Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs

PRESENTED BY **Athanasios Avgerinos**

ACCEPTED BY THE DEPARTMENT OF

Electrical and Computer Engineering

ADVISOR, MAJOR PROFESSOR — DATE 8/15/2014

DEPARTMENT HEAD — DATE 8/18/2014

APPROVED BY THE COLLEGE COUNCIL

DEAN — DATE

Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical & Computer Engineering

Athanasios (Thanassis) Avgerinos

Diploma, Electrical & Computer Engineering, National Technical University of Athens
M.S., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August, 2014

Abstract

Over the past 20 years, our society has become increasingly dependent on software. Today, we rely on software for our financial transactions, our work, our communications, even our social contacts. A single software flaw is enough to cause irreparable damage, and as our reliance on software increases, so does our need for developing systematic techniques that check the software we use for critical vulnerabilities.

In this dissertation, we investigate trade-offs in symbolic execution for identifying security-critical bugs. In the first part of the dissertation, we present symbolic execution systems capable of demonstrating control flow hijacks on real-world programs both at the source, and binary level. By exploiting specific trade-offs in symbolic execution, such as state pruning and careful state modeling, we show how to increase the efficacy of vanilla symbolic execution in identifying exploitable bugs.

In the second part of the dissertation, we investigate veritesting, a symbolic execution technique for exploiting the trade-off between formula expressivity and number of program states. Our experiments on a large number of programs, show that veritesting finds more bugs, obtains higher node and path coverage, and can cover a fixed number of paths faster when compared to vanilla symbolic execution. Using veritesting, we have checked more than 33,248 Debian binaries, and found more than 11,687 bugs. Our results have had real world impact with 202 bug fixes already present in the latest version of Debian.

Acknowledgments

First of all, I thank my adviser David Brumley. Without his continuous support during my first research steps, his constantly solid advice, and his guidance to follow my own ideas in research, this thesis would be impossible. The number of things I learned under your mentorship are innumerable, and I hope my future professional relationships are equally productive and educational.

Next, I would like to thank all three members of my committee: Virgil Gligor, André Platzer, and George Candea. Your comments during the early draft of my proposal steered my research in the right direction, and helped formulate my thesis statement. Your work and academic presence has been inspiring for me, and represents the main reason to consider an academic career.

I thank my friends, colleagues, and co-authors: Sang Kil Cha, Edward Schwartz, Alexandre Rebert, JongHyup Lee, and everyone one else in the research group. For all the great conversations we had on life and research, for the insightful comments and feedback I got during the drafting of this thesis, and for the collaborative environment you fostered, I thank you. This thesis would be impossible without you.

I thank my Greek "family" here in Pittsburgh: Yannis Mallios, Nektarios and Jill Leontiadis, Elli Fragkaki, Eleana Petropoulou, and everyone that stood by me. Without you, I would have never managed to survive for 5 years away from home. Thank you.

I thank my family: my father Yannis, my mother Sofia, and my brother Fotis. For the ethos you instilled in me, for your unconditional love, for your passion, and for your continuous support during my PhD, even in times of extreme hardship, I thank you and I love you forever. I also thank uncle Alexis, without whom I wouldn't have started the PhD, and my beloved grandparents, for everything they have done for me.

Last, but above all, I would like to thank the love of my life: Elisavet. We suffered through the pains of a long distance relationship for 5 years. I cannot wait for us to begin our life journey that lies ahead.

Thanassis Avgerinos / Θανάσης Αυγερινός

Funding Acknowledgments

This material was supported fully or in part by grants from the National Science Foundation (NSF), the Defense Advanced Research Projects Agency (DARPA), CyLab Army Research Office (ARO) grants, Lockheed Martin, and Northrop Grumman as part of the Cybersecurity Research Consortium. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

Preface

This dissertation is the compilation and adaptation of five papers [1, 2, 3, 4, 5] that have appeared in international conferences and journals. In compiling them, we have tried to remove duplicated material, update them according to recent advancements, enrich various sections with more detailed information, as well as include some of our most recent work.

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

AEG Automatic Exploit Generation. xii, xiv, xvii, 79–82, 89–101, 105, 109–111, 115–118, 122–145, 241, 247

BAP Binary Analysis Platform. 16

BIL Base Imperative Language. xii, xiii, 16, 22–28, 30, 43, 48, 49, 51, 52, 74, 75, 77

CFG Control Flow Graph. 25, 192, 196, 198, 199

QF_ABV Quantifier Free fixed-size Arrays & BitVectors. 20, 25, 28, 31, 48, 51, 74, 75, 77, 245

QF_BV Quantifier Free fixed-size BitVectors. xiii, 20, 24, 25, 28, 43, 72, 74, 227, 228, 245

SMT Satisfiability Modulo Theories. xiii, 28, 43–45, 50, 59, 62, 64, 66–69, 71, 74, 227, 236

Part I

Introduction

Chapter 1

Introduction

Write something down, and you may have just made a mistake.

— My Adviser, David, *Group meeting.*

Software bugs are expensive. A single software flaw is enough to take down spacecrafts [6], make nuclear centrifuges spin out of control [7], or recall 100,000s of faulty cars resulting in billions of dollars in damages [8]. Worse, security-critical bugs tend to be hard to detect, harder to protect against, and up to 100 times more expensive after the software is deployed [9]. The need for finding and fixing bugs in software before they become critical has led to the rapid development of automatic software testing tools.

Automatic testing allows developers and users to analyze their software for bugs and potential vulnerabilities. From blackbox random fuzzing [10, 11] to whitebox path-sensitive analyses [12, 13], the goal of automatic testing is the same: identify as many real flaws as possible with minimal user augmentation. Every discovered flaw is usually accompanied by a *test case*, an input that forces the program to exhibit the unexpected behavior. Test cases eliminate false positives, ensure reproducibility, and provide the developer with concrete and actionable information about the underlying problem. Even when no flaws are identified, the

set of exercised test cases serves as a regression test suite, a standard software engineering practice [14].

An increasingly popular[1] software testing technique is symbolic execution [16, 17, 18]. Unlike manual or random testing, symbolic execution systematically explores the program by analyzing one execution path at a time. For every feasible path, symbolic execution generates an input that exercises the path and then checks for potential vulnerabilities. Over the past decade, numerous symbolic execution tools have appeared—both in academia and industry—showing the effectiveness of the technique in a vast number of areas, including finding crashing inputs [19, 20], generating test cases with high coverage [21], creating input filters [22], exposing software vulnerabilities [23], and analyzing malware [24].

Albeit a well-researched technique, symbolic execution still faces two significant scalability challenges (Chapter 3). The first challenge, stems from the path-based nature of the technique: every branch in the program potentially doubles the number of paths that need to be explored. This doubling effect—colloquially known as the *path (or state) explosion*[2] problem—is exacerbated in larger programs, where the number of paths is typically very large. Path explosion is a well studied problem that persists throughout modern symbolic executor implementations [21, 23, 29, 3].

The second challenge comes from reasoning about safety. For every analyzed path, symbolic execution generates the condition (logical formula) under which all possible executions of the path are safe (Chapter 2). If the formula is falsifiable, safety can be violated and a counterexample is generated. Unfortunately, checking whether a formula is falsifiable is an NP-hard problem, and formula solving can quickly become the bottleneck. To mitigate the

[1] With more than 150 publications over the last decade according to an online listing [15].

[2] Depending on the context, the two terms may be used interchangeably [25, 26]—an "execution state" corresponds to a program path to be explored. Note, that the same term is used to define different problems (and thus should not be confused) both in symbolic execution [27], and model checking [28].

high solving times, the symbolic execution community has invested a lot time and effort in developing countermeasures, such as caches [21], and simplifications [21, 30, 12], which mitigate the problem but do not solve it in the general case.

This dissertation investigates trade-offs in symbolic execution, with the goal of finding security bugs. More specifically, the thesis is that *symbolic execution is capable of modeling, finding and demonstrating security-critical bugs such as control flow hijack attacks, and that state space management techniques such as state space pruning (Chapter 5), reduction (Chapter 6), and segmentation (Chapter 7) improve the effectiveness of symbolic execution as a testing and bug-finding tool.* We measure effectiveness—in the context of automatic testing—using four metrics: 1) the number of *real* bugs found, 2) the amount of code covered by generated test cases, and 3) the number of (distinct) exercised paths in a fixed amount of time, and 4) the amount of time required to explore a fixed number of paths. In this dissertation, we will show that state space management techniques can improve all of the above.

The underlying assumption of the thesis is that the core issues of symbolic execution (path explosion and formula solving) are unavoidable; a program can—in the worst case—have a number of states that is exponential in the number of branches, and solving a formula may require inverting a cryptographic hash function. Nevertheless, through state space pruning (Chapter 5), careful modeling of the execution state (Chapter 6), and the use of verification techniques for multi-path analysis (Chapter 7), state space management techniques can allow symbolic execution to find more bugs (Chapters 4, 6 and 7), cover more code and paths faster (Chapter 7), and become a more practical and effective testing tool for a large number of real programs.

Section 1.1 provides a brief description of the state space management techniques we propose with this thesis, Section 1.2 lists our contributions, and Section 1.3 provides the outline.

1.1 Scope

1.1.1 Automatic Exploit Generation

Not all bugs are equal. Some are irrelevant, e.g., an off-by-one error at the source code level may be always safe due to memory alignment; some are functional, e.g., an email client deletes an email instead of sending it; and some are security-critical, e.g., opening a malformed audio file gives user-level privileges to an attacker. Security-critical bugs are usually among the most dangerous, and thus should be identified and fixed first. However, how do we guarantee that the identified bugs are *indeed* security-critical and eliminate false positives?[3]

To tackle this question, we introduce the Automatic Exploit Generation (AEG) challenge (Chapters 4 and 5. Given a program, the AEG research challenge consists of automatically finding bugs *and* generating working exploits. The generated exploits unambiguously demonstrate that a bug is security-critical. Each feasible path is checked for exploitability by adding a set of constraints that are satisfied only by exploiting inputs. Our research targets memory corruption vulnerabilities, and control flow hijack exploits—i.e., inputs that overwrite the instruction pointer of the program and allow an attacker to run arbitrary code (shellcode) on a target system. We develop the first systems that can automatically find and reason about control flow hijacks (Chapters 5 and 6). While our framework may be applicable to other classes of exploits, e.g., information disclosure vulnerabilities; we do not explore them within the scope of the thesis.

1.1.2 State Pruning

Our first attempts to find security-critical vulnerabilities with symbolic execution on real programs were not very successful. Despite using state-of-the-art symbolic executors, such

[3]Note that the answer may differ per application. For instance, for a high-performance server a simple crash may be considered critical.

as KLEE [21], the analysis was unable to find exploitable paths in real programs due to state explosion (it is unlikely to find an exploitable state among the vast number of possibly irrelevant states). This observation was the motivation for preconditioned symbolic execution (Chapter 5).

