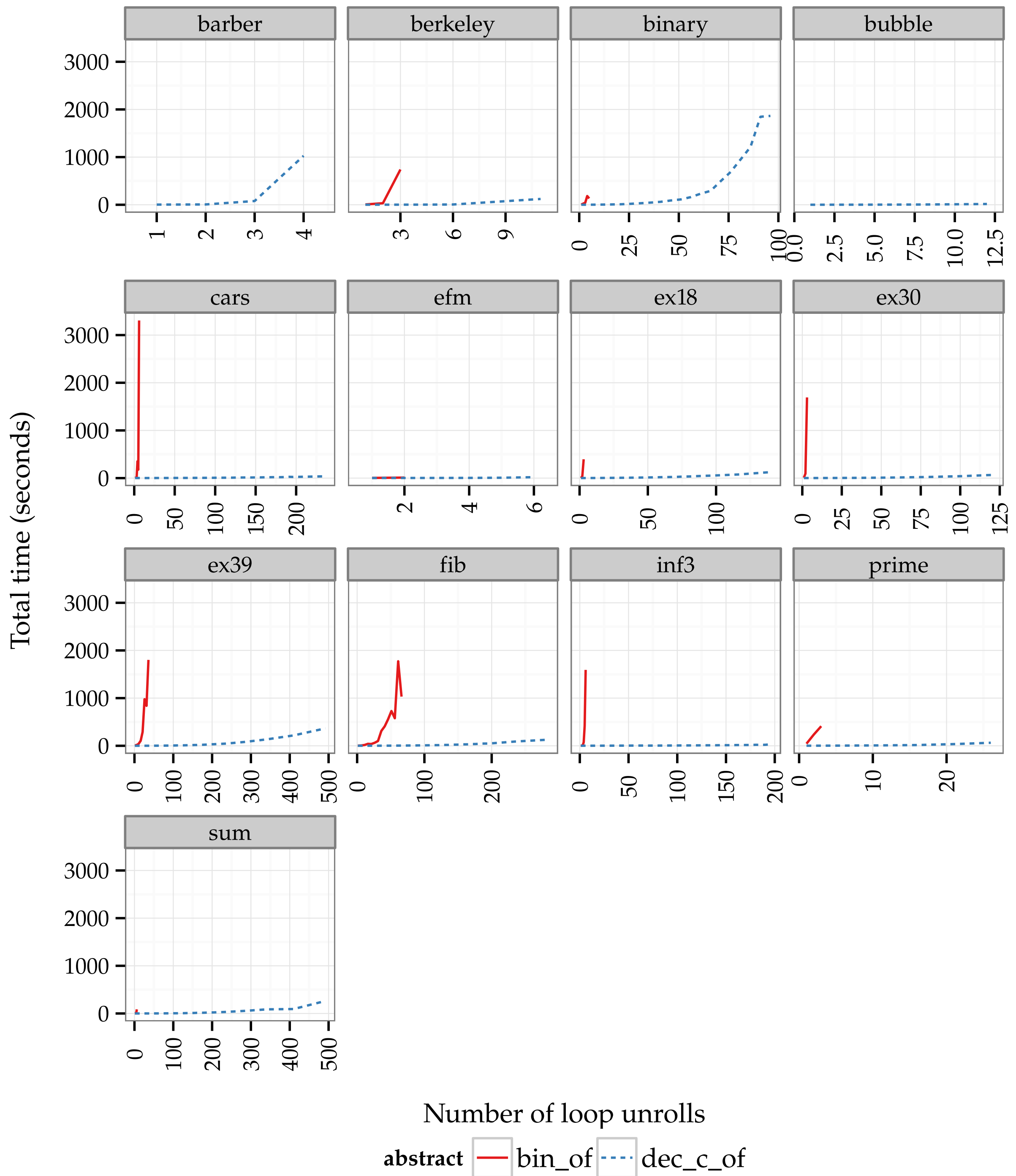


Figure 5.3: Total time required to test benchmark programs for buffer overflows compared to number of loop unrolls.

5.1.2 Why is recovering and analyzing source code faster than analyzing the binary?

The previous section established that VC-based analyses are more efficient when analyzing dec representations than bin programs, but not why. Figure 5.4, which shows the total analysis time as a function of formula size, suggests size is a possible factor. (Figure 5.4 and other figures in this section only report results for correctness; the results for overflow checking are similar and thus omitted.) Unsurprisingly, larger formula sizes generally lead to longer total analysis times. Figure 5.5 shows that bin programs do produce larger formula sizes than dec programs. Intuitively, this is because concrete representations include more details about the computation.

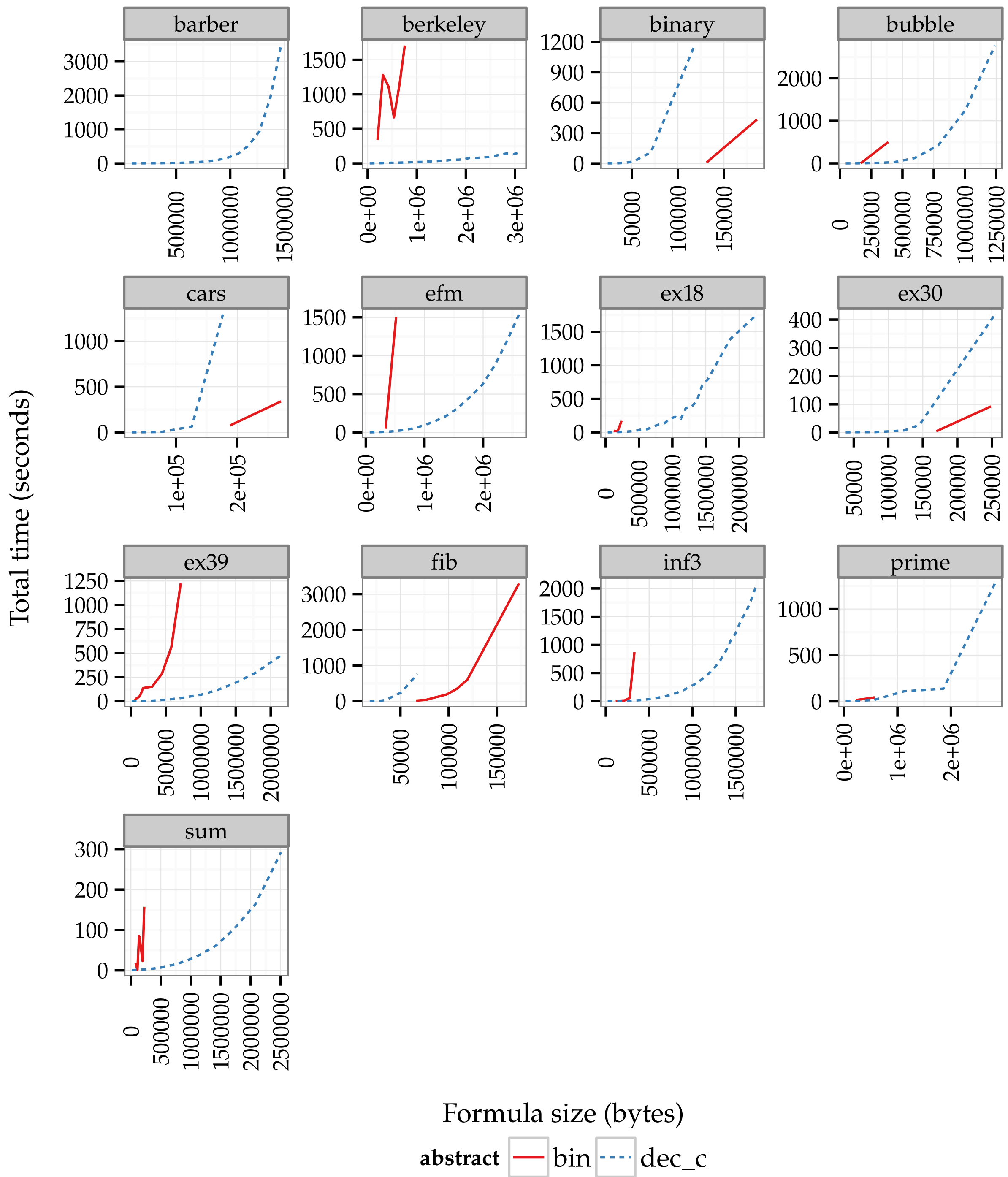


Figure 5.4: Total time required to verify correctness of benchmark programs compared to formula size.

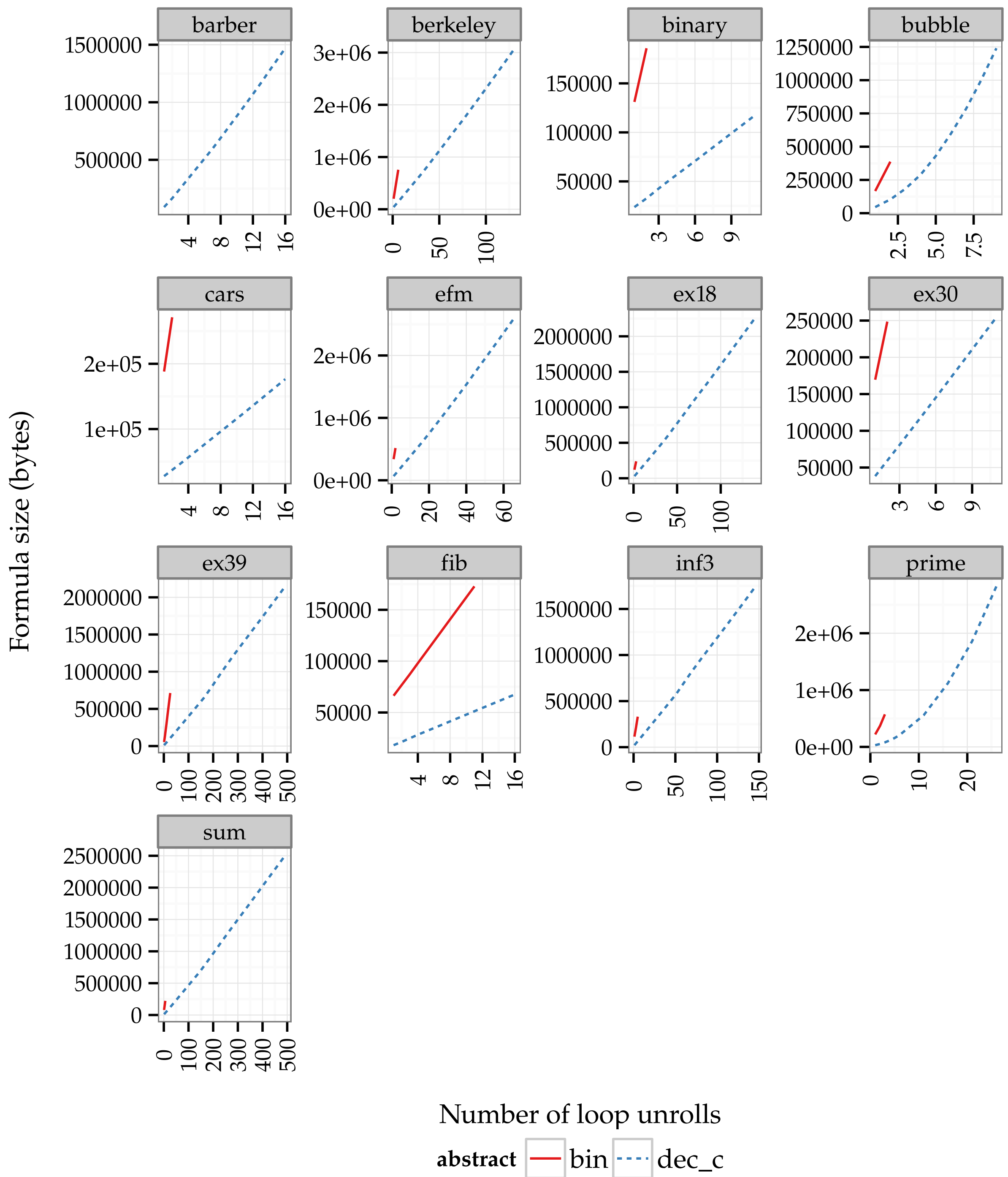


Figure 5.5: Formula size of VCs that verify correctness of benchmark programs compared to number of loop unrolls.

Formula size is not the only factor, however. The `inf3` benchmark is a good example. Figure 5.5 shows that both the bin and dec representations produced similar formula sizes, but Figure 5.4 shows the total analysis time for the bin programs is larger, which suggests additional factors.

To identify some of these factors, several artificial concretization functions were developed to simulate some of the effects of compilation:

notypes VCs use type abstractions to allocate precisely sized bitvector and memory (array) variables. notypes simulates the loss of such abstractions by changing each variable to a very large array. Although type abstractions are removed, each variable in the program still has a separate variable in the transformed program.

novars VCs use variable abstractions to give distinct variables in the program distinct variables in the resulting formula. novars removes these abstractions by moving all variables to a stack. This is similar to how most compilers implement local variables. Type abstractions are used to properly allocate positions on the stack.

inflate The previous results showed a correlation between formula size and analysis time. One explanation is that larger formulas *cause* a longer analysis time. Another explanation is that more complex programs require larger formulas to describe, and the complexity is what increases the longer time. The inflate transformation tests the earlier explanation by artificially inflating the program size. It does this by adding dead statements to the program.

Figure 5.6 shows the time to verify correctness as a function of unrolls, for the bin, dec, notypes, novars, and inflate program representations. It is clear that all three transformations have a noticeable effect on the verification time. notypes and novars often timeout early, making it difficult to observe their effect from the figure. However, the maximum number of unrolls successfully completed by each transformation and the difference from the bin representation is reported in Table 5.1.