Preconditioned symbolic execution first performs lightweight analysis to determine the heuristic conditions to exploit any lurking bugs, and then prunes the search space of paths that do not meet these conditions. For example, a lightweight program analysis may determine the minimum length to trigger any buffer overflow, which can then be used to prune symbolic execution paths corresponding to string inputs smaller than the minimum length.

Preconditioned symbolic execution is based on a trade-off: exchanging completeness (not all states will be explored, potentially missing vulnerabilities, or useful test cases), to explore fewer, more likely to be exploitable states. In our experiments (Chapter 5) we find that pruning the state space has significant impact on exploitable bug detection, going from 1 to 16 exploitable bugs in 14 open source applications. Two of the identified exploitable bugs were against previously unknown vulnerabilities. Our pruning techniques can be extended to preserve completeness — for example, by converting pruning to prioritization techniques — or combined with static analysis. Such extensions are possible future work and are considered outside the scope of this thesis.

1.1.3 State Reduction

In 2010, we started developing a binary symbolic executor, called MAYHEM (Chapter 6), capable of identifying security-critical bugs in binary programs (our previous attempts required source). Among the many new technical challenges at the binary level, a recurring one in exploitable bugs detection was pointer aliasing. The lack of source code abstractions such as types and variables, made resolving the value of a pointer much harder. For example, it

was not uncommon to have pointers that point to potentially thousands of distinct memory locations, or even all of memory (2^{32} cells in a 32-bit system).

One approach for handling aliased pointers is to perform concretization [13]. Concretization selects only one of the possible pointer values, and symbolic execution does not need to reason about aliasing. Conceptually, by concretizing the pointer, we simplify the logical formulas in symbolic execution, at the cost of restricting the possible values of pointers. Reasoning about all pointer values requires "forking" a single state for every possible value, leading faster to state explosion. The alternative approach is to encode all possible memory values in the formula [31, 32].

We found concretization overly constraining for identifying control flow hijacks in binary code (missing 40% of known exploitable bugs in our test suite [3]). On the other hand, encoding all possible memory information in logical formulas proved prohibitively expensive for formula solving. To address the technical challenge, we introduced an index-based memory model that allowed handling multiple memory values up to a threshold, and concretizing only as a fallback (Chapter 6). We also developed a number of encodings for multi-valued pointers, to simplify the amount of aliasing that occurs during formula solving.

The index-based memory model is again based on a trade-off: it uses more expressive formulas than concretization, since it encodes multiple pointer values per state, but does not attempt to encode *all* of them, which would be too expensive. Using the index-based memory model, we were able to find 40% more exploitable vulnerabilities in our experiments (Chapter 6).

1.1.4 State Segmentation

Symbolic execution for testing suffers from state explosion because of its path-based nature: every path is explored individually. On the other end of the spectrum, static verification techniques analyze all execution paths simultaneously [33, 34]. Reasoning about *all* program

paths at once has its own practical challenges; for example, solving the generated logical constraints that encode the absence of bugs becomes intractable—especially for large programs. We reconcile the two techniques to get the best of both worlds in a technique called veritesting.

Veritesting utilizes static multi-path verification techniques to allow symbolic execution to analyze multiple—not necessarily all—paths simultaneously (Chapter 7). By enabling multi-path symbolic execution, veritesting mitigates path explosion, and by avoiding merging all paths, formula solving does not become very expensive. Loops, and other practical aspects that are difficult to handle statically are addressed by symbolic execution.

The idea of veritesting is again based on a trade-off: veritesting uses static verfication techniques to exchange more expressive formulas for fewer states[4]. Further, veritesting allows symbolic execution to capitalize on performance improvements in formula solving; as formula solvers become faster, veritesting can be used to explore the trade-off further. We used veritesting in a system called MergePoint [5], to analyze thousands of programs. Our experimental results show, that veritesting can find twice as many bugs, cover orders of magnitude more paths, explore a fixed number of paths faster, and achieve higher code coverage than vanilla symbolic execution.

1.2 Contributions

This dissertation makes the following high-level contributions.

- An approach based on symbolic execution for automatically finding security-critical software bugs, such as control flow hijacks, in source and binary code (Chapters 4 to 6). The dissertation documents the design and implementation of the first end-to-end systems, and describes their effectiveness on real programs.

[4]Veritesting was inspired as an approach by other work in predicate abstraction [35], and model checking [36], which are also exploiting similar trade-offs between formula expressivity and number of states.

- A set of search strategies, including preconditioned symbolic execution and path prioritization heuristics (Chapter 5), for retargeting symbolic execution towards specific classes of exploitable bugs.

- A scalarized memory model for symbolic execution, along with a set of caching schemes and analyses, that allow more efficient reasoning on programs with symbolic pointers (Chapter 6).

- A technique, called veritesting, utilizing multi-path static analyses in symbolic execution to find more bugs, cover more code, and explore paths faster (Chapter 7).

Last, the dissertation documents current state-of-the-art approaches and trade-offs in symbolic execution (Chapter 3), as well as experimental data from applying symbolic execution on thousands of programs (Chapter 7).

1.3 Thesis Outline

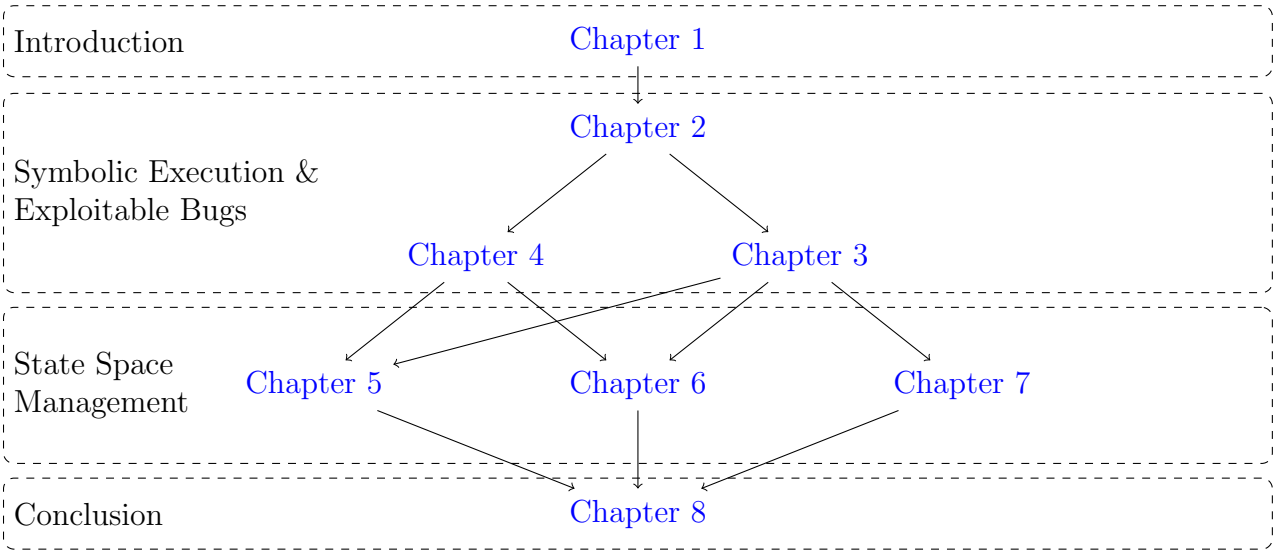


Figure 1.1: Chapter dependencies. An edge $a \rightarrow b$ means that a should be read before b.

Part I (Introduction). Chapter 1 introduces the reader to the problem of bug-finding, exploitability, and gives a high-level view of the main challenges in symbolic execution. The introduction ends with a (self-referencing) outline. Below, we provide a description of the three main parts of the thesis:

Part II (Symbolic Execution & Exploitable Bugs). Chapter 2 provides the necessary symbolic execution background for the rest of the thesis: the language definition, the execution semantics, and so on. Even if the reader is familiar with symbolic execution, we suggest a skim read of the chapter to get used to the notation and ensure a good grasp of the concepts. Chapter 3 builds a cost model for symbolic execution, and presents the (currently) main scalability challenges and trade-offs taxonomized by symbolic execution component. The chapter sets up the scaffolding and motivation for the state space management techniques presented in Part III.

Chapter 4 goes through the basics of standard memory corruption vulnerabilies, and security-critical bugs within the scope of the thesis. Using symbolic execution, we describe in detail how we can model, find, and demonstrate security-critical vulnerabilities. We present the unique challenges of automatic exploit generation, and discuss possible alternatives to the modeling problem.

Part III (State Space Management). The last part of the thesis focuses on state space management techniques. Chapter 5 details state pruning and prioritization techniques, such as preconditioned symbolic execution, and other prioritization heuristics we have used for more effective bug detection. Chapter 6 presents state reduction and query elimination techniques based on a memory model we developed, suited for finding and demonstrating exploitable bugs, and avoiding redundant queries (e.g., via caching). Chapter 7 introduces veritesting. We present the motivation and the base algorithm; we show how to integrate the technique with standard symbolic execution, and explore the trade-offs of the technique.

Part IV (Conclusion). Chapter 8 concludes the thesis. First, we give a brief summary of the thesis, present lessons learned, and propose a list of open problems for future research in the area.

We tried to make each chapter as self-contained as possible, but there are still certain dependencies. Figure 1.1 shows the dependencies between the various chapters; we recommend following the dependence graph to get a better understanding of advanced chapters.

Part II

Symbolic Execution & Exploitable Bugs

Chapter 2

Symbolic Execution

The beginning is the most important part of the work.

— Plato, *The Republic*

In this chapter, we introduce symbolic execution. We build up from basic notions, and gradually introduce notation and the main concepts. We assume familiarity with alphabets, sets, functions, languages, operational semantics and try to follow—wherever possible—the notation used by Sipser [37] and Pierce [38, 39].

The advanced reader can skip the language definition (Section 2.1) and introduction to symbolic execution (Section 2.3), and move ahead to the main algorithm (Figure 2.1). We advise against skipping the entire chapter, since the notation and terms will be reused heavily in follow-up chapters.

2.1 A Base Imperative Language (BIL)

Program analyses are defined with respect to a language. A typical example is the WHILE language [40], a simple, structured, imperative programming language. Table 2.1 shows the syntax of WHILE statements. WHILE has empty statements (skip), assignments (:=),

stmt S	::=	skip | $var := exp$ | S_1 ; S_2 | if exp then S_1 else S_2 | while exp do S

Table 2.1: WHILE: a simple, structured imperative language.

sequences of statements (;), conditional branches (if then else), and a looping construct (while do). We will use WHILE as a baseline for structured languages.

The WHILE definition shown in Table 2.1 is missing two important features. First, the language has no construct for checking properties and modeling failures, e.g., an assert statement. Second, the language cannot directly model unstructured programs, i.e., programs that use `goto` statements.

Throughout the thesis we present algorithms and techniques for analyzing programs written in a low-level, unstructured, imperative programming language (typically assembly). We make heavy use of assert statements to check for properties and `goto`s are necessary for modeling low-level code. Thus, we will use WHILE only for demonstration purposes; to model and analyze actual programs, we introduce a base imperative language (Sections 2.1.1 to 2.1.3) capable of expressing low-level programming constructs and errors.

We start by introducing the input domain of programs (Section 2.1.1). Next, we define the expressions of the language (Section 2.1.2); note that the expression definition (*exp*) in WHILE was intentionally left open, since expressions will be defined in a follow-up section. Finally, we introduce the BIL language, an unstructured language capable of modeling assembly programs for commodity architectures. All languages we present are customized variants of the Binary Analysis Platform (BAP) Intermediate Language [41].

2.1.1 Input Domain

Program inputs are modeled as bitvectors of finite length[1]. Specifically:

[1]In this work we only consider finite-sized inputs.

exp	::=	$\mathsf{value} \mid var \mid \Diamond_u\, exp \mid exp\, \Diamond_b\, exp \mid \mathsf{ite}(exp, exp, exp) \mid \mathsf{let}\ var = exp\ \mathsf{in}\ exp$
		$\mid \mathsf{value}_\mu \mid \mathsf{load}(exp, exp) \mid \mathsf{store}(exp, exp, exp)$ (ABV only)
value_μ	::=	a concrete bitvector array, e.g., $\{\texttt{0xbfffff8c} \rightarrow \texttt{0x41}\}$ $(: bv_{32} \rightarrow_\mu bv_8)$
value	::=	a concrete bitvector, e.g., $\texttt{0xbfffff8c}$ $(: bv_{32})$
$\Diamond_b$	::=	typical binary operators, e.g., $+, -, \%, ::$ etc.
$\Diamond_u$	::=	typical unary operators, e.g., $\neg$, 2's complement, etc.

Table 2.2: BV and ABV expression syntax. Type annotations are omitted for brevity.

Definition 1 (Input.)**.** *Let Σ be the binary alphabet $\Sigma = \{0, 1\}$, and Σ^* be the set of all finite binary strings. Every element $\iota \in \Sigma^*$ is an input.*

Definition 2 (Input Size.)**.** *The size, or length, or bitlength of an input $\iota \in \Sigma^*$ is the number of symbols (bits) that it contains, and is denoted by $|\iota|_b$. The set of all inputs of fixed size n is denoted by Σ^n.*

In the rest of the section we consider programs with a single fixed size input. Programs with multiple inputs can be modeled similarly by encoding them in a single binary string. Programs with inputs of potentially infinite size cannot be fully modeled and their input spaces need to be bounded or truncated.

2.1.2 Expressions & Types

Program inputs are modeled as bitvectors, and so are expressions. Specifically, our expression definition contains two main types: 1) bitvectors of a specific length n denoted by bv_n, and 2) arrays of bitvectors denoted by $bv_m \rightarrow_\mu bv_n$ for an array indexed by bv_m bitvectors and containing bv_n bitvectors. For example, a single byte has type bv_8, while a 32-bit addressable

byte-level memory has type $bv_{32} \rightarrow_{\mu} bv_8$. We also use the *boolean* type, which is syntactic sugar for bv_1 (with the mapping $0 \rightarrow false$ and $1 \rightarrow true$).

The syntax for expressions is shown in Table 2.2, showing the two expression languages. The first language, which we will call the BV expression language, contains only scalar bitvector expressions and consists of constants (value), variables (var), binary ($e_1 \diamondsuit_b e_2$), unary ($\diamondsuit_u e$), and ternary if-then-else operators ($\mathsf{ite}(b, e_1, e_2)$). The expression definition also includes let expressions ($\mathsf{let}\ var = e_1\ \mathsf{in}\ e_2$), which are used to "store" intermediate expressions in temporary variables. In a follow-up chapter, we will see that `let` constructors are not necessary and merely syntactic sugar for expressing expression reuse in textual format (Chapter 7).

The ABV language is a superset of BV and contains three more constructs for modeling arrays: concrete arrays of bitvectors (value_{μ}), loading from an array ($\mathsf{load}(e_{\mu}, e_i)$), and storing to an array ($\mathsf{store}(e_{\mu}, e_i, e_v)$), where e_{μ} is the array expression, and e_i is the index. For simplicity, we define the following shorthands for load/store operations: $e_{\mu}[e_i]$ is equivalent to $\mathsf{load}(e_{\mu}, e_i)$ and $e_{\mu}[e_i] \leftarrow e_v$ is equivalent to $\mathsf{store}(e_{\mu}, e_i, e_v)$.

Evaluating Expressions. An expression can be evaluated and reduced to a concrete value—a bitvector or array of bitvectors. We now provide the computational meaning of expressions by giving the semantics of the expression evalution operator $\Downarrow$ under a variable context. We use the notation $\Gamma \vdash e \Downarrow v$ to denote that variable context Γ entails ($\vdash$) that expression e evaluates ($\Downarrow$) to value v. The variable context Γ maps each variable to a concrete value. For example, using a context $\Gamma = \{x \rightarrow 5\}$ we can reduce $x + 2$ to 7, denoted as $\{x \rightarrow 5\} \vdash x + 2 \Downarrow 5 + 2 = 7$. Thus, evaluating expressions reduces to performing variable substitution using Γ:

$$\frac{}{\Gamma \vdash v \Downarrow v}\ \textsc{Const} \qquad \frac{var \in \Gamma \quad v = \Gamma[var]}{\Gamma \vdash var \Downarrow v}\ \textsc{Var} \qquad \frac{\Gamma \vdash e \Downarrow v}{\Gamma \vdash \Diamond_u\, e \Downarrow \Diamond_u\, v}\ \textsc{Unop}$$

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma \vdash e_2 \Downarrow v_2}{\Gamma \vdash e_1 \Diamond_b e_2 \Downarrow v_1 \Diamond_b v_2}\ \textsc{Binop} \qquad \frac{\Gamma \vdash b \Downarrow v_b \quad \Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma \vdash e_2 \Downarrow v_2}{\Gamma \vdash \mathsf{ite}(b, e_1, e_2) \Downarrow \mathsf{ite}(v_b, v_1, v_2)}\ \textsc{Ite}$$

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma[var \to v_1] \vdash e_2 \Downarrow v_2}{\Gamma \vdash \mathsf{let}\ var = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}\ \textsc{Let}$$

$$\frac{\Gamma \vdash e_\mu \Downarrow v_\mu \quad \Gamma \vdash e_i \Downarrow v_i}{\Gamma \vdash e_\mu[e_i] \Downarrow v_\mu[v_i]}\ \textsc{Load} \qquad \frac{\Gamma \vdash e_\mu \Downarrow v_\mu \quad \Gamma \vdash e_i \Downarrow v_i \quad \Gamma \vdash e_v \Downarrow v_v}{\Gamma \vdash e_\mu[e_i] \leftarrow e_v \Downarrow v_\mu[v_i] \leftarrow v_v}\ \textsc{Store}$$

The evaluation rules above are read bottom to top, left to right, and have the form $\frac{premise}{expr \Downarrow evaluated\_expr}$. If the premise fails, the rule does not apply and evaluation fails. For example, in the VAR rule, concretely evaluating a variable will fail if the variable is not in the context ($\texttt{var} \notin \Gamma$). Note that we use a standard convention [39] and leave the evaluation relation undefined for expressions where evaluation fails; there is an implied set of rules (the complement of the ones we present) that leads to a failed evaluation.

The example evaluation above ($\{x \to 5\} \vdash x + 2 \Downarrow 5 + 2 = 7$) can now be explained in detail by showing the sequence of evaluation rules that apply:

$$\dfrac{\dfrac{x \in \{x \to 5\} \quad 5 = \{x \to 5\}[x]}{\{x \to 5\} \vdash x \Downarrow 5}\ \textsc{Var} \qquad \dfrac{}{\{x \to 5\} \vdash 2 \Downarrow 2}\ \textsc{Const}}{\{x \to 5\} \vdash x + 2 \Downarrow 7\ (= 5 + 2)}\ \textsc{Binop}$$

Note that we use operator overloading in our semantics. For instance, the binop ($\Diamond_b$) operator used in the BINOP rule is not the same on the left and right of the $\Downarrow$ operator. The $\Diamond_b$ on the left of $\Downarrow$ is an expression (*exp*) operator, while the $\Diamond_b$ on the right is the concrete binary operator operating on values, which is why we can reduce $5 + 2$ to 7 in the example above.

Our expression definition has no quantifiers, functions, or support for recursion. We believe adding such constructs is important, and can be attempted in future work. In this thesis, we use the basic primitives introduced by our *BV* and *ABV* languages (Table 2.2). The naming convention used for our expression languages is intentional; to reason about programs we will rely on the QF_BV and Quantifier Free Arrays & BitVectors (QF_ABV) logics (Section 2.2.2).

Atoms & Compound Expressions. Our expression definition is recursive, i.e., some expressions are defined with respect to other expressions. For example, $e_1 \diamondsuit_b e_2$ requires two other expressions (e_1 and e_2) to be constructed first, while a constant value does not. Thus, we distinguish expressions into two groups: atoms and compound expressions.

Definition 3 (Atom.)**.** *An atom is an instance of the exp construct (Table 2.2) with no deeper expression structure, i.e., an atom definition does not require other expressions.*

Definition 4 (Compound Expression.)**.** *A compound expression is an instance of the exp construct (Table 2.2) with deeper expression structure, i.e., a compound expression definition requires other expression instances. We refer to expressions required for the definition of a compound expression as* child *expressions.*

Example atomic expressions are value, value_μ, *var*; all other *exp* constructors are compound, e.g., $e_1 \diamondsuit_b e_2$ is a compound expression and e_1, e_2 are child expressions.

Expression Size, Depth & Variable Size. We now define three expression attributes that will help us quantify the size of expressions: 1) size, 2) depth, and 3) variable size.