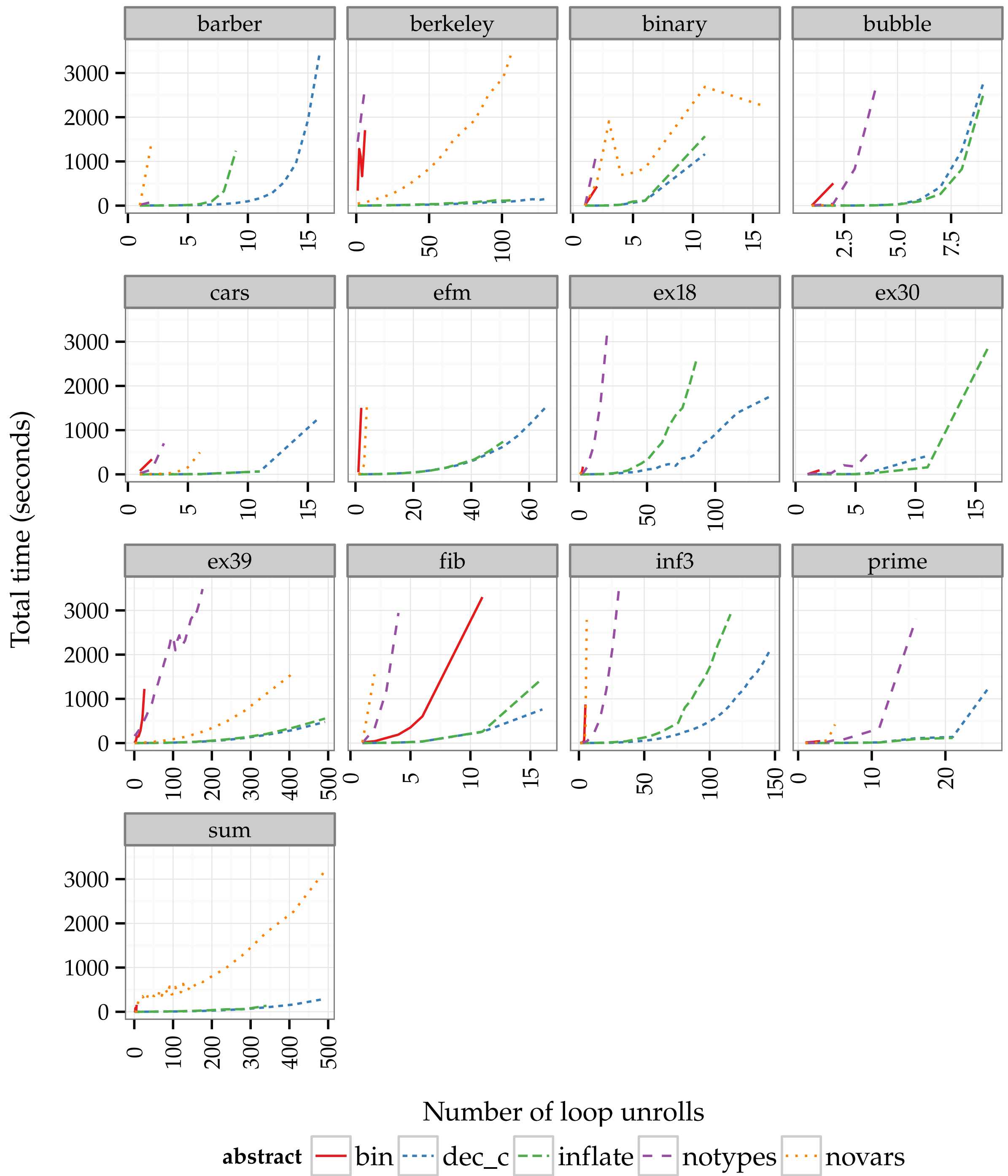


Figure 5.6: Total time required to verify correctness of benchmark programs compared to number of loop unrolls. The program representations include artificial transformations that simulate the loss of abstractions.

Program	bin		inflate		notypes		novars		dec	
	#U	Δ	#U	Δ	#U	Δ	#U	Δ	#U	Δ
barber	1	0	9	8	2	1	2	1	16	15
berkeley	6	0	106	100	6	0	106	100	131	125
binary	2	0	11	9	2	0	16	14	11	9
bubble	2	0	9	7	4	2	2	0	9	7
cars	2	0	11	9	3	1	6	4	16	14
efm	2	0	51	49	1	-1	4	2	66	64
ex18	3	0	86	83	21	18	2	-1	140	137
ex30	2	0	16	14	6	4	1	-1	11	9
ex39	26	0	492	466	176	150	410	384	492	466
fib	11	0	16	5	4	-7	2	-9	16	5
inf3	5	0	116	111	31	26	6	1	146	141
prime	3	0	21	18	16	13	5	2	26	23
sum	6	0	341	335	6	0	492	486	492	486
All programs										
Total	–	0	–	1214	–	207	–	983	–	1501
Median	–	0	–	18	–	1	–	2	–	23
St. dev.	–	0	–	144	–	41	–	163	–	168
With `ex39` and `sum` excluded										
Total	–	0	–	413	–	57	–	113	–	549
Median	–	0	–	14	–	1	–	1	–	15
St. dev.	–	0	–	41	–	10	–	30	–	57

Table 5.1: Maximum number of unrolls before failure for bin, dec, and artificial transformations notypes, novars, and inflate. #U represents the maximum number of unrolls completed, and Δ represents the difference of #U for that representation and the bin representation.

Table 5.1 indicates that all three transformations have a high error when predicting the maximum number of unrolls a program will be verified for at the binary level. However, the median for notypes and novars are low, suggesting that outliers may be a factor. When excluding the `ex39` and `sum` programs, notypes and novars perform much better. One potential cause is that these two programs exhibit a large amount of concrete reasoning, which differentiates them from the other programs.

Even with `ex39` and `sum` excluded, inflate is still a poor predictor, which is consistent with Figure 5.6. This suggests that program size is *not* a cause of longer analysis times. Instead, the correlation between size and analysis time is likely caused by complexity. Concrete programs include more details, and this complexity increases the size of the program and makes the formulas harder to solve, which increases the total analysis time.

notypes and novars perform better as predictors, indicating that both type and variable abstractions improve analysis time, which is not surprising. novars produces formulas that do not have variables corre-

sponding to the original program's value. Instead, every time a variable in the original program is written or read, a memory operation is performed. The solver must decide which reads and writes alias, or overlap, which is a difficult problem. notypes produces formulas that have distinct variables, but each assignment in the original program is still implemented as a memory operation in the new program. Unlike the novars transformation, each memory operation begins at index zero however. Somewhat surprisingly, notypes is a better predictor than novars. This suggests that the solver still spends a significant amount of time reasoning about memory aliasing, even though scalar variables in the original program will always be read and written to the same position.

5.1.3 How does analyzing the decompiled source code compare to the original source code?

The previous section demonstrated that VC-based analyses are more scalable when analyzing decompiled source code than low-level binary code. This section compares how efficiently such analyses can analyze decompiled source code compared to the original source code. Ideally, if most abstractions are recovered by the decompiler, the difference should be small. A large difference would suggest the decompiler is not recovering some abstractions.

Figures 5.7 and 5.8 demonstrate the total analysis time for checking correctness and overflow, for the original C source code and the source code recovered by Phoenix. The results are surprisingly complex. A few programs are analyzed more quickly and scale to more loop iterations from the decompiled source code than the original. A few programs exhibit no noticeable differences between the representations. However, several programs in the overflow experiments scale to more iterations using the original source code than the decompiled source code. This raises the natural question of why this occurs.

Tables 5.2 and 5.3 show the stage of analysis and the number of loop unrolls for each failure. Most of the benchmarks that have a substantial difference in the number of loop unrolls for the c and dec representations failed during the formula generation stage. This typically means that the formula generation process ran out of resources, such as memory or disk space, rather than timing out. The resources required to unroll the program and keep the representation in memory is the most likely cause: the size of decompiled programs in Tables 5.2 and 5.3 are always larger than the original, and this difference is amplified by unrolling loops. In contrast, the bin representation fails during solving for all benchmarks. These results show that Phoenix is recovering an abstract representation of the program, but not the most compact representation. Manual examination of the difference between the c and dec representations revealed that the increase in size is primarily a result of extra casts that are added because of uncertainty in the TIE type inference system (Section 2.3.1); the exact number of casts is reported in Tables 5.2 and 5.3.

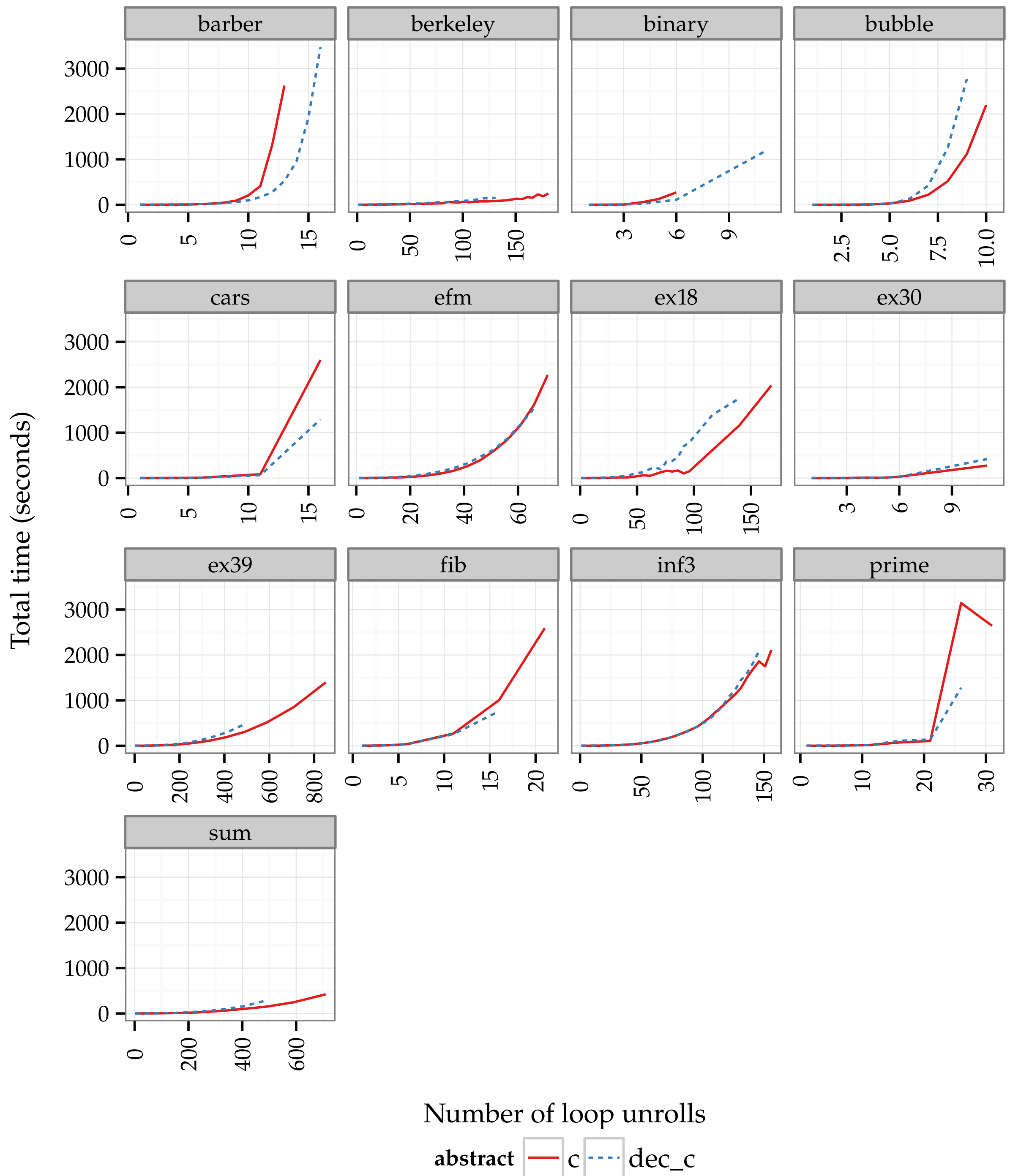


Figure 5.7: Total time required to verify correctness of benchmark programs compared to number of loop unrolls on original and recovered source code.

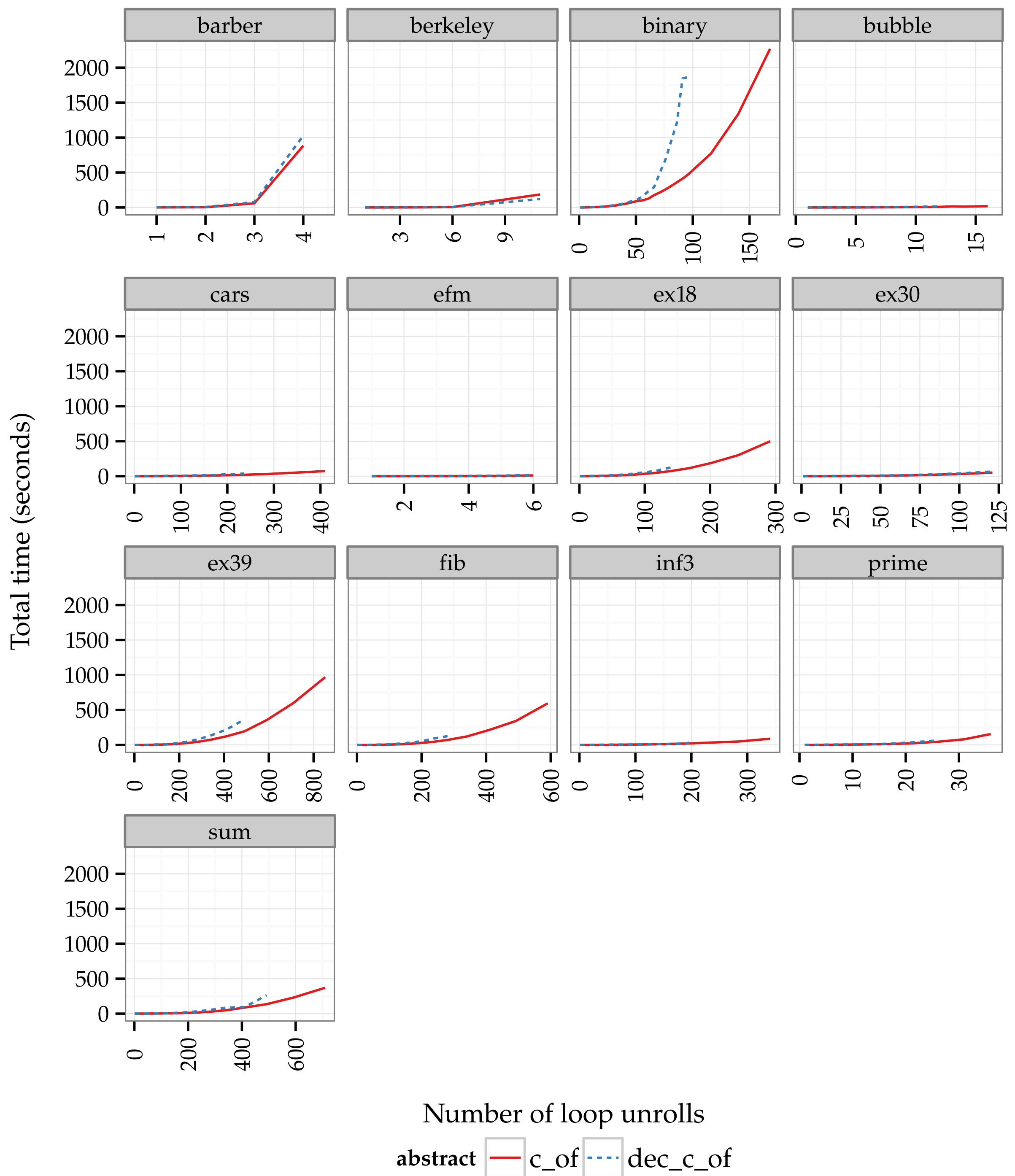


Figure 5.8: Total time required to test benchmark programs for buffer overflows compared to number of loop unrolls on original and recovered source code.