Definition 5 (Expression Size.). *The size of an expression e, denoted by $|e|_s^e$ is:*

$$|e|_s^e = \begin{cases} 1 & , \textit{ if } e \textit{ is an atom} \\ 1 + \sum_{e_c \in children(e)} |e_c|_s^e & , \textit{ otherwise} \end{cases}$$

Definition 6 (Expression Depth.). *The depth of an expression e, denoted by $|e|_d^e$ is:*

$$|e|_d^e = \begin{cases} 1 & , \textit{ if } e \textit{ is atom} \\ 1 + \max_{e_c \in children(e)} |e_c|_d^e & , \textit{ otherwise} \end{cases}$$

Definition 7 (Expression Variable Size.). *Let Var be the set of variables in an expression e. The variable size of expression e is defined as the number of variable bits:*

$$|e|_v^e = \sum_{var \in Var} |var|_b$$

Example 1 (Expression Attributes.). *Consider the expression $x + (y + 2)$, where x, y, 2 are 32-bit bitvectors (bv_{32}). We compute the various attributes:*

$$|x + (y + 2)|_s^e = 5$$

$$|x + (y + 2)|_d^e = 2$$

$$|x + (y + 2)|_v^e = 64$$

Using the above attributes we can construct special classes of the BV, ABV expression languages, that restrict the possible valid expressions of the language. For example, we can restrict the BV language to expressions with depth up to 1 (either atoms or composites with atoms as children), and denote the language by $BV_{|e|_d^e \leq 1}$. Similarly, we describe other expression classes with constraints on size (e.g., $BV_{|e|_s^e \leq c}$), variable size (e.g., $BV_{|e|_v^e \leq c}$) or combinations of constraints (e.g., $BV_{|e|_d^e \leq 2, |e|_s^e \leq 5}$). We will explore these ideas further in the next section (Section 2.1.3).

$program$	::=	$\langle \mathsf{value}, stmt \rangle^*$
$stmt$	::=	$var := exp_1 \mid \mathsf{assert}\ exp_1 \mid \mathsf{if}\ exp_1\ \mathsf{jump}\ exp_1 \mid \mathsf{halt}$
exp_1	::=	$atom \mid \Diamond_u\ atom \mid atom\ \Diamond_b\ atom \mid \mathsf{ite}(atom, atom, atom) \mid$
		$\mathsf{let}\ var = atom\ \mathsf{in}\ atom \mid \mathsf{load}(atom, atom) \mid \mathsf{store}(atom, atom, atom)$
$atom$	::=	$\mathsf{value} \mid \mathsf{value}_\mu \mid var$

Table 2.3: BIL: a simplified version of the BAP IL [41]. BIL is close to 3-address code (with flat expressions) and models low-level assembly-like languages. For brevity, we omit type annotations, and features such as dynamically generated code, unmodeled behavior, etc.

2.1.3 The Base Intermediate Language

To model programs, we use BIL, a representative low-level imperative language with assignments, assertions, conditional jumps and a termination statement. Table 2.3 shows BIL in BNF. BIL is close to 3-address code (with flat expressions—$ABV_{||^e_d \leq 1}$). To demonstrate how programs in BIL execute we present the concrete operational semantics in Figure 2.1, in the form of small-step transition relations $state \rightsquigarrow state'$ from an abstract state to another. Again, rules are read bottom to top, left to right, and have the form $\frac{premise}{state \rightsquigarrow state'}$. The abstract state is the union of three types:

$$\mathsf{Halted} \quad | \quad \mathsf{Error} \quad | \quad \Gamma$$

1) Halted is a final (accepting) state and signifies the program terminated normally, 2) Error is a final state and signifies the program terminated with an error, and 3) Γ is a variable context (the current machine state) mapping variables to their values.

We define two unique variables that always populate the variable context:

$$\frac{\Gamma[pc] \in P \quad var := e = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow v \quad \Gamma' = \Gamma[var \to v]}{\Gamma \rightsquigarrow \Gamma'[pc \to \Gamma[pc] + 1]} \textsc{Assign}$$

$$\frac{\Gamma[pc] \in P \quad \mathsf{assert}\ e = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow true}{\Gamma \rightsquigarrow \Gamma[pc \to \Gamma[pc] + 1]} \textsc{Assert}$$

$$\frac{\Gamma[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow false}{\Gamma \rightsquigarrow \Gamma[pc \to \Gamma[pc] + 1]} \textsc{FCond}$$

$$\frac{\Gamma[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow true \quad \Gamma \vdash e_1 \Downarrow v_1}{\Gamma \rightsquigarrow \Gamma[pc \to v_1]} \textsc{TCond}$$

$$\frac{\Gamma[pc] \in P \quad \mathsf{halt} = \mathsf{instFetch}(P, \Gamma[pc])}{\Gamma \rightsquigarrow \mathsf{Halted}} \textsc{Halt}$$

Figure 2.1: Concrete execution semantics of BIL for a given program P.

- The program counter, denoted by pc, that determines the next statement to be executed. $\mathsf{instFetch}(P, pc)$ returns the statement—if it exists—of program P with counter pc (note statements are labeled by a value, as shown in Table 2.3).

- The program memory, denoted by μ. Unless explicitly mentioned, we will assume that all modeled programs have a single globally addressable memory (e.g., for a 32-bit system: $\mu{:}bv_{32} \to_{\mu} bv_8$).

The transition relation is undefined for states where the execution is considered unsafe. For instance, the transition relation is undefined if the program counter is outside the program code (i.e., when $pc \notin P$ and $\mathsf{instFetch}(P, pc)$ would also fail). Similarly, the transition relation is undefined for assertions that fail (the expression can only evaluate to true). For all those cases there is an implicit transition to the Error state.

Adding more features to the language and semantics is straightforward. For instance, adding dynamically generated code requires a redefinition of the $\mathsf{instFetch}(,)$ primitive. Instead of fetching a fixed statement from P, it would translate $\mu[pc]$ ($\mathsf{instFetch}(\mu, pc)$) to a

statement (based on a decoding scheme) and execution would proceed as above. To avoid relying on a specific statement decoding scheme our default BIL definition does not include dynamically generated code. We will present techniques that handle dynamically generated code (**??**), but our default notion of a program will not include dynamically generated code, unless otherwise specified.

2.1.4 Combining, Restricting, & Enhancing Languages

Our statement definition for BIL above is the scaffolding for expressing other language variants. Similarly, we can use QF_BV expressions with statements from the WHILE langauge to create a variant of the WHILE language. To express language variants we will use the syntax:

$$\mathsf{StatementLanguage}_{stmt\_modifiers}(\mathsf{ExpressionLanguage}_{exp\_modifiers})$$

Expression modifiers were introduced above (Section 2.1.2), and regard expression attributes such as size, depth, etc. Similarly, we introduce the following modifiers for statements:

- Acyclicity, denoted by $\mathcal{A}$. Acyclic programs cannot execute the same statement twice. For certain languages this can be enforced syntactically, e.g., by removing the `while` construct from the WHILE language; others require control flow analysis.
- Scalar, denoted by $\mathcal{S}$. Scalar programs contain only scalar variables (no array variables), and can only operate on QF_BV expressions.
- Memory-only, denoted by $\mathcal{M}$. Memory-only programs do not have scalar variables, and the program operates on a single global array: the memory variable μ.
- Known (constant) control flow, denoted by $\mathcal{K}$. All jump targets are known constants, i.e., $\mathsf{if}\ e\ \mathsf{jumpvalue}$ is the only acceptable control flow construct. This contraint simplifies

the recovery of a Control Flow Graph (CFG), ensures each conditional jump has at most two possible next statements, but also prohibits important low-level features (e.g., computed jumps).

We will use the above modifiers to compactly express language definitions. The explicit modifiers allows us to relate the expressivity of the language with its properties.

Example 2 (Language Definitions.)**.** *We present a set of sample language definitions for exposition purposes:*

- $BIL(QF\_ABV_{||^e_d \leq 1})$. *BIL statements with QF_ABV expressions up to depth 1 (the default definition BIL shown in Table 2.3).*
- While(QF_BV). While *statements with arbitrary QF_BV expressions.*
- $BIL_{\mathcal{A}}(QF\_BV_{||^e_s \leq 1})$. *BIL acyclic programs with QF_BV expressions up to size 1.*
- While$_{\mathcal{A},\mathcal{S}}(QF_BV)$. *Acyclic* While *programs without memories (only scalar variables) and arbitrary QF_BV expressions.*
- $BIL_{\mathcal{M},\mathcal{K}}(QF\_ABV)$. *BIL programs without scalar variables (only a global memory), known constant jump targets and arbitrary QF_ABV expressions.*

2.2 Basic Definitions

We now define basic concepts for BIL programs, including the notion of traces, paths, and correctness.

2.2.1 Traces, Paths & Programs

Definition 8 (Trace.). *A trace is the sequence of transition states during program execution. Given a deterministic*[2] *program* P *and an input* $\iota \in \Sigma^n$*, the trace is denoted by* $tr(P, \iota)$*.*

The size n of the input domain Σ^n depends on the size of the initialized variables (inputs) of Γ when execution starts.

Definition 9 (Program Path.). *A program path is the sequence of statements fetched during program execution, and is denoted by* π*. The path derived from a trace* $tr(P, \iota)$ *is denoted by* $\pi_{tr(P,\iota)}$*.*

We say that a trace $tr(P, \iota)$ traverses a program path π when $\pi = \pi_{tr(P,\iota)}$. Note that multiple traces can traverse the same path, e.g., two different inputs may execute the same sequence of statements. The execution of a trace and the transition between states are defined by the operational semantics of the language (e.g., Figure 2.1 for BIL).

Definition 10 (Program, Trace, and Path Size.). *The size of a program* P *is defined as the number of statements in* P*, and denoted by* $|P|$[3]*. The size of a trace* $|tr(P, \iota)|$ *(respectively path* $|\pi|$*) is defined as the number of states (statements) in a trace* $tr(P, \iota)$ *(path* π*).*

Note that for programs with loops, the size of a trace (or path) may exceed the size of a program. Also, note that the size of a path derived from a trace is equal to the size of the trace $|tr(P, \iota)| = \left|\pi_{tr(P,\iota)}\right|$.

Definition 11 (All Paths.). *Given a program* P*, and an input domain* Σ^n*, the set of all paths is defined as:*

$$\mathcal{F}\left(P, \Sigma^n\right) = \bigcup_{\iota \in \Sigma^n} \pi_{tr(P,\iota)}$$

[2] In this work we only consider deterministic programs; we do not discuss non-deterministic constructs such as the choice ($\square$) operator in the Guarded Command Language [42].

[3] Our program size definition considers static programs without dynamically generated code.

For a given Γ, input $\iota \in \Sigma^n$ corresponds to an n-bit bitvector representing all input variables in Γ, where n is the sum of their bitlengths.