Program	c				dec_c				bin			
	S	C	U	F	S	C	U	F	S	C	U	F
barber	15634	24	14	solving	18372	84	17	solving	86264	N/A	2	solving
berkeley	5105	6	186	generation	6584	25	136	generation	34310	N/A	11	solving
binary	3896	10	11	solving	6544	41	16	solving	23367	N/A	3	solving
bubble	5235	18	11	solving	8815	54	10	solving	17383	N/A	3	solving
cars	6054	2	21	solving	7199	21	21	solving	54492	N/A	3	solving
efm	8631	10	76	solving	11060	39	71	solving	63494	N/A	3	solving
ex18	4354	8	202	solving	6336	36	168	generation	21577	N/A	4	solving
ex30	5928	11	16	solving	8457	48	16	solving	35432	N/A	3	solving
ex39	2216	2	1023	generation	2971	16	591	generation	15353	N/A	31	solving
fib	4514	16	26	solving	7776	49	21	solving	17603	N/A	16	solving
inf3	4086	6	161	solving	5442	30	151	solving	19779	N/A	6	solving
prime	4932	8	36	solving	7202	40	31	generation	29253	N/A	4	solving
sum	1738	3	852	generation	2656	18	591	generation	12103	N/A	11	solving

Table 5.2: Failures during verification. Data reported includes the program size in characters before unrolling (S), the number of cast operations before unrolling (C), the number of unrolls that each benchmark failed at (U) and the cause of failure (F).

Program	c_of				dec_c_of				bin_of			
	S	C	U	F	S	C	U	F	S	C	U	F
barber	15634	24	5	solving	18372	84	5	solving	86264	N/A	2	solving
berkeley	5105	6	16	solving	6584	25	16	solving	34310	N/A	4	solving
binary	3896	10	202	solving	6544	41	116	solving	23367	N/A	6	solving
bubble	5235	18	17	generation	8815	54	13	generation	17383	N/A	2	solving
cars	6054	2	492	generation	7199	21	284	generation	54492	N/A	11	solving
efm	8631	10	11	solving	11060	39	11	solving	63494	N/A	3	solving
ex18	4354	8	351	generation	6336	36	168	generation	21577	N/A	4	solving
ex30	5928	11	126	solving	8457	48	126	generation	35432	N/A	4	solving
ex39	2216	2	1023	generation	2971	16	591	generation	15353	N/A	41	solving
fib	4514	16	710	generation	7776	49	341	generation	17603	N/A	71	solving
inf3	4086	6	410	generation	5442	30	236	generation	19779	N/A	11	solving
prime	4932	8	41	generation	7202	40	31	generation	29253	N/A	4	solving
sum	1738	3	852	generation	2656	18	591	generation	12103	N/A	11	solving

Table 5.3: Failures during overflow checking. Data reported includes the program size in characters before unrolling (S), the number of cast operations before unrolling (C), the number of unrolls that each benchmark failed at (U) and the cause of failure (F).

5.2 Gadget Abstractions

Return-oriented programming (ROP) attacks (Chapter 4) are a specialization of code reuse attacks. In a code reuse attack, the goal is to put the program into a goal state by only executing code that is part of the program. Code reuse attacks are applicable when DEP prevents an attacker from injecting new executable code into the program.

Unfortunately, the definition of code reuse attacks is very general, and does not suggest a specific strategy or technique for attackers to find such attacks. The difficulty of finding code reuse attacks was the motivation for ROP attacks, which are code reuse attacks that only execute sequences of instructions ending in `ret` (or similar instructions). ROP attacks are a strict subset of code reuse attacks, which means that a program could be invulnerable to a ROP attack but still be vulnerable to a different code reuse attack. However, the advantage of ROP is that the notion of gadgets lends itself to a technique simple enough that humans can manually create attacks.

This section compares the scalability of the ROP attacks from Chapter 4 to the more general code reuse attack. First, a theoretical worst-case analysis shows that the search space grows exponentially in the number of indirect jumps that are taken. Second, the Mayhem binary symbolic executor is used to actually search the state space for code reuse attacks.

5.2.1 Theoretical Analysis

This section finds a lower-bound on the size of the search space for a code reuse attack. The attacker can put the program in different concrete states by sending distinct inputs, and in many cases the attacker can put the program in too many different concrete states for an explicit search to be effective. For instance, if all inputs that are 200 bytes long trigger a vulnerability, which is common for overflow vulnerabilities, then the attacker can put the program into $2^{200*8} = 2^{1600} \approx 10^{480}$ states after the vulnerability. Explicitly searching such a large state space is infeasible, since there are only an approximated 10^{80} atoms in the universe.

Symbolic analysis is an analysis technique that can cope with such large program state spaces [24,75]. Chapter 3 described an algorithm for generating verification conditions, which is a form of symbolic analysis that can statically reason about an entire program at once. Unfortunately, a static technique like this requires a static control flow graph of the program in advance, and in this case the control flow graph of the vulnerable program is not known. The code reuse problem instead lends itself to a dynamic symbolic executor, which can reason about symbolic states representing one program path at a time. The symbolic state space is much smaller than the concrete state space, and the control flow graph of the program does not need to be known in advance.

Code reuse can be framed as a symbolic state reachability problem with the following parameters:

- σ is the initial symbolic state. Generally, σ will immediately succeed a vulnerability that gives the attacker control over the instruction pointer. If there are multiple paths to the vulnerability, the attacker may have a set of initial states instead.
- $\hookrightarrow$ is the transition relation between program states.
- $\hookrightarrow^*$ is the transitive closure of $\hookrightarrow$.
- G is the goal state function. Example goal states include storing 42 in the `%eax` register, or ensuring that a system call will be executed on all reachable paths.

The goal of the code reuse problem is to find a σ' such that $\sigma \hookrightarrow^* \sigma' \wedge G(\sigma')$.

Assume that the attacker has control of the instruction pointer in σ, and that she only redirects control to unrandomized, executable program locations, which allows her to construct a deterministic attack in the presence of ASLR. Let e be the number of unrandomized executable program locations. Then there are e symbolic program states σ' that may lead to the goal state and $\sigma \hookrightarrow \sigma'$. Let F be the fraction of these that can reach an indirect jump (there exists a σ'' such that $\sigma' \hookrightarrow^* \sigma''$ and σ'' immediately precedes an indirect jump). Some states will not reach an indirect jump because they terminate, crash, or execute an infinite loop. Let $b = F * s$ be the branching factor for each indirect jump; each time an indirect jump is reached, at least b other states can reach an indirect jump. Thus, there are at least b symbolic states σ' such that $\sigma \hookrightarrow^* \sigma'$ and contain at most one indirect jump. Each one of these b states can indirectly jump to another b statements, so there are at least b^2 states reachable from σ that execute two or fewer indirect jumps. More generally, there are at least b^m symbolic states reachable from σ that execute m or fewer indirect jumps. As long as $b = F * s > 1$, the number of symbolic states reachable from σ that execute at most m indirect jumps grows exponentially in the maximum number of indirect jumps traversed. If a large number of indirect jumps are required to reach the goal state, the search space will quickly become infeasibly large.

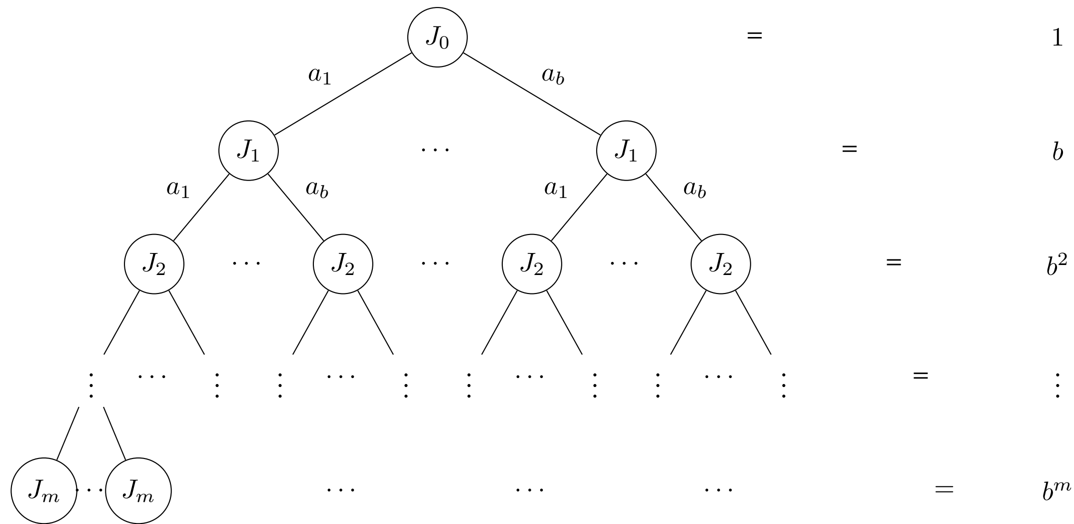


Figure 5.9: Symbolic state space in a code reuse attack. J_i represents the ith indirect jump. a_i represents the ith unrandomized executable address.

5.2.2 Code Reuse Attack Experiments

The last section established that the symbolic state space grows very quickly when indirect jumps are encountered. However, it did not establish how many symbolic states correspond to successful code reuse attacks. If there are code reuse attacks that only use a small number of indirect jumps, the search space will be small. On the other hand, if a large number of indirect jumps are required, the search space increases exponentially and will quickly become infeasible. Thus it is important to empirically measure how big the state space is for a real program, and additionally to estimate the number of code reuse attacks in the search space.

The Mayhem [28] dynamic binary symbolic evaluator was used to symbolically explore the state space for several synthetic vulnerable programs. Mayhem was modified for this experiment:

- Indirect jumps were restricted to unrandomized code addresses. This fits the assumption above that the attacker is only interested in addresses that she knows the bytes of.
- An option for detecting goal states (i.e., when $G(\sigma)$ is true) was added. Exploits for the goal state are emitted when found.
- A robust timeout mechanism was added. Unmodified versions of Mayhem became stuck when exploring strange behaviors caused by nonsensical executions. Examples include infinite loops and waiting for input.

Name	Gadget added	#Addrs
NullOverflow	None	4096
LoadconstOverflow	`pop %eax` `ret`	4096
StoreOverflow	`mov %eax, (%ebx)` `ret`	4096

FullStoreOverflow	`pop %eax`
	`pop %ebx`
	`ret`
	`mov %eax, (%ebx)`
	`ret`
		4096

Table 5.4: Gadgets added to the code reuse test programs. #Addrs denotes the number of unrandomized, executable addresses.

Name	Goal condition as BAP expression
LoadConst	`R_EAX_32:u32 == 0x42424242:u32`
StoreMem	`0x42424242:u32 == mem_32:u32?u8[0x8049500:u32, e_little]:u32`
BigStoreMem	`0x4242424242424242:u64 == mem_32:u32?u8[0x8049500:u32, e_little]:u64`

Table 5.5: Goal states.

- By default, Mayhem only explores up to 256 targets of an indirect jump. An option was added that resolves all feasible targets.

There are some differences between how Mayhem explores the symbolic search space and the ideal conditions assumed in the analysis of the previous section. One difference is that Mayhem prunes unreachable states, allowing it to effectively shrink the search space. Mayhem also uses *memory concretization*, which means that sometimes a write (or load) to (or from) a symbolic memory address is replaced by a concrete address that is represented by the symbolic address. This means that some states will not be explored because they were effectively pruned by these artificial concretization constraints. Unfortunately, memory concretization is a central design point of Mayhem and other similar systems to ensure good performance.

Mayhem was used to explore the state space of several small binary programs that each reads 200 bytes from a file to a 10 byte buffer and then return. Because the vulnerability is so trivial to trigger, Mayhem spends its time searching the state space after the vulnerability for the goal state, which is the objective of the code reuse attack. A small program was also chosen because experiments with larger programs did not complete in a reasonable amount of time, and thus revealed little useful information.

Figure 5.10 lists the source code of the base program that was used to create each test program. Additional programs were created by linking this program to object files containing artificial gadgets, which are described in Table 5.4. Each program was tested with the goal states listed in Table 5.5.

Each program and goal combination was tested with both Q and Mayhem on an Intel i7 920 CPU with eight hardware threads. Each experiment instance was given a maximum of 8 h to run. Mayhem runs

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int f() {
  char foo[10];

  int fd = open("symb", O_RDONLY);
  read(fd, foo, 200);
}

int main(int argc, char *argv[]) {

  f();

  return 0;
}
```

Figure 5.10: Code reuse test program.

were limited to explore a configurable number of symbolic indirect jumps. This number was initially set to one, and was increased until (1) Mayhem found an exploit, or (2) the experiment timed out. In each experiment instance, the time required to discover the first payload and whether a payload was found was recorded for both Q and Mayhem. The number of paths explored and the highest number of symbolic indirect jumps tried was additionally recorded for Mayhem runs. Table 5.6 reports the results of the code reuse experiments.

The experiment results show that using gadget abstractions to find ROP payloads is significantly faster than searching for code reuse payloads without abstractions, with Q completing in seconds and Mayhem completing in minutes or hours. The most time consuming stage of Q is searching for gadgets. Q searches each executable address in the binary for gadgets, which is $O(e)$ where e is the number of unrandomized executable program locations. The later analysis phases in Q use the recovered gadgets, and are relatively simple and fast as a result of using the gadget abstractions. In contrast, the expensive stage in Mayhem is searching a tree of execution states. As described in Section 5.2.1, this tree is $O(b^j)$ where j is the number of symbolic indirect jumps explored.

This exponential explosion in the code reuse search space can be observed in the experimental results. Mayhem was able to find payloads for the LoadConst goal states in every program with $j = 1$. But the other goals required at least two indirect jumps. For instance, one solution to achieve the StoreMem goal state with the FullStoreOverflow program would be to direct control to the first gadget to load values into `%eax` and `%ebx`, and then to redirect control to the memory write gadget; this requires two indirect jumps. Unfortunately,

Program	Payload	Mayhem				Q	
		Pay	Paths	Jumps	Time	Pay	Time
NullOverflow	LoadConst	✓	904	1	10 m		4.16 s
NullOverflow	StoreMem		50,224	2	>8 h		5.59 s
NullOverflow	BigStoreMem		50,224	2	>8 h		5.57 s
LoadconstOverflow	LoadConst	✓	904	1	10 m	✓	3.82 s
LoadconstOverflow	StoreMem		50,224	2	>8 h		5.40 s
LoadconstOverflow	BigStoreMem		50,224	2	>8 h		5.49 s
StoreOverflow	LoadConst	✓	902	1	15 m		4.44 s
StoreOverflow	StoreMem		50,224	2	>8 h		5.90 s
StoreOverflow	BigStoreMem		50,224	2	>8 h		5.87 s
FullStoreOverflow	LoadConst	✓	904	1	15 m	✓	4.24 s
FullStoreOverflow	StoreMem		50,224	2	>8 h	✓	4.25 s
FullStoreOverflow	BigStoreMem		50,224	2	>8 h	✓	4.26 s

Table 5.6: Experimental comparison between ROP (Q) and generalized code reuse (Mayhem). Pay indicates whether a payload was successfully produced.

Mayhem timed out after running for 8 h and exploring over 50,000 paths. These results suggest that code reuse is currently only feasible for few indirect jumps.

A common theme in this dissertation is that abstraction has advantages and disadvantages, and this applies to these results as well. Although employing gadgets greatly speeds up analysis times, it excludes some valid code reuse payloads. For instance, Mayhem was able to find LoadConst payloads even for the NullOverflow program, but Q could not. The advantage of not employing abstraction recovery is that Mayhem can find additional payloads. In fact, manual analysis revealed some of the Mayhem payloads are significantly more sophisticated than the payloads Q generates; some Mayhem payloads executed "gadgets" that were hundreds of instructions long. We also witnessed Mayhem using jump-oriented programming (JOP) [30], which is similar to ROP but uses indirect jumps to registers instead of `ret` instructions. Mayhem was unable to produce StoreMem payloads for the FullStoreOverflow program during the 8 h time limit, but we expect that if Mayhem ran with $j = 2$ until completion, it would be able to find a payload, since there is a simple exploit using two jumps. Thus, searching for code reuse attacks via Mayhem has utility in its own right, but the poor scalability is a serious limitation in practice.

Chapter 6

Conclusions

This dissertation supports the thesis that abstraction recovery is possible, and can perform better than low-level analyses. Abstraction recovery is a new approach to building static binary analyses, which allows users to analyze software properties without the assistance of the programmer and source code. One of the major challenges preventing the wide-scale use of static binary analysis has been scalability. Abstraction recovery improves scalability by recovering and then analyzing an abstract representation of the program, effectively abstracting away the irrelevant implementation details of a binary that can slow down analysis.

The claim that abstraction recovery is possible is supported by empirical examples, in the form of two real-world binary analysis systems that utilize abstraction recovery. The first system is a decompiler that recovers C abstractions from x86 binaries produced by a C compiler (Chapter 2). Researchers have studied decompilation extensively, but with an emphasis on reverse engineering applications, in which decompilers emit pseudocode that human reverse engineers can read. Unfortunately, the emphasis on reverse engineering has largely demoted correctness to secondary importance, and few decompiler studies evaluate correctness at all. In the context of abstraction recovery, the correctness of recovered abstractions is important, since recovering incorrect abstractions could lead the analysis to make mistakes. This dissertation rigorously evaluates our decompiler and others on correctness and other properties. While our decompiler performed the best, there remains ample room for advances in decompilation, particularly for abstraction recovery and other applications that require decompilation to be correct.

The second example of abstraction recovery is an automatic system for creating return-oriented programming (ROP) attacks (Chapter 4). ROP attacks bypass the DEP defense through the combination of gadgets, which are small code fragments that perform useful actions and can be chained together. Gadgets were first proposed to help humans manually craft attacks, but they can also be thought of as abstractions that describe the types of computations an attacker can induce. The automatic ROP system builds on this idea

by recovering these gadget abstractions and then using them to reason about whether an attacker can complete a larger objective such as spawning a shell. ROP attacks are generally mitigated by the ASLR defense, which randomizes the addresses of code. However, some ASLR implementations allow small amounts of code to be unrandomized, and it was previously unknown whether these small amounts of code were sufficient to enable attacks. The automated ROP system was able to automatically use these code fragments to create attacks for most programs on a typical modern Linux desktop, demonstrating that allowing even small amounts of unrandomized code to reside in memory, as Linux does, can undermine the DEP defense.

The second major claim of this thesis is that the process of recovering abstractions and analyzing them can be faster than simply analyzing the binary directly. Chapter 5 evaluates this claim by implementing several analyses that use abstraction recovery and comparing their scalability to analyses that do not employ abstraction. To evaluate the scalability of abstraction recovery using source abstractions, several analyses based on verification conditions (VCs) were constructed. VCs are a building block for analyses that can be applied to both abstract and low-level program representations, which makes them ideal for evaluating the performance improvements that abstractions provide. These VC experiments demonstrate that, when checking correctness properties and for the presence of buffer overflow vulnerabilities, abstraction recovery is faster, more scalable, and consumes less resources than low-level analysis. A similar experiment compared a ROP attack based on gadget abstractions to a similar code reuse attack which does not employ abstractions. These experiments showed that the abstraction-based ROP attack is overwhelmingly faster and more scalable. However, the scalability of abstraction does come with a price: the code reuse system was able to find attacks that do not use gadgets, and thus the ROP system could not find.

The results in this dissertation suggest a promising new direction for new static binary analyses. In short, abstraction recovery is a alternative approach to binary analysis, and this dissertation has demonstrated it is possible and more scalable than traditional approaches for several applications. Thus, abstraction recovery should be considered as an option when designing future binary analyses. Beyond the performance advantages explored in this dissertation, there are practical advantages as well. Many binary analyses have common sub-tasks that could be posed as abstraction recovery; if abstraction recovery becomes popular enough, it may be possible to reuse and share such abstraction recovery processes. We envision that in the future, each binary analysis will employ one of many high-quality abstraction recovery algorithms as a pre-processing step, so that binary analysis bears a much greater resemblance to modern source code analysis.

Despite these promising results, there is no reason to assume that abstraction recovery is appropriate for all applications. Abstraction recovery, and the use of abstraction in general, is limited by the identification of appropriate abstractions. Some binary analysis problems may not lend themselves to obvious abstractions, and such problems do not benefit. Likewise, some abstractions are irreversibly lost during compilation,

such as variable names. Abstraction recovery often involves making reasonable assumptions about the binaries being analyzed based on the analysis context. For instance, a decompiler may assume that its input is produced by a benign compiler. However, such an analysis is not safe to use on malware, since the adversary could intentionally violate these assumptions to invalidate the results.

6.1 Future Work

Abstraction recovery opens the door to several new directions of research, the first being the development of additional abstractions. This dissertation explored the recovery of C abstractions, but there are many other languages used in practice, from C++ to more esoteric languages such as Haskell. Clearly, a C decompiler would struggle to recover abstractions from a Haskell program. However, many compilers use similar implementation techniques; for example, many languages have functions and employ the standard `call` and `ret` instructions. An important open question is whether there are more general abstractions that can be recovered from many types of compiled languages.

Most existing work in decompilation emphasizes usability for human reverse engineers, often at the expense of correctness. However, to be used for abstraction recovery, it is critical for decompilers to recover abstractions correctly, since recovering incorrect abstractions could break an otherwise correct analysis. Unfortunately, the experiments from Chapter 2 demonstrate that decompilers have room for improvement with regard to correctness. Even the decompiler developed for this dissertation, which was the most correct of the tested decompilers, could only decompile half of the tested programs correctly. Hopefully, if developers of binary analyses begin to utilize abstraction recovery and decompilation, the demand for more correct decompilers will increase.

This dissertation explores gadgets as a general abstraction, since they can be recovered from any type of binary. General abstractions correspond most closely to traditional binary analysis, which often makes few assumptions about input programs. It is important to develop abstractions for these types of applications as well. Unfortunately, it is challenging to identify abstractions that are both useful and can apply to any binary. It may be important for such abstractions to be opportunistic: if a certain pattern of abstraction is observed, that artifact can be represented abstractly, but with the ability to resort to a less abstract representation if necessary.

6.2 Conclusion

Abstraction recovery is possible, and can help scale some analyses of binary executables. Abstraction recovery is a tool, and like most tools, it does not solve every problem, but can be incredibly effective on the right

problem. This dissertation has only explored a few such problems. However, abstraction is a persistent theme in computer science, and there are likely other static binary analysis problems waiting to be solved by recovering the right abstraction.

Part III

Appendices

Appendix A

Abstraction Recovery Proofs

This chapter contains the proofs for theorems in Chapter 1.

A.1 Proof of Theorem 1.1

Lemma A.1. Given $\mathbb{A}$, $\mathbb{C}$, $\mathbb{O}$, and γ, a solution to the abstraction recovery problem $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ exists for all $\llbracket - \rrbracket$ if γ is injective:

$$\forall a_1, a_2 \in \mathbb{A}.\, a_1 \neq a_2 \implies \gamma(a_1) \cap \gamma(a_2) = \varnothing.$$

Proof. Proof by construction. Let $\alpha : \mathbb{C} \to \mathbb{A}$ be the reverse mapping of γ, such that

$$\alpha(x) = (y \mid \gamma(y) = x)\,. \tag{A.1}$$

α is a well-defined function from $\mathbb{C}$ to $\mathbb{A}$ because there is exactly one mapping for each element in the image of γ, by the definition of an injective function, and the assumption that γ is surjective. Then $(\mathbb{A}, \alpha, \llbracket - \rrbracket)$ is a solution since

$$\begin{aligned} \forall a \in \mathbb{A}. \forall c \in \gamma(a).\, \llbracket a \rrbracket &= \llfloor \alpha(c) \rrfloor \\ &= \llbracket y \mid \gamma(y) = c \rrbracket \\ &= \llbracket a \rrbracket \end{aligned} \tag{A.2}$$

by definitions. □

Lemma A.2. Given $\mathbb{A}$, $\mathbb{C}$, $\mathbb{O}$, and γ, a solution to the abstraction recovery problem $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ does not exist for all $\llbracket - \rrbracket$ if γ is not injective:

$$\neg \forall a_1, a_2 \in \mathbb{A}.\, a_1 \neq a_2 \implies \gamma(a_1) \cap \gamma(a_2) = \varnothing.$$

Proof. Proof by contradiction. Assume for the purposes of contradiction that $\langle \mathbb{R}, \alpha, \llbracket - \rrbracket \rangle$ is a correct solution. Because γ is not injective, there exists some $a_1, a_2 \in \mathbb{A}$ such that $\gamma(a_1) \cap \gamma(a_2) \neq \varnothing$. Let $c \in \gamma(a_1) \cap \gamma(a_2)$, and let $\llbracket - \rrbracket := \lambda a.(a = a_1)$. Then

$$\llbracket a_1 \rrbracket = \llbracket \alpha(c) \rrbracket \qquad \llbracket a_2 \rrbracket = \llbracket \alpha(c) \rrbracket \tag{A.3}$$

by the definition of a correct recovery analysis. In addition,

$$\llbracket a_1 \rrbracket = \mathsf{true} \qquad \llbracket a_2 \rrbracket = \mathsf{false} \tag{A.4}$$

by the definition of $\llbracket - \rrbracket$. Thus, $\mathsf{true} = \mathsf{false}$ by transitivity and Equations A.3 and A.4, which is a contradiction. □

Proof of Theorem 1.1. By Lemmas A.1 and A.2. □

A.2 Proof of Theorem 1.2

Lemma A.3. Let $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ be an instance of the abstraction recovery problem, and let $\mathbb{A}/\sim$ be the equivalence class defined by $\sim$ where $x \sim y := \llbracket x \rrbracket = \llbracket y \rrbracket$. Then, a solution to the abstraction recovery problem $\langle \mathbb{R}, \alpha, \llbracket - \rrbracket \rangle$ exists if each concrete program only implements abstract programs from one equivalence class:

$$\forall c \in \mathbb{C}.\, \exists! E \in \mathbb{A}/\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E.$$

Proof. Proof by construction. Let $g : \mathbb{C} \to \mathbb{A}/\sim$ be the function that maps each concrete program to the unique equivalence class that maps to it:

$$g(x) = \{E \in \mathbb{A}/\sim\, : \forall a \in \mathbb{A}.\, x \in \gamma(a) \implies a \in E\}, \tag{A.5}$$

and $s : \mathbb{A}/\sim \to \mathbb{A}$ be a representative element for each equivalence class. A unique mapping in g is guaranteed to exist by the hypothesis. Then $(\mathbb{A}, s(g(c)), \llbracket - \rrbracket)$ is a correct solution, because

$$\begin{aligned} \forall a \in \mathbb{A}.\, \forall c \in \gamma(a).\, \llbracket a \rrbracket &= \llbracket s(g(c)) \rrbracket \\ &= \llbracket s(E \in \mathbb{A}/\sim\, : \forall a' \in A.\, c \in \gamma(a') \implies a' \in E) \rrbracket \\ &= \llbracket a \rrbracket \end{aligned} \tag{A.6}$$

by definitions. □

Lemma A.4. Let $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ be an instance of the abstraction recovery problem, and let $\mathbb{A}/\!\sim$ be the equivalence class defined by $\sim$ where $x \sim y := \llbracket x \rrbracket = \llbracket y \rrbracket$. Then, a solution to the abstraction recovery problem $\langle \mathbb{R}, \alpha, \llparenthesis - \rrparenthesis \rangle$ does not exist if each concrete program does not implement abstract programs from exactly one equivalence class:

$$\neg \forall c \in \mathbb{C}.\, \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E.$$

Proof. Proof by contradiction. Assume for the purposes of contradiction that there exists a correct solution $\langle \mathbb{R}, \alpha, \llparenthesis - \rrparenthesis \rangle$. By the hypothesis, there is not a unique equivalence class for each concrete program:

$$\begin{aligned} &\neg \forall c \in \mathbb{C}.\, \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E. \\ =\; &\exists c.\, \neg \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E. \end{aligned} \tag{A.7}$$

There must be at least one equivalence class for each concrete program because γ is surjective, thus there are at least two equivalence classes that map to c. Let $[a_1]$ and $[a_2]$ represent two such classes.

By the definition of correctness, $\llbracket a_1 \rrbracket = \llparenthesis \alpha(c) \rrparenthesis = \llbracket a_2 \rrbracket$. However, by definition of $\sim$, $\llbracket a_1 \rrbracket \neq \llbracket a_2 \rrbracket$, which is a contradiction. □

Proof of Theorem 1.2. By Lemmas A.3 and A.4. □

A.3 Proof of Theorem 1.3

Lemma A.5. Given $\mathbb{A}, \mathbb{C}, \mathbb{O}$, $\llbracket - \rrbracket$ and $[\![-]\!]$, a fully abstract semantics of $\mathbb{A}$, a solution to the abstraction recovery problem $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llbracket - \rrbracket \rangle$ exists for all γ that respect the semantics of $[\![-]\!]$ if $\llbracket - \rrbracket$ computes an observable property with respect to $[\![-]\!]$.

Proof. Let $\mathbb{A}/\!\sim$ be the equivalence class defined by $\sim$ where $x \sim y := \llbracket x \rrbracket = \llbracket y \rrbracket$, let $\llbracket - \rrbracket$ be an arbitrary analysis that computes an observable property with respect to $[\![-]\!]$, and let γ be an arbitrary concretization function that respects the semantics of $[\![-]\!]$. Then

$$\forall c \in \mathbb{C}.\, \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E. \tag{A.8}$$

holds by the following. Assume that it did not for the purposes of contradiction. Then

$$\begin{aligned} &\neg \forall c \in \mathbb{C}.\, \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E \\ =\; &\exists c \in \mathbb{C}.\, \neg \exists! E \in \mathbb{A}/\!\sim.\, \forall a \in \mathbb{A}.\, c \in \gamma(a) \implies a \in E \end{aligned} \tag{A.9}$$

There must be at least one equivalence class for each concrete program because γ is surjective, thus there are at least two equivalence classes that map to c. Let $[a_1]$ and $[a_2]$ represent two such classes.

$$\llceil a_1 \rrceil \neq \llceil a_2 \rrceil \tag{A.10}$$

by the definition of $\sim$, and

$$\llbracket a_1 \rrbracket = \llbracket a_2 \rrbracket \implies \llceil a_1 \rrceil = \llceil a_2 \rrceil \tag{A.11}$$

by the definition of an observable property. Thus, by Equations A.10 and A.11 and the contrapositive,

$$\llbracket a_1 \rrbracket \neq \llbracket a_2 \rrbracket . \tag{A.12}$$

Because γ respects the semantics of $\llbracket - \rrbracket$,

$$\forall a_1, a_2 \in \mathbb{A}.\ \llbracket a_1 \rrbracket \neq \llbracket a_2 \rrbracket \implies \gamma(a_1) \cap \gamma(a_2) = \varnothing . \tag{A.13}$$

But $c \in \gamma(a_1) \cap \gamma(a_2)$, and thus there is a contradiction by Equations A.12 and A.13.

Finally, by Theorem 1.2 and Equation A.8, there exists a solution for $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llceil - \rrceil \rangle$.

□

Lemma A.6. Given $\mathbb{A}, \mathbb{C}, \mathbb{O}$, $\llceil - \rrceil$ and $\llbracket - \rrbracket$, a fully abstract semantics of $\mathbb{A}$, a solution to the abstraction recovery problem $\langle \mathbb{A}, \mathbb{C}, \mathbb{O}, \gamma, \llceil - \rrceil \rangle$ does not exist for all γ that respect the semantics of $\llbracket - \rrbracket$ if $\llceil - \rrceil$ does not compute an observable property with respect to $\llbracket - \rrbracket$.

Proof. Proof by construction. By the hypothesis, $\llceil - \rrceil$ does not compute an observable property with respect to $\llbracket - \rrbracket$, and thus

$$\begin{aligned} &\neg \forall a_1, a_2 \in \mathbb{A}.\ \llbracket a_1 \rrbracket = \llbracket a_2 \rrbracket \implies \llceil a_1 \rrceil = \llceil a_2 \rrceil \\ &= \exists a_1, a_2 \in \mathbb{A}.\ \llbracket a_1 \rrbracket = \llbracket a_2 \rrbracket \wedge \llceil a_1 \rrceil \neq \llceil a_2 \rrceil \end{aligned} \tag{A.14}$$

by the definition of an observable property.

Let γ be an arbitrary concretization function that respects the semantics of $\llbracket - \rrbracket$. (Note that if such a γ does not exist, then the lemma is trivially true.) Let c_1 be an arbitrary program that implements a_1 in γ, and c_2 be an arbitrary program that implements a_2 in γ. These are guaranteed to exist because γ is surjective.

- If $c_1 = c_2$ then no solution exists because a correct solution must have both

$$\llceil a_1 \rrceil = \llfloor \alpha(c_1) \rrfloor \qquad \llceil a_2 \rrceil = \llfloor \alpha(c_2) \rrfloor = \llfloor \alpha(c_1) \rrfloor \tag{A.15}$$

but $\llceil a_1 \rrceil \neq \llceil a_2 \rrceil$ by Equation A.14.

- If $c_1 \neq c_2$ then let γ' be γ with c_2 as an additional implementation of a_1. γ' still respects the semantics of $[\![-]\!]$ because $[\![a_1]\!] = [\![a_2]\!]$ by Equation A.14. Finally, no solution exists because a correct solution must have both

$$\lceil\!\lceil a_1 \rceil\!\rceil = \lfloor\!\lfloor \alpha(c_1) \rfloor\!\rfloor \qquad \lceil\!\lceil a_2 \rceil\!\rceil = \lfloor\!\lfloor \alpha(c_2) \rfloor\!\rfloor = \lfloor\!\lfloor \alpha(c_1) \rfloor\!\rfloor \tag{A.16}$$

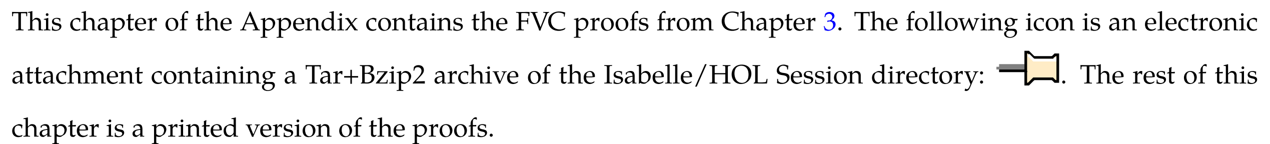
but $\lceil\!\lceil a_1 \rceil\!\rceil \neq \lceil\!\lceil a_2 \rceil\!\rceil$ by Equation A.14.

□

Proof of Theorem 1.3. By Lemmas A.5 and A.6. □

Appendix B

Forward Verification Conditions Isabelle/HOL Proofs

This chapter of the Appendix contains the FVC proofs from Chapter 3. The following icon is an electronic attachment containing a Tar+Bzip2 archive of the Isabelle/HOL Session directory: . The rest of this chapter is a printed version of the proofs.

B.1 Arithmetic and Boolean Expressions