Example 3 (Traces & Paths.)**.** *Consider the following example, (where input* : bv_{32}*):*

1 if $input == 42$ jump 3

2 $x := input + 17$

3 halt

The program above has three statements, so $|P| = 3$*. There are* 2^{32} *possible inputs, thus there are* 2^{32} *distinct traces. It has two distinct paths* $|\mathcal{F}(P, \Sigma^{32})| = 2$*; the first path* π_1 *(statements 1, 2, 3) corresponds to* $2^{32} - 1$ *traces of size* $|\pi_1| = 3$*, while the second* π_2 *(statements 1, 3) corresponds to a single trace of size* $|\pi_2| = 2$

2.2.2 Correctness & Bitvector Logics

The execution of a BIL program can result in three possible states: Halted, Error, or Γ (loop forever). We define correctness of an execution with respect to the resulting state:

Definition 12 (Trace Correctness.)**.** *A trace* $tr(P, \iota)$ *is correct when the final transition state—if it exists—is not the* Error *state*[4]*.*

Definition 13 (Path Correctness.)**.** *A program path is correct, when all traces traversing the same path are correct. Given an program* P*, an input domain* Σ^n *and path* π*:*

$$\pi \text{ is correct} \Leftrightarrow \forall \iota \in \Sigma^n : \pi = \pi_{tr(P,\iota)} \implies tr(P, \iota) \text{ is correct}$$

Definition 14 (Program Correctness.)**.** *Given an input domain* Σ^n*, we define correctness for a program* P *as:*

$$P \text{ is correct} \Leftrightarrow \forall \pi \in \mathcal{F}(P, \Sigma^n) : \pi \text{ is correct} \Leftrightarrow \forall \iota \in \Sigma^n : tr(P, \iota) \text{ is correct}$$

[4]Our correctness definition marks infinite loops as correct.

To reason about correctness, we will use techniques that reduce the problem of reasoning about programs and paths written in a programming language (BIL in our examples) to the domain of logic. Specifically we will convert correctness to a SMT problem [43] and we will rely on two main theories: QF_BV and QF_ABV. The syntax of formulas written for these theories is identical to our expression syntax (Table 2.2). Thus, we need a technique for converting a program written in BIL to an expression of type *boolean* that determines correctness: a logical *formula*.

Before proceeding to the technique (Section 2.3) we define the notions of validity and satisfiability for a logical formula:

Definition 15 (Validity & Satisfiability.)**.** *A logical formula f that contains n free variables $x_1, \ldots, x_n$ is:*

$$\text{valid } \textit{iff } \forall x_1, \ldots, x_n : f(x_1, \ldots, x_n) = \textit{true}$$

$$\text{satisfiable } \textit{iff } \exists x_1, \ldots, x_n : f(x_1, \ldots, x_n) = \textit{true}$$

2.3 Basics of Symbolic Execution

Symbolic execution [] is a program analysis technique that enables reasoning about program correctness in the domain of logic[5]. The name of the analysis conveys its main principles. First, the analysis is *symbolic*; instead of operating on concrete inputs (e.g., instances of the value type), program inputs are substituted by symbols (variables) that represent all possible inputs. Second, the analysis is an *execution*; program statements are evaluated in the forward direction—similar to an interpreter, program values are computed as a function of the input symbols, and a symbolic execution context (denoted by Γ) mapping each variable to a value is maintained throughout execution.

[5]In this thesis, we use the term symbolic execution to refer to path-based dynamic symbolic execution; when we need to refer to static symbolic execution, we will explicitly mention it.

The Path Predicate. For each path, symbolic execution builds up a logical formula that represents the condition under which the program will execute the exact same path. The logical formula is expressed in terms of the input variables and is called the *path predicate*[6] (denoted by Π). In the following sections, we show how to construct the path predicate and the symbolic execution context by providing the operational semantics of symbolic execution both for traces/paths (Section 2.3.1) and entire programs (Section 2.3.2).

The Goal of Symbolic Execution. The two main elements offered by symbolic execution are: 1) the path predicate (Π), and 2) the current symbolic execution context (Γ). Using these elements, symbolic execution allows us to construct and potentially answer queries of the form: "given Π, does property X hold on state Γ?", i.e., we can check specific properties—for any trace following the same path—based on the current abstract machine state. One such property is the validity of assertions (e.g., in $\mathsf{assert}\ e$, is e always *true*?), and thus reason about correctness. In follow-up chapters Chapters 4 to 6 we will discuss other properties that can be checked.

2.3.1 Trace-Based Symbolic Execution

Given a trace, symbolic execution allows us to reason about all other traces that traverse the same path. To do so, it uses symbols to represent inputs and converts the statements of the path to a logical formula. Figure 2.2 shows the operational semantics of symbolic execution on a trace. The abstract state is a 3-tuple $(\Pi, \Gamma, \mathit{concrete})$, where Π is the path predicate, and Γ is the symbolic mapping for variables, and *concrete* is the concrete state of the execution ($\mathsf{Halted} \mid \mathsf{Error} \mid \Gamma^c$). The concrete context of the trace Γ^c guides symbolic execution.

[6] In the bibliography it is also found as: path condition, path formula, or path constraint. Typically, these terms are used interchangeably.

$$\frac{\Gamma^c[pc] \in P \quad var := e = \mathsf{instFetch}(P, \Gamma^c[pc]) \quad \Gamma^c \vdash e \Downarrow_s v \quad \Gamma^{c\prime} = \Gamma^c[var \to v] \quad \Gamma \vdash e \Downarrow_s v_s \quad \Gamma' = \Gamma[var \to v_s]}{\Pi, \Gamma, \Gamma^c \rightsquigarrow \Pi, \Gamma'[pc \to \Gamma[pc]+1], \Gamma^{c\prime}[pc \to \Gamma^c[pc]+1]} \textsc{Assign}$$

$$\frac{\Gamma^c[pc] \in P \quad \mathsf{assert}\ e = \mathsf{instFetch}(P, \Gamma^c[pc]) \quad \Gamma^c \vdash e \Downarrow_s true \quad \Gamma \vdash e_1 \Downarrow_s v_s \quad \mathsf{isValid}(\Pi \implies v_s) \quad \Pi' = \Pi \wedge v_s}{\Pi, \Gamma, \Gamma^c \rightsquigarrow \Pi', \Gamma[pc \to \Gamma[pc]+1], \Gamma^c[pc \to \Gamma^c[pc]+1]} \textsc{Assert}$$

$$\frac{\Gamma^c[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma^c[pc]) \quad \Gamma^c \vdash e \Downarrow_s true \quad \Gamma^c \vdash e_1 \Downarrow_s v_1 \quad \Gamma \vdash e \Downarrow_s v_s \quad \Gamma \vdash e_1 \Downarrow_s v_{1s} \quad \Pi' = \Pi \wedge v_s \wedge v_{1s} = v_1}{\Pi, \Gamma, \Gamma^c \rightsquigarrow \Pi', \Gamma[pc \to v_{1s}], \Gamma^c[pc \to v_1]} \textsc{TCond}$$

$$\frac{\Gamma^c[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma^c[pc]) \quad \Gamma^c \vdash e \Downarrow_s false \quad \Gamma \vdash e \Downarrow_s v_s \quad \Pi' = \Pi \wedge \neg v_s}{\Pi, \Gamma, \Gamma^c \rightsquigarrow \Pi', \Gamma[pc \to \Gamma[pc]+1], \Gamma^c[pc \to \Gamma^c[pc]+1]} \textsc{FCond}$$

$$\frac{\Gamma^c_{pc} \in P \quad \mathsf{halt} = \mathsf{instFetch}(P, \Gamma^c[pc])}{\Pi, \Gamma, \Gamma^c \rightsquigarrow \Pi, \Gamma, \mathsf{Halted}} \textsc{Halt}$$

Figure 2.2: Symbolic execution operational semantics for BIL traces. The first line in the premise contains the concrete semantics (same as in the concrete semantics), while the second line—when it exists—contains the symbolic execution semantics.

We note that on every conditional jump or assertion, the path predicate is updated to keep track of all conditions that can affect program execution. The premise of the Assert rule also contains an isValid() routine to ensure that no assertion can be violated.

The symbolic evaluation semantics for expressions ($\Downarrow_s$) are also straightforward (we show some of the rules for exposition; the rest are similar):

$$\frac{}{\Gamma \vdash v \Downarrow_s v}\ \textsc{Const} \qquad \frac{var \in \Gamma \quad v_s = \Gamma[var]}{\Gamma \vdash var \Downarrow_s v_s}\ \textsc{Var}$$

$$\frac{var \in \Gamma \quad v_s = \Diamond_u\, \Gamma[var]}{\Gamma \vdash \Diamond_u\, var \Downarrow_s v_s}\ \textsc{Unop} \qquad \frac{var_1, var_2 \in \Gamma \quad v_s = \Gamma[var_2]\ \Diamond_b\ \Gamma[var_2]}{\Gamma \vdash var_1\ \Diamond_b\ var_2 \Downarrow_s v_s}\ \textsc{Binop}$$

Note that our evaluation operator $\Downarrow_s$ does not need to be recursively defined because our language definition has flat expressions ($QF\_ABV_{||^e_d \leq 1}$ as shown in Table 2.3); intermediate values are stored in variables. This is intentional and will be used to bound the cost of symbolically evaluating a statement (Chapter 3). Extending the operator to recursive expressions is straightforward.

Concrete vs Symbolic Values. Each program variable in Γ is either concrete, when it is mapped to a value (or value_μ) instance, or symbolic (every other expression). The semantics of the operators above (e.g., $\Diamond_b$) are intentionally left open and are typically overloaded to handle concrete and symbolic values differently. For instance, adding $x + y$ two concrete variables reduces to a concrete value, e.g., $\Gamma = \{x \to 17, y \to 25\}$ reduces to 42. In contrast, if either of the variables is symbolic the result remains symbolic[7].

Example 4 (Trace-Based Symbolic Execution.)**.** *Consider the example below, (where input* : bv_{32}*). Given a trace taking the false branch, we show the contents of the symbolic execution*

[7]The observant reader may notice that the rule "an expression is symbolic if any part of it is symbolic" is very similar to a taint analysis policy [44]. This is not a coincidence, and symbolic execution can be reduced to certain types of taint analysis (the reduction is left as an exercise to the reader).

state during each step of the execution:

	Π	Γ	Γ^c
1 if $input == 42$ jump 3	$true$	$\{pc \to 1\}$	$\{pc \to 1, input \to 1\}$
2 $x := input + 17$	$\neg input = 42$	$\{pc \to 2\}$	$\{pc \to 2, input \to 1\}$
3 halt	$\neg input = 42$	$\{x \to input + 17, pc \to 3\}$	$\{x \to 18, pc \to 3, input \to 1\}$
	$\neg input = 42$	$\{x \to input + 17, pc \to 3\}$	Halted

Using the path predicate and the context we can check properties. For instance, we can check whether variable x can take the value 59 in the final state by checking isSat$(\neg input = 42 \implies input + 17 = 59)$*, which is always unsatisfiable. Thus, x can never take the value 59 along this path (for any trace).*

2.3.2 Multi-Path Symbolic Execution

Symbolic execution checks programs by systematically enumerating program paths. At every branch point, symbolic execution checks the feasibility of following each branch, and "forks" a fresh executor to explore each branch target. Figure 2.3 shows the operational semantics of multi-path symbolic execution. The state is again the union of three types: 1) Halted when all program paths have been explored and checked, 2) Error when a path with an error was found, and 3) a set of states S (with tuple elements (Π, Γ)) representing paths remaining to explore.