```
theory AExp imports Main begin
```

B.1.1 Arithmetic Expressions

```
type-synonym vname = string
type-synonym val = int
type-synonym state = vname ⇒ valvalue aval (Plus (V ''x'') (N 5)) (λx. if x = ''x'' then 7 else 0)
```

The same state more concisely:

```
value aval (Plus (V ''x'') (N 5)) ((λx. 0) (''x'':= 7))
```

A little syntax magic to write larger states compactly:

```
definition null-state (<>) where
 null-state ≡ λx. 0
syntax
 -State :: updbinds => 'a (<->)
translations
 -State ms => -Update <> ms
```

We can now write a series of updates to the function $\lambda x.\ 0$ compactly:

```
lemma <a := Suc 0, b := 2> = (<> (a := Suc 0)) (b := 2)
 by (rule refl)

value aval (Plus (V ''x'') (N 5)) <''x'' := 7>
```

Variables that are not mentioned in the state are 0 by default in the <>($a := b$) syntax:

value *aval* (*Plus* (*V ''x''*) (*N 5*)) <*''y'' := 7*>

Note that this <...> syntax works for any function space $\tau_1 \Rightarrow \tau_2$ where τ_2 has a *0*.

fun *aval-size* :: *aexp* ⇒ *nat* **where**
 aval-size (*N -*) = *1*
| *aval-size* (*V -*) = *1*
| *aval-size* (*Plus* e_1 e_2) = (*aval-size* e_1) + (*aval-size* e_2) + *1*

The size of *aexp* expressions.

B.1.2 Constant Folding

Evaluate constant subsexpressions:

theorem *aval-asimp-const*:
 aval (*asimp-const a*) *s* = *aval a s*
apply(*induction a*)
apply (*auto split*: *aexp.split*)
done

Now we also eliminate all occurrences 0 in additions. The standard method: optimized versions of the constructors:

lemma *aval-plus*[*simp*]:
 aval (*plus a1 a2*) *s* = *aval a1 s* + *aval a2 s*
apply(*induction a1 a2 rule*: *plus.induct*)
apply *simp-all*
done

Note that in *asimp-const* the optimized constructor was inlined. Making it a separate function *AExp.plus* improves modularity of the code and the proofs.

value *asimp* (*Plus* (*Plus* (*N 0*) (*N 0*)) (*Plus* (*V ''x''*) (*N 0*)))

theorem *aval-asimp*[*simp*]:
 aval (*asimp a*) *s* = *aval a s*
apply(*induction a*)
apply *simp-all*
done

end

theory *BExp* **imports** *AExp* **begin**

B.1.3 Boolean Expressions

type-synonym *bstate* = *vname* ⇒ *bool*

type-synonym *state* = *AExp.state* ∗ *bstate*

definition *null-bstate* **where** *null-bstate* ≡ (λ*x*. *True*)
definition *null-state* **where** *null-state* ≡ (<>, *null-bstate*)**value** *bval* (*Less* (*V ''x''*) (*Plus* (*N 3*) (*V ''y''*)))
 (<*''x''* := 3, *''y''* := *1*>, *null-bstate*)

value *bval* (*Equal* (*V ''x''*) (*V ''x''*)) (<*''x''* := 3>, *null-bstate*)

For now, we define the size of *aexp* to always be one. This is because our proofs never manipulate expressions of type *aexp*, so their size can be left constant.

fun *bval-size* :: *bexp* ⇒ *nat* **where**

bval-size (*Bc* -) = *1*
| *bval-size* (*BV* -) = *1*
| *bval-size* (*Not e*) = (*bval-size e*) + *1*
| *bval-size* (*And* b_1 b_2) = (*bval-size* b_1) + (*bval-size* b_2) + *1*
| *bval-size* (*Or* b_1 b_2) = (*bval-size* b_1) + (*bval-size* b_2) + *1*
| *bval-size* (*Implies* b_1 b_2) = (*bval-size* b_1) + (*bval-size* b_2) + *1*
| *bval-size* (*BEqual* b_1 b_2) = (*bval-size* b_1) + (*bval-size* b_2) + *1*
| *bval-size* (*Less* b_1 b_2) = (*aval-size* b_1) + (*aval-size* b_2) + *1*
| *bval-size* (*Equal* b_1 b_2) = (*aval-size* b_1) + (*aval-size* b_2) + *1*

The size of *bexp* expressions.

To improve automation:

lemma *bval-And-if* [*simp*]:
bval (*And b1 b2*) *s* = (*if bval b1 s then bval b2 s else False*)
by(*simp*)

lemma *bval-Or-if* [*simp*]:
bval (*Or b1 b2*) *s* = (*if bval b1 s then True else bval b2 s*)
by(*simp*)

lemma *bval-Implies-if* [*simp*]:
bval (*Implies b1 b2*) *s* = (*if bval b1 s then bval b2 s else True*)
by(*simp*)

declare *bval.simps*(*4*)[*simp del*] — remove the original eqn
declare *bval.simps*(*5*)[*simp del*] — remove the original eqn
declare *bval.simps*(*6*)[*simp del*] — remove the original eqn

B.1.4 Constant Folding

Optimizing constructors:

lemma [*simp*]: *bval* (*less a1 a2*) *s* = (*aval a1* (*fst s*) < *aval a2* (*fst s*))
apply(*induction a1 a2 rule*: *less.induct*)
apply *simp-all*
donelemma *bval-and*[*simp*]: *bval* (*and b1 b2*) *s* = (*bval b1 s* ∧ *bval b2 s*)
apply(*induction b1 b2 rule*: *and.induct*)
apply *simp-all*
donelemma *bval-or*[*simp*]: *bval* (*or b1 b2*) *s* = (*bval b1 s* ∨ *bval b2 s*)
apply(*induction b1 b2 rule*: *or.induct*)
apply *simp-all*
donelemma *bval-implies*[*simp*]: *bval* (*implies b1 b2*) *s* = (*bval b1 s* ⟶ *bval b2 s*)
apply(*induction b1 b2 rule*: *implies.induct*)
apply *simp-all*
donelemma *bval-bequal*[*simp*]: *bval* (*bequal b1 b2*) *s* = (*bval b1 s* = *bval b2 s*)
apply(*induction b1 b2 rule*: *bequal.induct*)
apply *simp-all*
donelemma *bval-not*[*simp*]: *bval* (*not b*) *s* = (¬ *bval b s*)
apply(*induction b rule*: *not.induct*)
apply *simp-all*
donelemma *bval-equal*[*simp*]: *bval* (*equal a1 a2*) *s* = (*aval a1* (*fst s*) = *aval a2* (*fst s*))
apply(*induction a1 a2 rule*: *equal.induct*)
apply *simp-all*
done

Now the overall optimizer:

value *bsimp* (*And* (*Less* (*N 0*) (*N 1*)) *b*)

value *bsimp* (*And* (*Less* (*N 1*) (*N 0*)) (*B True*))

theorem *bval* (*bsimp b*) *s* = *bval b s*
apply(*induction b*)
apply *simp-all*
done

theorem *bexp-and-comm*: *bval* (*And x y*) *s* = *bval* (*And y x*) *s*
apply *auto*
done

end

theory *FVC*
imports *Main* ~~/*src*/*HOL*/*Library*/*LaTeXsugar AExp BExp*
begin

B.2 Forward Verification Conditions

In this section we verify our proofs are correct via Isabelle, a widely-used proof assistant. In Isabelle, we assume all programs have been passified, thus we do not need to use substitution rule.

B.2.1 Basic Definitions

type-synonym *predicate* = *bexp*

Predicates are modeled as boolean expressions, and have type *bexp*.

definition *true-predicate* :: *predicate* **where**
true-predicate = *Bc True*

fun *inverses* :: *bexp* ⇒ *bexp* ⇒ *bool* **where**
inverses e (*Not e'*) = (*e* = *e'*)
| *inverses* (*Not e'*) *e* = (*e* = *e'*)
| *inverses* - - = *False*

Inverse conditions.

datatype *GCLS* =
Assert bexp
| *Assume bexp*
| *Choice GCLS GCLS*
| *Seq GCLS GCLS*

The datatype representing programs written in GCL. There is no statement type for assignments, because we assume that assignments have already been converted to assertions or assumptions during passification.

fun *gclsize* :: *GCLS* ⇒ *nat* **where**
gclsize (*Assert e*) = (*bval-size e*) + *1*
| *gclsize* (*Assume e*) = (*bval-size e*) + *1*