The pickNext() function models a priority queue and selects the next state to explore when given a set of active states [26][8]. isSat() checks the satisfiability of a logical formula. values(f, v) returns all possible values of v when formula f is true (when v is a constant only one value is possible). The straightforward implementation is based on repeated querying.

[8]Prioritizing states is an active area of research [21, 2, 45], and will be discussed at length in follow-up chapters (**??**).

$$\frac{S', (\Pi, \Gamma) = \mathsf{pickNext}(S) \quad \Gamma[pc] \in P \quad var := e = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow_s v_s \quad \Gamma' = \Gamma[var \to v_s]}{S \rightsquigarrow S' \cup \{(\Pi, \Gamma'[pc \to \Gamma[pc] + 1])\}} \textsc{Assign}$$

$$\frac{S', (\Pi, \Gamma) = \mathsf{pickNext}(S) \quad \Gamma[pc] \in P \quad \mathsf{assert}\ e = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow_s v_s \quad \mathsf{isValid}(\Pi \implies v_s) \quad \Pi' = \Pi \wedge v_s}{S \rightsquigarrow S' \cup \{(\Pi', \Gamma[pc \to \Gamma[pc] + 1])\}} \textsc{Assert}$$

$$\frac{S', (\Pi, \Gamma) = \mathsf{pickNext}(S) \quad \Gamma[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow_s v_s \quad \Gamma \vdash e \Downarrow_s v_{1s} \quad \mathsf{isValid}(\Pi \implies v_s) \quad \forall v_i \in \mathsf{values}(\Pi \wedge v_s, v_{1s}) : \Pi'_i = \Pi \wedge v_s \wedge v_{1s} = v_i}{S \rightsquigarrow S' \cup \{(\Pi'_1, \Gamma, v_1), \ldots, (\Pi'_k, \Gamma, v_k)\}} \textsc{TCond}$$

$$\frac{S', (\Pi, \Gamma) = \mathsf{pickNext}(S) \quad \Gamma[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma[pc]) \quad \mathsf{isValid}(\Pi \implies \neg v_s) \quad \Gamma \vdash e \Downarrow_s v_s \quad \Pi' = \Pi \wedge \neg v_s}{S \rightsquigarrow S \cup \{(\Pi', \Gamma[pc \to \Gamma[pc] + 1])\}} \textsc{FCond}$$

$$\frac{S', (\Pi, \Gamma) = \mathsf{pickNext}(S) \quad \Gamma[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma[pc]) \quad \Gamma \vdash e \Downarrow_s v_s \quad \Gamma \vdash e \Downarrow_s v_{1s} \quad \mathsf{isSat}(\Pi \wedge v_s) \quad \mathsf{isSat}(\Pi \wedge \neg v_s) \quad \Pi'_0 = \Pi \wedge \neg v_s \quad \forall v_i \in \mathsf{values}(\Pi \wedge v_s, v_{1s}) : \Pi'_i = \Pi \wedge v_s \wedge v_{1s} = v_i}{S \rightsquigarrow S' \cup \{(\Pi'_0, \Gamma[pc \to \Gamma[pc] + 1]), (\Pi'_1, \Gamma, v_1), \ldots, (\Pi'_k, \Gamma, v_k)\}} \textsc{ForkCond}$$

$$\frac{S', (\Pi, \Gamma) = \mathsf{pickNext}(S) \quad \Gamma[pc] \in P \quad \mathsf{halt} = \mathsf{instFetch}(P, \Gamma[pc])}{S \rightsquigarrow S'} \textsc{Halt}$$

$$\frac{S = \emptyset}{S \rightsquigarrow \mathsf{Halted}} \textsc{Finish}$$

Figure 2.3: Symbolic execution operational semantics for the language of Table 2.1.

Fist, we find a satisfying assignment to f, which gives value v_1 to v. Next we query $f \wedge v = v_1$ for satisfiability, getting value v_2. The process continues until the formula is no longer satisfiable, the values $v_1, \ldots, v_n$ gathered is the result of $\mathsf{values}(f, v)$.

Our semantics formulation of symbolic execution diverges slightly from the standard ("parallel tree") construction for symbolic execution. For example, a typical alternative formulation for the ForkCond rule is:

$$\Gamma[pc] \in P \quad \mathsf{if}\ e\ \mathsf{jump}\ e_1 = \mathsf{instFetch}(P, \Gamma[pc])$$

$$\Gamma \vdash e \Downarrow_s v_s \quad \Gamma \vdash e \Downarrow_s v_{1s} \quad \mathsf{isSat}(\Pi \wedge v_s) \quad \mathsf{isSat}(\Pi \wedge \neg v_s)$$

$$\Pi \wedge \neg v_s, \Gamma[pc \ \rightarrow \ pc + 1] \ \rightsquigarrow \ \mathsf{Halted}$$

$$\frac{\forall v_i \in \mathsf{values}(\Pi \wedge v_s, v_{1s}) : \ \Pi \wedge v_s \wedge v_{1s} = v_i, \Gamma[pc \rightarrow v_i] \rightsquigarrow \mathsf{Halted}}{\Pi, \Gamma \rightsquigarrow \mathsf{Halted}} \ \textsc{ForkCond}$$

In this formulation the rules recurse over the program structure while following the program execution, and rule applications form a tree structure. While useful for proofs because of its simplicity, the above formulation does not explicitly convey the notion of a priority queue (one needs to be added on top). The formulation in Figure 2.3 makes the search decisions and priority explicit in the rules (via pickNext()).

Further, the semantics shown in Figure 2.3 ensure that all statements are executed sequentially, i.e., there is no parallelism. Symbolic execution is a highly parallelizable task [46, 47], and modifying the semantics to account for bounded (up to n cores) or full parallelism (arbitrary number of cores) is straightforward, but may complicate notation and follow-up cost analyses (Chapter 3). In this thesis, we focus on analyzing sequential symbolic execution (one execution step at a time), and discussion on parallelism will be limited to hints for extending our framework to support parallelism.

Using the semantics of the symbolic execution operator ($\rightsquigarrow$) we can define a relation between execution states, to denote whether is is feasible to transition from one state to another in a single execution step.

Definition 16 (Execution Relation.)**.** *Two execution states s_1, s_2 belong in the execution relation of a program P, denoted by $s_1 \rightarrow_P s_2$, when $\exists S, S' : s_1 \in S \wedge s_2 \in S' \wedge s_2 \notin S \wedge S \rightsquigarrow S'$.*

Using the execution relation we can define when a state is reachable:

Definition 17 (Reachability.)**.** *State s_2 is reachable from s_1 for program P, denoted by $s_1 \rightarrow_P^* s_2$, when (s_1, s_2) is in the transitive closure of the execution relation.*

State (or Path) Explosion. The ForkCond rule highlights one of the main challenges in symbolic execution. Given a single execution state and a conditional jump statement where both branches are satisfiable, symbolic execution needs to "fork" a separate execution state for each branch. Thus, the number of states to explore may grow exponentially in the size of the program, a problem known as *state (or path) explosion*. Note the problem is exacerbated when the jump target is symbolic (in $\mathsf{if}\ e\ \mathsf{jump}\ e_1$, e_1 may be derived from user input), since symbolic execution needs to fork an extra state for every possible target. This behavior is typical in low-level assembly-like languages where control flow is often encoded with jump tables, e.g., `gcc` translates the `switch` statement in C to a jump table.

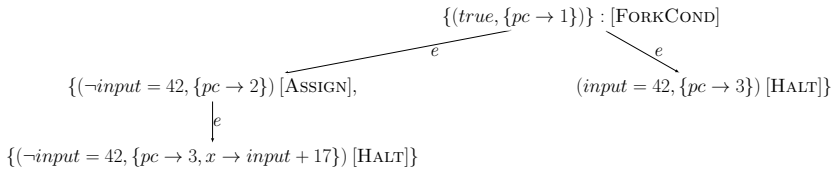
Example 5 (Symbolic Execution States.)**.** *Consider the following example ($input : bv_{32}$):*

1 $\mathsf{if}\ input == 42\ \mathsf{jump}\ 3$

2 $x := input + 17$

3 halt

A symbolic execution of the program above, with $\Pi = true$, $\Gamma = \{pc \rightarrow 1\}$ is (we only show the progression of resulting states after applying the rules from Figure 2.3):



The conditional statement "forks" two new states with ForkCond*: the first one (the true branch) completes after applying the* Halt *rule, while the second after applying an* Assign *followed by a* Halt *rule. The arrows signify the execution relation between states, i.e., the*

*progression of each state as the symbolic execution rules apply to the state set. Note that in the example we did not specify a scheduling order (*pickNext()*).*

Symbolic Execution Tree. The structure in the example above, visualizing the execution relation (the transition from each state to one or more new states), forms the *symbolic execution tree* [16]. Every node in the tree corresponds to an execution state, while edges to execution steps. Branching nodes correspond to conditional jumps, where execution can follow either one of the branches—note that if only one of the branches is satisfiable (as in the FCOND rule), there is no branching. Leaf nodes correspond to distinct execution path that are explored to completion (ending either in a Halted or Error). Similar to single path symbolic execution, a leaf node ending in a Halted means that all possible traces of the path are correct (e.g., no assertion failure).

Definition 18 (Symbolic Execution Tree.)**.** *Given an initial execution state s_{init} (root) and a program P, the symbolic execution tree $\mathcal{T}_P(s_{init})$ is a graph $G(V, E)$ where:*

- *Each node is an execution state reachable from the root: $\forall s \in V : s_{init} \rightarrow_P^* s$.*
- *Each edge corresponds to an execution relation: $\forall (s_1, s_2) \in E : s_1 \rightarrow_P s_2$.*

The size of a symbolic execution tree, denoted by $|\mathcal{T}_P(s_{init})|_t$, is the number of nodes $|V|$.

The symbolic execution tree is derived from the operational semantics of symbolic execution; each node corresponds to an execution step. Thus, the size of the symbolic execution tree represents the total number of steps symbolic execution needs to take in order to complete exploration for a program.

Timeouts. None of the algorithms we have presented so far handles non-termination. A single infinite loop will cause our analysis to not terminate, since the symbolic execution tree will be infinitely deep (Note, that the fan-out of the tree will be bounded, since we only consider bounded finite inputs). Despite existing work in detecting certain types of infinite

loops at runtime [48], practical symbolic execution analyses rely on hard timeouts to ensure termination. Modeling timeouts in our semantics is straightforward, it simply requires the addition of a *time* variable and ensuring in each step of the execution that *time* did not exceed a threshold. Similarly, our symbolic execution tree will be bounded, when a timeout applies.