```
| gclsize (Choice s1 s2) = (gclsize s1) + (gclsize s2) + 1
| gclsize (Seq s1 s2) = (gclsize s1) + (gclsize s2) + 1
```

The size of GCL statements.

```
fun gclwp :: GCLS ⇒ predicate ⇒ predicate
where
 gclwp (Assert e) b = And e b
| gclwp (Assume e) b = Implies e b
| gclwp (Choice s1 s2) b = And (gclwp s1 b) (gclwp s2 b)
| gclwp (Seq s1 s2) b = gclwp s1 (gclwp s2 b)
```

The traditional weakest precondition algorithm introduced by Dijkstra. Informally, *gclwp P Q* is true when the program *P* terminates normally in a state satisfying *Q*.

```
fun gclwlp :: GCLS ⇒ predicate ⇒ predicate
where
 gclwlp (Assert e) b = Implies e b
| gclwlp (Assume e) b = Implies e b
| gclwlp (Choice s1 s2) b = And (gclwlp s1 b) (gclwlp s2 b)
| gclwlp (Seq s1 s2) b = gclwlp s1 (gclwlp s2 b)
```

The weakest liberal precondition algorithm. Informally, *gclwlp P Q* is true when the program *P* terminates normally in a state satisfying *Q*, OR when it terminates abnormally.

```
abbreviation wptrue :: GCLS ⇒ predicate where
 wptrue s ≡ gclwp s (Bc True)

abbreviation wlpfalse :: GCLS ⇒ predicate where
 wlpfalse s ≡ gclwlp s (Bc False)


fun fvc-ms :: GCLS ⇒ predicate
 and fvc-mf :: GCLS ⇒ predicate
 where
 fvc-ms (Assert e) = Bc True
| fvc-ms (Assume e) = e
| fvc-ms (Choice s1 s2) = Or (fvc-ms s1) (fvc-ms s2)
| fvc-ms (Seq s1 s2) = And (fvc-ms s1) (Or (fvc-mf s1) (fvc-ms s2))
| fvc-mf (Assert e) = Not e
| fvc-mf (Assume e) = Bc False
| fvc-mf (Choice s1 s2) = Or (fvc-mf s1) (fvc-mf s2)
| fvc-mf (Seq s1 s2) = And (fvc-ms s1) (Or (fvc-mf s1) (fvc-mf s2))
```

FVC is implemented by computing two predicates. The may start predicate (MS) reflects whether a program execution may start (by satisfying assumptions for at least one set of non-deterministic choices). The may fail predicate (MF) reflects whether the program may fail an assertion.

```
definition fvc :: GCLS ⇒ predicate ⇒ predicate where
 fvc s b = (Implies (fvc-ms s) (And (Not (fvc-mf s)) b))
```

FVC produces the final verification condition by combining the predicates of *fvc-ms* and *fvc-mf*.

```
lemma leino-wlp: bval (gclwlp s b) st = (bval (Or (wlpfalse s) b) st)
 proof (induct s arbitrary: b)
  case (Choice s1 s2)
  show ?case
   apply (simp only: gclwp.simps gclwlp.simps bval.simps)
   apply (subst Choice, subst Choice, subst Choice, simp)
   done
  case (Seq s1 s2)
  show ?case
```

```
    apply (simp only: gclwp.simps gclwlp.simps bval.simps)
    apply (subst Seq, subst Seq, simp only: bval.simps)
    apply (subst Seq, subst Seq, subst Seq, simp)
    done
qed (simp-all)

lemma leino-wp: bval (gclwp s b) st = bval (And (wptrue s) (Or (wlpfalse s) b)) st
  proof (induct s arbitrary: b)
   case (Choice s1 s2) show ?case
    apply (simp only: gclwp.simps gclwlp.simps bval.simps)
    apply (subst Choice, subst Choice, subst Choice, subst Choice, simp)
    done
   case (Seq s1 s2) show ?case
    apply (simp only: gclwp.simps gclwlp.simps bval.simps)
     apply (subst Seq, subst Seq, simp only: bval.simps, subst Seq, subst Seq, subst Seq, subst leino-wlp, subst leino-wlp, subst
leino-wlp, simp)
    done
   qed (simp-all)
```

Leino's theorems demonstrate that, for passified programs, a weakest precondition computation with an arbitrary postcondition can be transformed into computations with postconditions *True* and *False*.

B.2.2 Size theorems

First, we demonstrate that FVC produces linear sized formulas for passified programs.

```
record fvcout =
  Vars :: predicate
  MS :: predicate
  MF :: predicate

fun fvc-ms-mf :: GCLS ⇒ fvcout where
  fvc-ms-mf (Assert e) = (| Vars=Bc True, MS=Bc True, MF=Not e |)
| fvc-ms-mf (Assume e) = (| Vars=Bc True, MS=e, MF=Bc False |)
| fvc-ms-mf (Choice s1 s2) =
  (let fvc1 = fvc-ms-mf s1 in
   let fvc2 = fvc-ms-mf s2 in
   (| Vars=Bc True, MS=Or (MS fvc1) (MS fvc2), MF=Or (MF fvc1) (MF fvc2) |))
| fvc-ms-mf (Seq s1 s2) =
  (let fvc1 = fvc-ms-mf s1 in
   let fvc2 = fvc-ms-mf s2 in
   (| Vars=Bc True, MS=And (MS fvc1) (Or (MF fvc1) (MS fvc2)), MF=And (MS fvc1) (Or (MF fvc1) (MF fvc2))|))
```

An alternate formulation of *fvc-ms* and *fvc-mf* as a single function that returns a tuple. This will be useful for the size proofs.

```
lemma fvc-ms-mf-equiv: MS (fvc-ms-mf S) = fvc-ms S ∧ MF (fvc-ms-mf S) = fvc-mf S
proof (induct S)
case (Assert e) show ?case by simp
case (Assume e) show ?case by simp
case (Choice s1 s2) from Choice show ?case
 apply (simp add: Let-def)
done
case (Seq s1 s2) from Seq show ?case
 apply (simp add: Let-def)
done
qed
```

The alternate formulation is equivalent to the original.

```
fun fvc-ms-mf-small :: GCLS ⇒ fvcout where
```

```
 fvc-ms-mf-small (Assert e) = (| Vars=Bc True, MS=Bc True, MF=Not e |)
| fvc-ms-mf-small (Assume e) = (| Vars=Bc True, MS=e, MF=Bc False |)
| fvc-ms-mf-small (Choice s1 s2) =
  (let fvc1 = fvc-ms-mf-small s1 in
   let fvc2 = fvc-ms-mf-small s2 in
   (| Vars=And (Vars fvc1) (Vars fvc2), MS=Or (MS fvc1) (MS fvc2), MF=Or (MF fvc1) (MF fvc2) |))
| fvc-ms-mf-small (Seq s1 s2) =
  (let fvc1 = fvc-ms-mf-small s1 in
   let fvc2 = fvc-ms-mf-small s2 in
   (| Vars=And (BEqual (BV ''ms1'') (MS fvc1)) (And (BEqual (BV ''mf1'') (MF fvc1)) (And (Vars fvc1) (Vars fvc2))),
    MS=And (BV ''ms1'') (Or (BV ''mf1'') (BV ''ms2'')),
    MF=And (BV ''ms1'') (Or (BV ''mf1'') (BV ''mf2''))|))
```

Now we write an alternate formulation that is the same as *fvc-ms-mf* except that it de-duplicates values in the *Vars* predicate. We do not show equivalence, but rather show that the same implementation using free variables would produce a linear size VC. The correctness of this technique has already been demonstrated by Flanagan and Saxe.

```
definition fvc-small :: GCLS ⇒ predicate ⇒ predicate where
fvc-small s b = (let fvc = fvc-ms-mf-small s in
            (Implies (Vars fvc) (Implies (MS fvc) (And (Not (MF fvc)) b))))

abbreviation y_1 :: nat where y_1 ≡ 17

lemma fvc-ms-mf-small-size: (let fvc = fvc-ms-mf-small S in
              bval-size (Vars fvc) + bval-size (MS fvc) + bval-size (MF fvc)) <= y_1*gclsize S
proof −
  show (let fvc = fvc-ms-mf-small S in
      bval-size (Vars fvc) + bval-size (MS fvc) + bval-size (MF fvc)) <= y_1*gclsize S
  proof (induct S)
    case (Assert e) show ?case by simp
    next case (Assume e) show ?case by simp
    next case (Choice s1 s2) from Choice show ?case
      apply (simp, case-tac fvc-ms-mf-small s1, simp, case-tac fvc-ms-mf-small s2, simp)
      done
    next case (Seq s1 s2) from Seq show ?case
      apply (case-tac fvc-ms-mf-small s1, simp, case-tac fvc-ms-mf-small s2, simp)
      done
  qed
qed


lemma fvc-ms-mf-small-size2: bval-size (Vars (fvc-ms-mf-small S)) + (bval-size (MS (fvc-ms-mf-small S)) + bval-size (MF
(fvc-ms-mf-small S))) <= y_1*gclsize S
  proof −
    have Let: bval-size (Vars (fvc-ms-mf-small S)) + (bval-size (MS (fvc-ms-mf-small S)) + bval-size (MF (fvc-ms-mf-small S)))
= (let fvc = fvc-ms-mf-small S in bval-size (Vars fvc) + bval-size (MS fvc) + bval-size (MF fvc)) by (simp add: Let-def)
    have Lemma: (let fvc = fvc-ms-mf-small S in
       bval-size (Vars fvc) + bval-size (MS fvc) + bval-size (MF fvc)) <= y_1*gclsize S by (rule fvc-ms-mf-small-size)
    show ?thesis by (subst Let, rule Lemma)
  qed

theorem fvc-small-size: bval-size (fvc-small s b) <= y_1 * (gclsize s) + bval-size b + 4
  apply (simp add: fvc-small-def Let-def)
  apply (rule fvc-ms-mf-small-size2)
done
```

fvc-small produces a linear sized formula for any passified statement given as input.

B.2.3 Equivalence theorems

Next we prove that FVC produces formulas that are equivalent to Dijkstra's WP algorithm.

```
lemma connection: (bval (fvc-ms s) st = bval (Or (Not (wptrue s)) (Not (wlpfalse s))) st) ∧ (bval (fvc-mf s) st = bval (Not
(wptrue s)) st) (is ?Pa s ∧ ?Pb s)
— This lemma connects the MS and MF predicates to wp and wlp.
proof (induct s)
case (Choice s1 s2)
assume H1: ?Pa s1 ∧ ?Pb s1
assume H2: ?Pa s2 ∧ ?Pb s2

from H1 and H2 and fvc-ms-def show ?case by simp

case (Seq s1 s2)
assume H1: ?Pa s1 ∧ ?Pb s1
assume H2: ?Pa s2 ∧ ?Pb s2

have wlpseq: bval (Or (wlpfalse s1) (wlpfalse s2)) st = bval (gclwlp s1 (wlpfalse s2)) st by (subst leino-wlp[where s=s1], simp)
have wpseq: bval (And (wptrue s1) (Or (wlpfalse s1) (wptrue s2))) st = bval (gclwp s1 (wptrue s2)) st by (subst leino-wp[where
s=s1], simp)

from H1 and H2 and wlpseq and wpseq and fvc-ms-def show ?case by auto

qed(simp-all)


theorem fvc-equiv: bval (fvc s b) st = bval (gclwp s b) st
— This theorem demonstrates that running the fvc algorithm is equivalent to running gclwp for all passified GCL pro-
grams.
proof −
 have bval (fvc s b) st = bval (Implies (fvc-ms s) (And (Not (fvc-mf s)) b)) st by (simp add: fvc-def)
 also have ... = bval (Or (Not (fvc-ms s)) (And (Not (fvc-mf s)) b)) st by simp
 also have ... = bval (Or (Not (Or (Not (wptrue s)) (Not (wlpfalse s)))) (And (Not (fvc-mf s)) b)) st by (simp add: connection)
 also have ... = bval (Or (And (wptrue s) (wlpfalse s)) (And (Not (fvc-mf s)) b)) st by simp
 also have ... = bval (Or (And (wptrue s) (wlpfalse s)) (And (Not (Not (wptrue s))) b)) st by (simp add: connection)
 also have ... = bval (Or (And (wptrue s) (wlpfalse s)) (And (wptrue s) b)) st by simp
 also have ... = bval (And (wptrue s) (Or (wlpfalse s) b)) st by simp
 also have ... = bval (gclwp s b) st by (subst leino-wp, simp)
 finally show ?thesis .
 qed


lemma inverses-are-negated: inverses e1 e2 ⟹ (bval e1 st = (¬ bval e2 st))
 apply (induct e1 e2 rule: inverses.induct)
 apply simp-all
 done

lemma inverses-are-negated2: inverses e1 e2 ==> (¬ bval e1 st) −−> bval e2 st
 apply (simp add: inverses-are-negated)
 done

fun total :: GCLS ⇒ bool where
 total (Assert e) = True
| total (Assume e) = False
| total (Choice (Seq (Assume e1) s1) (Seq (Assume e2) s2)) = (total s1 ∧ total s2 ∧ inverses e1 e2)
| total (Choice s1 s2) = (total s1 ∧ total s2)
| total (Seq s1 s2) = (total s1 ∧ total s2)
```

Total programs only use assumptions in if-then-else statements.