Definition 19 (Verification Condition (VC).)**.** *Given a program P and an input domain Σ^n, a verification condition (VC) is a logical formula, denoted by $VC(P, \Sigma^n)$, where:*

$$valid(VC(P, \Sigma^n)) \Leftrightarrow P \text{ is correct}$$

For example, to generate a VC with symbolic execution, we compute the disjunction of all path predicates generated during the exploration of all program paths.

Thus, the check $\mathsf{isValid}(\Pi \implies v_s)$ ensures at every assertion that under the current path predicate (Π), it is impossible to falsify the assertion argument (v_s). If that is not the case, symbolic execution will abort with Error. Thus, a successful termination (Halted) with symbolic execution on a specific path indicates the path is correct.

2.4 Macroscopic View of Symbolic Execution

Section 2.1 introduced our language and the concrete execution semantics, Section 2.2 added the concept of programs and traces, while Section 2.3 presented the symbolic execution semantics, states, and the mapping to logical formulas. Before closing this introductory chapter, we give a high-level view of symbolic execution, and how all previous elements are interconnected. Figure 5.4 provides an overview.

Inputs to Paths. Every concrete input ι maps to an path via concrete execution (and the trace semantics $\pi_{tr(P,\iota)}$); for feasible paths, the mapping is surjective (onto). Multiple inputs may map to the same path, thus making the path space smaller ($|\mathcal{F}(P, \Sigma^n)| \leq |\Sigma^n|$). The

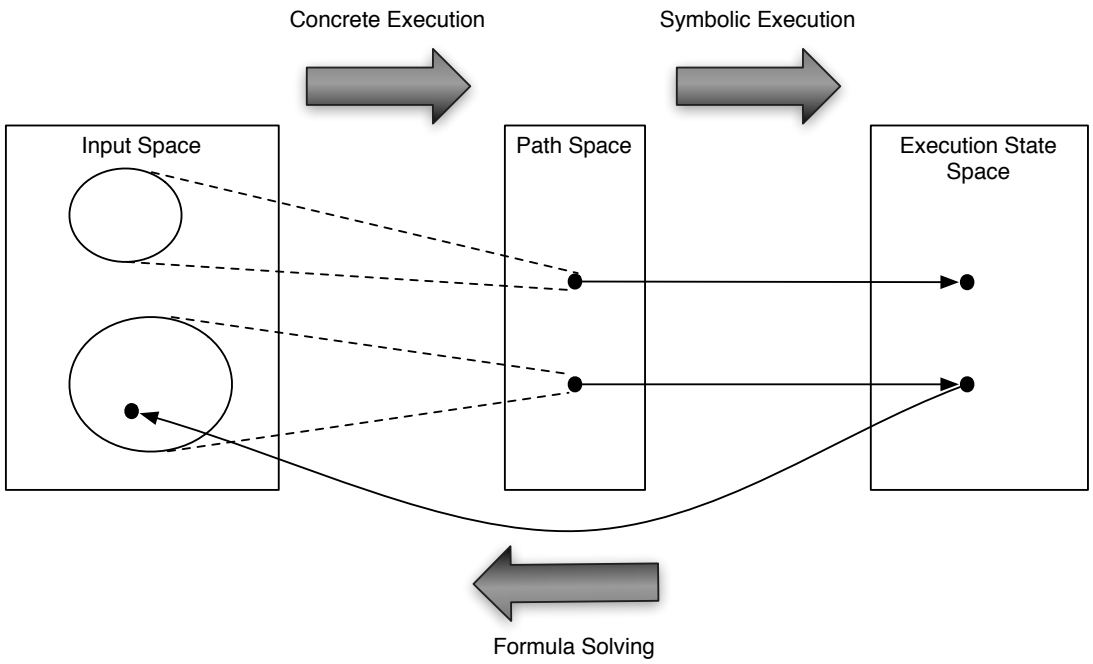


Figure 2.4: Inputs, paths, execution states and their connections.

potentially smaller path space is one of the reasons symbolic execution is expected to be better than enumerating inputs for testing.

Paths to Execution States. Given a starting execution state s_{init}, a path π can be mapped to a symbolic execution state (Π, Γ), via the symbolic execution semantics $s_{init} \rightarrow_P^* (\Pi, \Gamma)$. Every path corresponds to a single execution state, and two paths cannot map to the same state, i.e., the mapping is injective (one-to-one)—this is true for analyses that do not traverse multiple paths simultaneously (see Chapter 7).

Execution States to Inputs. By finding a satisfying assignment to the path predicate of a symbolic execution state, we get an element of the input space that, based on the semantics of symbolic execution, would generate a trace resulting in the same execution state. Note that a single state may correspond to multiple inputs.

The process of reasoning with symbolic execution is split in two clear steps: a forward operation converting statements to formulas, and an inversion operation finding inputs that satisfy the generated formulas.

Chapter 3

The Cost of Symbolic Execution

Your laptop is just a DFA.

— Yannis Mallios, *Dinner.*

Symbolic executor implementations are concrete interpreter instances of the symbolic execution semantics we presented in Chapter 2. By interpreting the semantics of program statements, symbolic executors can check correctness and identify possible flaws. In this chapter, we discuss the cost, in terms of time, of performing symbolic execution. We document the main trade-offs, and present current approaches for gaining scalability. We start by presenting a taxonomy of the cost (Section 3.1), then present a detailed component breakdown of concrete symbolic executor instances (Section 3.2), and the section ends with an example instance (Section 3.3)

3.1 Symbolic Execution Cost

In this section, we express the cost of the various symbolic execution components at a higher level of granularity. Building from the ground up, we start with instructions (Section 3.1.1), move on to paths (Section 3.1.2), and finally to programs (Section 3.1.3).

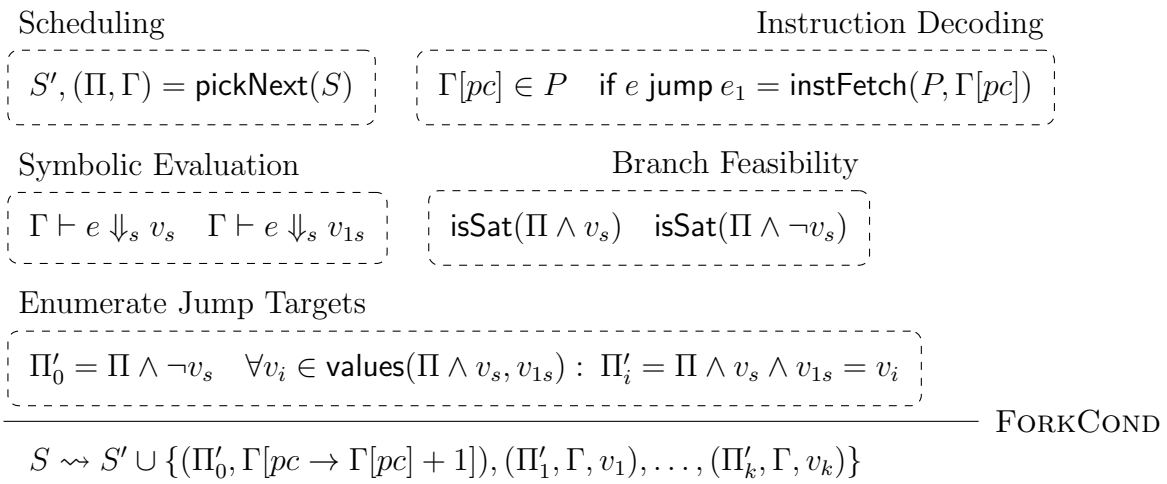


Figure 3.1: The ForkCond rule, and the component associated with each premise.

3.1.1 Instruction Level

Concretely executing an instruction in a modern system comes with a cost. The CPU, number of cores, caches, hardware design and architecture decisions determine performance. Similarly, executing an instruction (statement[1]) symbolically is also associated with a cost. In this section, we investigate the cost of executing a single instruction under a certain execution context.

As a first step, we classify the cost of symbolically executing an instruction based on the component where time is spent on. To do so, we dissect the operational semantics of a single symbolic execution rule (step). For example, Figure 3.1 breaks down the cost of applying the ForkCond rule[2], when evaluating a conditional jump instruction. We briefly describe each of the costs.

Scheduling & Context Switching. Before evaluating a statement, symbolic execution needs to determine the next state to execute among the set of pending states. There is a

[1] We use the terms instruction and statement indistinguishably.

[2] Arguably the most complex rule of symbolic execution, ForkCond utilizes all typical symbolic execution components.

wealth of work investigating strategies for selecting the next state to explore in the symbolic execution tree. Depth-first search, breadth-first search, generational search [12], and most graph-based algorithms are directly applicable in state scheduling. Pending states are typically organized in priority queues, and fetching the best element has constant ($\mathrm{O}(1)$) or logarithmic ($\mathrm{O}(\log n)$) in the number of pending states (n) cost; linear or higher-complexity algorithms are usually unacceptable since the number of states is very large (state explosion).

In practical implementations, selecting a state has an extra cost: context switching between states. After selecting a state with a context Γ, the symbolic executor must get access to the contents of all variables including memory, a process that may not be instantaneous. For example, concolic executors [13] only change states upon path completion and requires spawning a new process; online executors that use SMT solvers with incremental solving need to restore the state of the memory or solver [3, 29]. An implementation needs to be fully immutable (a possible performance headache) to ensure fast context switching.

Instruction Decoding Popular architectures, such as x86, ARM, etc. have a fixed number of instructions and possible encodings. Thus, fetching an instruction from memory and decoding it has a fixed upper bound in terms of cost ($\mathrm{O}(1)$).

Symbolic Evaluation. During statement evaluation, symbolic execution computes the program values that will be used by the context and path predicate. Computing these values requires evaluating ($\Downarrow_s$) the expressions of the statement under the current context, an operation that is linear in the size the expression ($||^e_s$). Thus, for languages with bounded expressions within statements (e.g., $BIL(QF\_BV_{||^e_s \leq 1})$)—a realistic assumption for popular low-level assembly-like languages—the cost of evaluating a single statement can also be bounded by a constant.

Despite the constant upper bound, symbolic evaluation represents a significant amount of the time spent in symbolic execution; for example, 37% of the MAYHEM symbolic executor is spent evaluating statements [3]. Using more lightweight techniques to identify instructions

that are irrelevant to symbolic reasoning and avoiding to execute them symbolically, e.g., taint analysis, are very common [3, 49, 32].

Also note, that more elaborate symbolic expression evaluation schemes ($\Downarrow_s$) may incur higher than linear cost. For example, certain simplifications may require repeated transformations of the initial expressions, leading to quadratic complexity or worse. In this section, we restrict discussion to evaluation functions that are linear in the size of the expression.