```
lemma total-fvcms-true: total s ⟹ (bval (fvc-ms s) st = bval (Bc True) st)
 apply (induct s rule: total.induct)
 apply simp
 apply simp
 apply (simp add: bval.simps fvc-ms.simps inverses-are-negated2)
 apply (rule inverses-are-negated2, auto)
 done
```

fvc-ms is always true for total programs. Intuitively, this is because at least one *Assume* statement in each *Choice* statement is true.

```
definition total-fvc :: GCLS ⇒ predicate ⇒ predicate where
 total-fvc s b = And (Not (fvc-mf s)) b
```

Using *total-fvcms-true* we can now define an optimized version of FVC for total programs.

```
theorem total-fvc-equiv: total s ⟹ bval (total-fvc s b) st = bval (gclwp s b) st
proof –
assume det: total s
have bval (gclwp s b) st = bval (fvc s b) st by (simp add: fvc-equiv)
also have ... = bval (Implies (fvc-ms s) (And (Not (fvc-mf s)) b)) st by (simp add: fvc-def)
also from det have ... = bval (Implies (Bc True) (And (Not (fvc-mf s)) b)) st by (simp add: total-fvcms-true)
also have ... = bval (And (Not (fvc-mf s)) b) st by simp
also have ... = bval (total-fvc s b) st by (simp add: total-fvc-def)
finally show ?thesis by simp
qed
```

For total programs, the result of the optimized algorithm is the same as the classical weakest precondition algorithm.

```
end
```

Appendix C

Benchmark Programs

This chapter contains the source code for the programs in the FVC experiments from Chapters 3 and 5.

C.1 berkeley

berkeley.c source code:

```
/* expected: Valid */
/* post: The error condition specified in the original benchmark did not occur. */

#include "common.h"

int f(int invalid, int unowned, int nonexclusive, int exclusive)
{

  init_choice;

  if ((exclusive==0) && (nonexclusive==0) && (unowned==0) && (invalid>= 1))// no overfow check
  {

    while (make_choice & 1)
    {
      if (make_choice & 1)
      {
        if (invalid >= 1) {
          nonexclusive=nonexclusive+exclusive;
          exclusive=0;
          invalid=invalid-1;
          unowned=unowned+1;
        }
      }
      else
      {
        if (make_choice & 1)
        {
          if (nonexclusive + unowned >=1) {
```

```
            invalid=invalid + unowned + nonexclusive-1;
            exclusive=exclusive+1;
            unowned=0;
            nonexclusive=0;
          }
        }
        else
        {
          if (invalid >= 1) {
            unowned=0;
            nonexclusive=0;
            exclusive=1;
            invalid=invalid+unowned+exclusive+nonexclusive-1;
          }
        }
      }

      if (exclusive < 0 || unowned < 0 || invalid + unowned + exclusive + nonexclusive < 1) // moved in
        return 0; // reachable -> failure -> invalid

    }

    return 1;

  }

  return 1;
}
```

C.2 barber

barber.c source code:

```
/* expected: Valid */
/* post: The error condition specified in the original benchmark did not occur. */
#include "common.h"

int f() {
  unsigned int barber;
  unsigned int chair;
  unsigned int open;
  unsigned int p1;
  unsigned int p2;
  unsigned int p3;
  unsigned int p4;
  unsigned int p5;

  barber=0;
  chair=0;
  open=0;
  p1=0;
  p2=0;
  p3=0;
```

```
p4=0;
p5=0;

init_choice;

while (make_choice) {

  if (make_choice) { /* new customer 1 */
    if (p1 == 0 && barber >= 1) {
      barber = barber - 1;
      chair = chair + 1;
      p1 = 1;
    }
  }
  else if (make_choice) { /* new customer 2 */
    if (p2 == 0 && barber >= 1) {
      barber = barber - 1;
      chair = chair + 1;
      p2 = 1;
    }
  }
  else if (make_choice) { /* service customer 2 */
    if (p2 == 1 && open >= 1) {
      open = open - 1;
      p2 = 0;
    }
  }
  else if (make_choice) { /* new customer 3 */
    if (p3 == 0 && barber >= 1) {
      barber = barber - 1;
      chair = chair + 1;
      p3 = 1;
    }
  }
  else if (make_choice) { /* service customer 3 */
    if (p3 == 1 && open >= 1) {
      open = open - 1;
      p3 = 0;
    }
  }
  else if (make_choice) { /* new customer 4 */
    if (p4 == 0 && barber >= 1) {
      barber = barber - 1;
      chair = chair + 1;
      p4 = 1;
    }
  }
  else if (make_choice) { /* service customer 4 */
    if (p4 == 1 && open >= 1) {
      open = open - 1;
      p4 = 0;
    }
  }
  else if (make_choice) {
```

```
      if (p5 == 0) {
        barber = barber + 1;
        p5 = 1;
      }
    }
    else if (make_choice) {
      if (p5 == 1 && chair >= 1) {
        chair = chair - 1;
        p5 = 2;
      }
    }
    else if (make_choice) {
      if (p5 == 2) {
        open = open + 1;
        p5 = 3;
      }
    }
    else if (make_choice) {
      if (p5 == 3 && open == 0) {
        p5 = 0;
      }
    } else { /* service customer 1 */
      if (p1 == 1 && open >= 1) {
        open = open - 1;
        p1 = 0;
      }
    }
  }
  if (!(p5 >= open) || ! (p1 <= 1) || !(p2 <= 1) || !(p3 <= 1) || !(p4 <= 1) || !(p5 <= 3)
      || !(p4 >= 0) || !(p3 >= 0) || !(p2 >= 0) || !(p1 >= 0) || !(open >= 0) || !(chair >= 0)
      || !(barber >= 0))
    return false;

  return true;
}
```

C.3 binary

binary.c source code:

```
/* expected: Valid */
/* post: The output is the index of the specified element. */

#include "common.h"

int array[5000];

int f(int index, int max) {

  if (index < 0 || max <= 1) return true;
  if (index > max-1) return true;

  for (int i = 0; i < max; i++) {
```

```
    array[i] = i;
  }

  int elem = array[index];

  int imin = 0;
  int imax = max-1;
  int mid;

  while (imin <= imax) {
    mid = (imin + imax + 1) / 2;

    if (array[mid] < elem) {
      imin = mid + 1;
    } else if (array[mid] > elem) {
      imax = mid - 1;
    } else {
      imax = imin - 1;
    }
  }

  return (array[mid] == elem);
}
```

C.4 bubble

bubble.c source code:

```
/* expected: Invalid */
/* post: The output is sorted. */

#include "common.h"

int array[5000];

int f() {
  int len;
  for (len = 0; array[len]; len++) { }

  bool swapped = true;
  while (swapped) {
    swapped = false;

    for (int i = 1; i < len; i++) {
      if (array[i-1] > array[i]) {
        swapped = true;
        int tmp = array[i-1];
        array[i-1] = array[i];
        array[i] = tmp;
      }
    }
  }
```

```
  // array should be sorted
  for (int i = 1; i < len; i++) {
    if (array[i] < array[i-1]) return false;
  }

  return true;
}
```

C.5 cars

cars.c source code:

```
/* expected: Valid */
/* post: The error condition specified in the original benchmark did not occur. */

#include "common.h"

int f(int v1, int v2, int v3) {
  int x1;
  int x2;
  int x3;
  int t;

  x1=100;
  x2=75;
  x3=-50;
  t=0;

  init_choice;

  if ( (v3 >= 0) && (v1 <= 5) && (v1 -v3 >= 0) && (2* v2 - v1 - v3 == 0) && (v2 +5 >=0) && (v2 <= 5))
    {
      while (make_choice & 1)
        {
          if (-5 <= v2 && v2 <= 5)
            {
              if (2*x2-x1-x3 >= 0)
                {
                  x1 = x1 + v1;
                  x3 = x3 + v3;
                  x2 = x2 + v2;
                  v2 = v2 - 1;
                  t = t + 1;
                }
              else
                {
                  x1 = x1 + v1;
                  x3 = x3 + v3;
                  x2 = x2 + v2;
                  v2 = v2 + 1;
                  t = t + 1;
                }
            }
```

```
        }
      if (v1 > 5 || 2*v2 + 2*t  < v1 + v3 || 5*t  + 75 < x2 || v2 > 6 || v3 < 0 || v2 + 6 < 0
          || x2 + 5*t < 75 || v1 - 2*v2 + v3 + 2*t < 0 || v1 - v3 < 0)
        return 0;

    }
  return 1;
}
```

C.6 efm

efm.c source code:

```
/* expected: Valid */
/* post: The error condition specified in the original benchmark did not occur. */

#include "common.h"

int f(int X1, int X2, int X3, int X4, int X5, int X6) {

  init_choice;

  if ((X1>=1) && (X1<10000) /* overflow */ && (X2==0) && (X3==0) && (X4==1) && (X5==0) && (X6==0)) {

    while(make_choice & 1)
      {
        if (make_choice & 1)
          {
            if ((X1 >= 1) &&(X4 >= 1)) {
              X1=X1-1;
              X4=X4-1;
              X2=X2+1;
              X5=X5+1;
            }
            else if (make_choice & 1)
              {
                if (X2 >= 1 && X6 >= 1) {
                  X2 = X2 - 1;
                  X3 = X3 + 1;
                }
              }
            else if (make_choice & 1)
              {
                if (X3 >= 1 && X4 >= 1) {
                  X3 = X3 - 1;
                  X2 = X2 + 1;
                }
              }
            else if (make_choice & 1)
              {
                if (X3 >= 1) {
                  X3 = X3 - 1;
                  X1 = X1 + 1;
```

```
                    X6 = X6 + X5;
                    X5 = 0;
                  }
                }
              else
                {
                  if (X2 >= 1) {
                    X2 = X2 - 1;
                    X1 = X1 + 1;
                    X4 = X4 + X6;
                    X6 = 0;
                  }
                }
            }
        }

    if (X4 + X5 + X6 -1 != 0 || X4 + X5 > 1 || X5  < 0 || X4  < 0 || X3  < 0
        || X2  < 0 || X1 + X5 < 1 || X1 + X2 < X4 + X5 || X1 + X2 + X3 < 1)
      return false;
  }
  return true;
}
```

C.7 ex18

ex18.c source code:

```
/* expected: Valid */
/* post: Array accesses are within bounds. */

#include "common.h"

unsigned int a[5000];

int f(int k) {
  init_assume;
  init_assert;

  int i,j;

  assume(k >= 10); // original has 100, but that is pretty large

  for (i =0 ; i != k; i++) {
    assert(i >= 0 && i < k); // our check
    if (a[i] <= 1) break;
  }

  i--;

  for (j = 0; j < i; ++j) {
    assert(i >= 0 && i < k); // our check
    assert(j >= 0 && j < k); // our check
    a[j] = a[i];
```

```
  }

  myreturn(true);
}
```

C.8 ex30

ex30.c source code:

```
/* expected: Invalid */
/* post: The x and w arrays are equal. */

#include "common.h"

int x[1000];
int w[1000];

int f(int n)
{
  int max = 100;
  init_assert;
  if (n < 0 || n > max) return true;

  int xsize = n * 8;
  int wsize = 2 * n * 4;

  for (int i = 0; i < 2*n; i++) {
    assert (4*i < xsize);
    x[4*i] = -1;
  }

  for (int i = 0; i < n; i++) {
    assert (8*i < wsize);
    w[8*i] = -1;
    assert (8*i+4 < wsize);
    w[8*i+4] = -1; /* equal or not? */
  }

  for (int i = 0; i < 2*n; i++) {
    if (x[4*i] != w[4*i]) return false;
  }

  return assert_ok;
}
```

C.9 ex39

ex39.c source code:

```
/* expected: Valid */
/* post: The error condition specified in the original benchmark did not occur. */
```

```
#include "common.h"

int f()
{
  bool correct = false;

  int x,y;
  x = 0; y = 0;

  init_choice;
  init_assert;

  while (make_choice)
  {
    x++; y++;
  }

  x++; // bug

  while (x > 0) {
    x--;
    y--;
  }

  if (y != 0){
    assert(1==0);
  }

  myreturn(1);
}
```

C.10 fib

fib.c source code:

```
/* expected: Valid */
/* post: The computed value is the correct element in the fibonacci sequence. */

#include "common.h"

int fib[5000];

int f(int anything, int maxindex) {
  if (maxindex < 2) return true;

  int i = 0;

  fib[i++] = 1;
  fib[i++] = 1;

  for (int i = 2; i < maxindex; i++) {
    fib[i] = fib[i-1] + fib[i-2];
```

```
  }

  if (anything < maxindex && anything >= 2) {
    return fib[anything] == fib[anything-1] + fib[anything-2];
  }

  return true;
}
```

C.11 inf3

inf3.c source code:

```
/* expected: Valid */
/* post: The error condition specified in the original benchmark did not occur. */


/* modified from original: heap logic removed */

#include "common.h"

int a[5000];

int f(int n) {
  int tmp;

  init_assume;
  init_assert;

  if (n <= 0) {
    return true;
  }

  for (int i = 0; i<n; ++i) {
    a[i] = 0;
  }
  tmp = n;

  int sum=-1;
  for (int i = 0; i < n; i++) {
    if (sum == -1) {
      sum = a[i];
      assume(sum > 0);
    } else {
      sum += a[i];
      assume(sum > 0);
    }
  }

  if (sum <= 0) {
    assert(tmp <= 0);
  }
```

```
  myreturn(true);
}
```

C.12 prime

prime.c source code:

```
/* expected: Valid */
/* post: The chosen value is correctly identified as a prime or composite number. */


#include "common.h"

int prime[1000];

int f(int index, int sqrtmax, int anything) {

  if (sqrtmax <= 0 || index <= 0 || anything <= 0) return true;

  int max = sqrtmax*sqrtmax;
  if (max <= sqrtmax) return true;

  for (int i = 0; i < max; i++) {
    prime[i] = 1;
  }

  for (int i = 2; i <= sqrtmax; i++) {
    if (prime[i]) {
      for (int j = 2; i*j < max; j++) {
        prime[i*j] = 0;
      }
    }
  }

  int a = index;
  int b = anything;
  while (b > 0) {
    int tmp = b;
    b = a % b;
    a = tmp;
  }
  int gcd = a;

  if (index < max) {
    int table = prime[index];
    int pgcd = gcd == 1 || gcd == index;
    if (table) return pgcd;
  }

  return true;
}
```

C.13 sum

sum.c source code:

```
/* expected: Invalid */
/* but only for unrolls >= 7 */
/* post: The computed sum is always less than 200. */

#include "common.h"

int f()
{
  int count = 0;
  int sum = 0;
  int ok = 1;
  int x;
  do {
    x = make_choice;
    if (x > 30 || x < 0) {
      ok = 0;
    }
    sum += x;
  } while (x);

  if (count < 5) { return 1; }

  if (ok) { return (sum < 200); }

  return 1;

}
```

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1), November 2009. Cited on page 97.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 2nd edition, 2006. Cited on page 19.

[3] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, 1970. Cited on pages 42 and 44.

[4] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. Cited on page 79.

[5] T. Arons, E. Elster, S. Ozer, J. Shalev, and E. Singerman. Efficient symbolic simulation of low level software. In *Design, Automation and Test in Europe*, 2008. Cited on page 63.

[6] Bret Arsenault et al. Microsoft security intelligence report: Volume 8. Technical report, Microsoft, 2009. Cited on page 67.

[7] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, February 2011. Cited on pages 2, 4, 45, 69, 96, and 99.

[8] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the Association for Computing Machinery*, 57(2):74–84, February 2014. Cited on pages 4, 45, and 99.

[9] Domagoj Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008. Cited on page 64.

[10] Gogul Balakrishnan and Thomas Reps. WYSINWYX: what you see is not what you execute. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010. Cited on pages 2, 8, and 23.

[11] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2001. Cited on pages 46 and 63.

[12] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011. Cited on pages 2, 8, and 23.

[13] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, 2005. Cited on pages 47, 51, 52, 56, 57, and 64.

[14] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the Association for Computing Machinery*, 53(2):66–75, 2010. Cited on pages 2 and 18.

[15] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proceedings of the International Conference on Computer Aided Verification*, 2011. Cited on pages 2 and 18.

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2003. Cited on pages 2 and 18.

[17] Dionysus Blazakis. Interpreter exploitation. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2010. Cited on page 96.

[18] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie*, Wrocław, Poland, 2011. Cited on page 64.

[19] Peter Boonstoppel, Christian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2008. Cited on page 63.

[20] David Brumley. *Analysis and Defense of Vulnerabilities in Binary Code*. PhD thesis, Carnegie Mellon University School of Computer Science, September 2008. Cited on pages 6 and 8.

[21] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2007. Cited on page 18.

[22] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the International Conference on Computer Aided Verification*, 2011. Cited on pages 6, 23, 35, 41, 57, 86, and 100.

[23] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008. Cited on pages 69, 84, and 96.

[24] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), Aug 1986. Cited on page 115.

[25] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2008. Cited on pages 69 and 74.

[26] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, December 2008. Cited on pages 2, 45, 84, and 99.

[27] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the International Conference on Software Engineering*, 2011. Cited on page 46.

[28] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012. Cited on pages 2, 4, 45, 99, and 117.

[29] Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of low-level code using cooperating decompilers. In *Proceedings of the Static Analysis Symposium*, 2006. Cited on pages 19, 36, and 43.

[30] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2010. Cited on pages 94, 97, and 120.

[31] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of the Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, August 2009. Cited on pages 69 and 96.

[32] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the International Conference on Information Systems Security*, 2009. Cited on page 97.

[33] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994. Cited on pages 6, 19, 25, 36, 42, and 43.

[34] Cristina Cifuentes. Interprocedural data flow decompilation. *Journal of Programming Languages*, 4(2):77–99, 1996. Cited on pages 19 and 42.

[35] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995. Cited on page 26.

[36] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, 1998. Cited on pages 19 and 42.

[37] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007. Cited on page 84.

[38] John Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, 1970. Cited on page 44.

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, January 1977. Cited on pages 2, 15, and 18.

[40] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Symposium on Principles of Programming Languages*, 1989. Cited on page 51.

[41] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the ACM Workshop on Scalable Trusted Computing*, 2009. Cited on page 97.

[42] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the ACM Symposium on Information, Computer, and Communication Security*, 2011. Cited on page 97.

[43] Debian hardening. `https://wiki.debian.org/Hardening?action=recall&rev=73`. Checked 5/9/2014. Cited on page 72.

[44] The decompilation wiki. `http://www.program-transformation.org/Transform/DeCompilation`. Checked 5/9/2014. Cited on page 42.

[45] A detailed description of the data execution prevention (DEP) feature in windows XP service pack 2, windows XP tablet PC edition 2005, and windows server 2003. `http://support.microsoft.com/kb/875352/EN-US/`. Checked 5/9/2014. Cited on page 71.

[46] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. Cited on pages 24, 28, 35, and 44.

[47] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976. Cited on pages 47, 48, 50, and 57.

[48] DISC: Decompiler for TurboC. `http://www.debugmode.com/dcompile/disc.htm`. Checked 5/9/2014. Cited on pages 36 and 43.

[49] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008. Cited on pages 2 and 18.

[50] Thomas Dullien and Tim Kornau. A framework for automated architecture-independent gadget search. In *Proceedings of the USENIX Workshop on Offensive Technologies*, August 2010. Cited on pages 68, 69, 72, 74, 92, and 95.

[51] Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`. Checked 5/9/2014. Cited on page 58.

[52] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. Enhanced structural analysis for C code reconstruction from IR code. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, 2011. Cited on pages 19, 20, 22, 40, and 44.

[53] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2002. Cited on pages 45, 64, and 99.

[54] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001. Cited on pages 47, 51, 52, 53, 56, 57, 63, and 64.

[55] A. Fokin, K. Troshina, and A. Chernov. Reconstruction of class hierarchies for decompilation of C++ programs. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2010. Cited on pages 36 and 44.

[56] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ decompilation. In *Proceedings of the Working Conference on Reverse Engineering*, 2011. Cited on pages 36 and 44.

[57] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. Automatic discovery of API-level exploits. In *Proceedings of the International Conference on Software Engineering*, May 2005. Cited on page 96.

[58] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the International Conference on Computer Aided Verification*, July 2007. Cited on page 86.

[59] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005. Cited on pages 46 and 63.

[60] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008. Cited on pages 2, 45, and 99.

[61] Mike Gordon and Hélène Collavizza. Forward with Hoare. In *Reflections on the Work of C.A.R. Hoare*, chapter 5. Springer London, 2010. Cited on pages 47 and 51.

[62] Ilfak Guilfanov. Decompilers and beyond. Technical report, Hex-Rays SA, 2008. Cited on pages 19, 20, 26, 36, and 43.

[63] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proceedings of the ACM International Symposium on Foundations of Software Engineering*, 2006. Cited on page 58.

[64] Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009. Cited on page 96.

[65] Hex-rays decompiler: Manual: Failures and troubleshooting. `https://www.hex-rays.com/products/decompiler/manual/failures.shtml`. Checked 5/9/2014. Cited on page 37.

[66] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3), 1979. Cited on pages 6, 8, and 23.

[67] Michael Howard, Matt Miller, John Lambert, and Matt Thomlinson. Windows ISV software security defenses. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`. Checked 5/9/2014. Cited on pages 71 and 72.

[68] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the USENIX Security Symposium*, 2009. Cited on pages 68, 69, 72, 92, and 95.

[69] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual — volume 3A: System programming guide, part 1. Document number 253668, 2010. Cited on page 71.

[70] ISO/IEC JTC 1/SC 22. Information technology — programming languages — C. ISO/IEC 9899:2011, 2011. Cited on page 13.

[71] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011. Cited on page 14.

[72] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM Program Protection and Reverse Engineering Workshop*, 2014. Cited on page 44.

[73] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003. Cited on pages 46 and 63.

[74] Johannes Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010. Cited on pages 2, 6, 8, and 23.

[75] James C. King. Symbolic execution and program testing. *Communications of the Association for Computing Machinery*, 19(7):386–394, 1976. Cited on pages 46, 57, 63, and 115.

[76] Alfred Koelbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, December 2005. Cited on page 64.

[77] Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universität Bochum, 2009. Cited on pages 69, 74, and 95.

[78] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://www.suse.de/~krahmer/no-nx.pdf`, 2005. Checked 5/9/2014. Cited on page 95.

[79] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004. Cited on page 23.

[80] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2012. Cited on page 63.

[81] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2001. Cited on page 18.

[82] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium*, February 2011. Cited on pages 19, 23, 41, and 44.

[83] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005. Cited on pages 47 and 52.

[84] K. Rustan M. Leino. This is boogie 2. Technical Report KRML 178, Microsoft Research, 2008. Cited on pages 45, 64, and 99.

[85] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *International Conference on Dependable Systems and Networks*, 2008. Cited on page 96.

[86] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2010. Cited on pages 19 and 44.

[87] Le Dinh Long. Payload already inside: data re-use for ROP exploits. `https://media.blackhat.com/bh-us-10/whitepapers/Le/`

BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf, 2010. Checked 5/9/2014. Cited on page 96.

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005. Cited on pages 2 and 86.

[89] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *Proceedings of the International Conference on Compiler Construction*, 2002. Cited on page 43.

[90] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. Cited on page 23.

[91] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. Cited on pages 19, 20, 24, 27, 28, 33, 35, and 40.

[92] Tilo Müller. ASLR smack & laugh reference, 2008. http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf. Checked 5/9/2014. Cited on page 72.

[93] NECLA static analysis benchmarks. http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php. Cited on page 58.

[94] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction*, 2002. Cited on pages 2, 37, and 58.

[95] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989. Cited on page 48.

[96] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. PhD thesis, University of Cambridge, 2004. Cited on page 2.

[97] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005. Cited on pages 2 and 84.

[98] Larry Paulson, Tobias Nipkow, and Markarius Wenzel. Isabelle proof assistant. http://isabelle.in.tum.de. Checked 5/9/2014. Cited on page 48.

[99] PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt. Checked 5/9/2014. Cited on page 72.

[100] PaX non-executable stack (NX). http://pax.grsecurity.net/docs/noexec.txt. Checked 5/9/2014. Cited on page 71.

[101] Mathias Payer. Too much PIE is bad for performance. Technical Report 766, Eidgenössische Technische Hochschule Zürich, May 2012. Cited on page 72.

[102] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977. Cited on page 12.

[103] Alin Rad Pop. DEP/ASLR implementation progress in popular third-party windows applications. Technical report, Secunia, 2010. Cited on pages 67, 71, and 72.

[104] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2003. Cited on page 97.

[105] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009. Cited on pages 46 and 63.

[106] REC studio 4 — reverse engineering compiler. http://www.backerstreet.com/rec/rec.htm. Checked 5/9/2014. Cited on pages 36 and 43.

[107] Ryan Glenn Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, University of California, San Diego, 2009. Cited on pages 68, 69, 74, 92, and 95.

[108] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the Annual Computer Security Applications Conference*, 2009. Cited on pages 86 and 96.

[109] Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of the Joint Modular Languages Conference*, 2003. Cited on page 2.

[110] Edward J. Schwartz. The danger of unrandomized code. *USENIX ;login:*, 36(6), December 2011. Cited on page 97.

[111] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010. Cited on pages 2 and 84.

[112] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, 2011. Cited on pages 16, 45, and 99.

[113] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, 2013. Cited on page 16.

[114] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the ACM International Symposium on the Foundations of Software Engineering*, 2005. Cited on pages 2, 46, and 63.

[115] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007. Cited on pages 68, 69, 72, 77, and 95.

[116] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2004. Cited on page 72.

[117] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980. Cited on pages 20, 40, and 44.

[118] Doug Simon. Structuring assembly programs. Honours thesis, University of Queensland, 1997. Cited on page 43.

[119] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2011. Cited on pages 19 and 44.

[120] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections. `http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf`, 2008. Checked 5/9/2014. Cited on pages 72 and 96.

[121] Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. Directed proof generation for machine code. In *Proceedings of the International Conference on Computer Aided Verification*, 2010. Cited on page 2.

[122] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1995. Cited on page 64.

[123] Ubuntu security/features. https://wiki.ubuntu.com/Security/Features?action=recall&rev=94. Checked 5/9/2014. Cited on page 72.

[124] Arjan van de Ven. New security enhancements in red hat enterprise linux v.3, update 3. Technical report, Red Hat Linux, 2004. http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf. Checked 5/9/2014. Cited on pages 71 and 72.

[125] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007. Cited on pages 19, 36, 42, and 43.

[126] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -Overify: Optimizing programs for fast verification. In *Proceedings of the USENIX Conference on Hot Topics in Operating Systems*, 2013. Cited on page 66.

[127] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2012. Cited on page 18.

[128] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2013. Cited on page 4.

[129] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the Symposium on Principles of Programming Languages*, 2005. Cited on page 64.