Branch Feasibility & Enumerate Jump Targets. During branch feasibility, symbolic execution uses an SMT solver for determining whether the branch condition can be both *true* and $false$. During jump target enumeration, the SMT is utilized to identify all possible values an expression may take. In both cases, the cost of the queries performed (or the number of queries) is not dependent solely on the statement evaluated. The statements in the path that precede the current instruction can also affect the queries. We will discuss entire paths in a follow-up section (Section 3.1.3).

Note that performing more than one query per instruction is not restricted to computed jumps; symbolic executors may choose to query and fork new states even in straightline code. For example, MAYHEM uses the solver to determine the range of symbolic pointers [3]. If the range exceeds a certain threshold multiple executors are forked for different memory regions; S2E [29] follows a similar approach.

Instruction Cost Formula. Based on the FORKCOND example, cost can be broken down into two main categories:

1. **Instrumentation**. The instrumentation cost includes the cost to select a state (Scheduling & Context Switching), fetch the current instruction (Instruction Decoding), and evaluate it (Symbolic Evaluation). All instructions come with an instrumentation cost, and modern symbolic executor implementations invest heavily in minimizing it; we will discuss several optimization techniques in follow-up chapters (Chapters 6 and 7).

Instrumentation cost is typically predictable, i.e., we have tight performance bounds for the algorithms used during instrumentation.

2. **SMT Reasoning**. Symbolic execution uses SMT solvers to answer queries (reason) about the current execution path, including whether alternative paths can be taken (Branch Feasibility), and finding all possible targets of a jump (Enumerate Jump Targets). Note that not all instructions require reasoning, e.g., evaluating assignments typically does not involve SMT queries. Optimization attempts focus on two challenges: minimizing the number of queries performed, and minimizing the time spent resolving the queries. Unlike instrumentation cost, reasoning cost is currently unpredictable, i.e., we do not have tight performance bounds, since reasoning reduces to an NP-hard problem.

The cost of symbolically executing an instruction (statement) i, selected from a set of pending states S can be written as:

$$\mathcal{C}_I^S(i) = \mathcal{I}_I^S(i) + \mathcal{R}_I^S(i) \tag{3.1}$$

where $\mathcal{I}_I^S(i)$ is the instrumentation cost, and $\mathcal{R}_I^S(i)$ is the reasoning cost of instruction i. The instrumentation cost, includes the scheduling $\mathcal{S}_I^S(i)$ and instruction evaluation cost $\mathcal{E}_I^S(i)$, or more simply:

$$\mathcal{I}_I^S(i) = \mathcal{S}_I^S(i) + \mathcal{E}_I^S(i) \tag{3.2}$$

The reasoning cost is the total cost of all queries performed:

$$\mathcal{R}_I^S(i) = \sum_j^{\mathcal{N}_I^S(i)} \mathcal{Q}_{I,j}^S(i) \tag{3.3}$$

where $\mathcal{N}_I^S(i)$ is the number of queries performed, and $\mathcal{Q}_{I,j}^S(i)$ is the cost of resolving the j^{th} query of instruction i. The next sections explore how these costs change when moving from single instructions to paths, and full programs.

3.1.2 Path Level

Trace-based symbolic execution processes a single program path, one instruction at a time, as specified by a trace (Section 2.3.1). Given a program path π and a starting execution state s_{init}, the cost of symbolically executing the path to completion is:

$$\mathcal{C}_p^{\{s_{init}\}}(\pi) = \sum_{i \in \pi} \mathcal{C}_I^{\{s_i\}}(i) \tag{3.4}$$

which adds up the costs of executing each instruction under a different execution context s_i. The cost for symbolically executing a path specified by a trace $\pi_{tr(P,\iota)}$ follows the same formula. Categorizing the path cost to components is straightforward; notation follows the same convention as with instruction cost. The instrumentation cost and reasoning for the path will be respectively:

$$\mathcal{I}_p^S(\pi) = \sum_{i \in \pi} \mathcal{I}_I^{S_i}(i) \quad \text{and} \quad \mathcal{R}_p^S(\pi) = \sum_{i \in \pi} \mathcal{R}_I^{S_i}(i) \tag{3.5}$$

Similar to instructions, the costs can be further broken down to scheduling ($\mathcal{S}_p^S(\pi)$), evaluation ($\mathcal{E}_p^S(\pi)$), queries ($\mathcal{N}_p^S(\pi)$, $\mathcal{Q}_{p,j}^S(\pi)$), etc. Note that for trace-based symbolic execution, there is really no scheduling—$\mathcal{S}_I^S(i)$ is constant time—and can be considered part of instruction evaluation (decoding). Finally, note that the above formula is meaningful only for finite or truncated traces; a non-terminating trace will never terminate and will have infinite cost.

3.1.3 Program Level

Given a program P and input space Σ^n, the set of feasible paths is $\mathcal{F}(P, \Sigma^n)$ (Chapter 2). Thus, the cost of symbolically executing all possible program paths from an initial execution state s_{init} with a symbolic bitvector input $\iota : bv_n$ (denoted as $s_{init}(\iota)$) can be written as:

$$\mathcal{C}^{\{s_{init}\}}(P) = \sum_{\pi \in \mathcal{F}\left(P, \Sigma^{|s_{init}(\iota)|_b}\right)} \mathcal{C}_p^S(\pi) \tag{3.6}$$

The cost formula above assumes that symbolic execution explores one path at a time, starting execution from the beginning of the program every time. While this assumption is true for certain trace-based (offline) symbolic executors [12, 3], it is not true for other implementations [21, 29]. The semantics of symbolic execution we provided (Chapter 2) show the tree-based exploration, where after forking a state, execution may continue from the last statement (instead of starting the path from the beginning of the program).

Thus, the alternative encoding of the cost is based on the symbolic execution tree. Symbolically executing the program corresponds to the total cost of executing all nodes in the tree from the starting root state s_{init}:

$$\mathcal{C}^{\{s_{init}\}}(P) = \sum_{i \in \mathcal{T}_P(s_{init})} \mathcal{C}_I^{S_i}(i) \tag{3.7}$$

where S_i is the set of states during the execution of the i^{th} instruction in the tree. The above expression, completes our series of basic cost formulas for symbolic execution. In the next section, we present concrete instances of the above formulas, as found in modern implementations.

3.2 Component Breakdown & Tradeoffs

Section 3.1 presented a series of formulas for describing the cost of symbolic execution. This section discusses each cost term presented above individually; we will discuss example combinations in the next section (Section 3.3). We start by discussing instruction evaluation (Section 3.2.1) and scheduling (Section 3.2.2), and then we move to the number and cost of queries (Section 3.2.3).

3.2.1 Intruction Evaluation

Evaluating an instruction (or statement) requires: 1) a constant cost, e.g., fetching the program statement[3], and 2) a potentially variable cost which includes the evaluation of all expressions in the statement and updating the execution state. Below, we discuss the various options while evaluating a statement.

3.2.1.1 Language & Expressivity

The BIL definition in Chapter 2 is close to 3-address code with flat expressions $QF\_ABV_{||_d^e \leq 1}$, thus allowing us to bound the cost of evaluating statement expressions by a small constant (e.g., three expression lookups and one expression creation while evaluating a $\mathsf{store}(e_1, e_2, e_3)$). Note that a different language choice would have an entirely different complexity, e.g., $BIL(QF\_ABV)$ with a recursive $\Downarrow_s$ would be $\mathrm{O}\left(|e|_s^e\right)$, where e is the largest expression used in the instruction.

Similarly, updating the context—for example, during an assignment—can also be constant time if implemented as a hashtable. Thus, it is possible to have a $\mathrm{O}\left(1\right)$ implementation for

[3]We only consider typical finite instruction sets; languages with non-constant instruction lookups are left as out of scope.

performing instruction evaluation in BIL. However, there is a number of considerations in real implementations:

Big Constants & Loops. Instructions in current architectures typically compute finite expressions, and thus their cost can be bounded by a constant. Unfortunately, these constants can be non-negligible. For example, the `aesenc` instruction in x86 computes a single round of AES encryption, hardly a trivial operation. In a trace with millions or billions of instructions, the cost of symbolic evaluation can quickly become the bottleneck, despite a "constant" instruction execution cost. Because concrete execution is usually much faster than symbolic evaluation, multiple techniques have been used to avoid symbolically executing uninteresting instructions. Examples include taint analysis [44], which is routinely used in combination with symbolic execution [12, 3], and selective symbolic execution [29].

Instructions may also have internal loops, e.g., instructions with a `rep` prefix on x86 are described in the manual with a `while` loop. To represent such instructions atomically, we would need a more expressive language; for instance, by allowing recursion at the level of expressions. By choosing to perform our analysis on BIL, we relinquish the ability to analyze such instructions atomically. Instead, such instructions are desugared down to a lower level representation (BIL). For example, instructions with a `rep` prefix are broken down to a loop with multiple instructions. The upside is we still analyze a simple language for the analysis; the downside is we are blowing up the instruction to a potentially less concise representation (by a factor of at least $2\times$ on our experiments on x86).

Simplifications. The expression evaluation operator $\Downarrow_s$ we presented in Chapter 2 performs variable substitution based on the execution context. However, modern symbolic executors do not perform just substitution. Term rewriting using algebraic simplifications (e.g., $x \oplus x = 0$), and expression normalization (e.g., nesting associative binary operators in a specific order—$(a + b) + c$) are very common [21, 12, 3, 4]. Simplifications improve performance in two ways: 1) they shrink the size of symbolic expressions (or even eliminate them completely as with

the $\oplus$ example above), thus making evaluation faster, and 2) the formulas passed to the solver are smaller/simpler and thus faster to solve.

As a concrete example, we mention one of the transformations applied by the MAYHEM [3] symbolic executor, and the rationale behind it. A formula requiring more than 30 seconds to solve is a rare occurrence in MAYHEM's benchmark suite (only 1 in approximately 10 million formulas exceeds 30 seconds, see more statistics in Section 3.2.3). Nevertheless, we found that half of those formulas, had the following common subexpression added:

```
extract:63:32[ x * 0xccccccccd ] >> 3
```

The expression above is equivalent to $x/10$. After program inspection, we noticed that this is a well-known compiler optimization [50], that uses multiplication instead of division to speed up concrete execution. Adding a simplification that reduces the above term to a division ($x/10$) reduced solving time to milliseconds (using the Z3 SMT solver).

Simplification rules are typically hard-coded or added in an ad hoc manner to resolve performance issues (even in SMT solvers [51]). While there are principled ways to check such simplification rules for correctness (simply by checking equivalence before and after the transformation), up until recently there was no work on generating them automatically. Romano *et al.* recently proposed a technique based for automatically generating reduction rules [52]. We believe this is an interesting avenue for future work (Chapter 8).

Simplification rules are applied using a visitor and pattern matching on the expression in a linear pass. However, if the rewriting engine is recursive, a single simplification may trigger more simplifications in the expression, leading potentially to quadratic costs (or worse, depending on the implementation). Also note, that the size of the expression may be linear in the size of the trace $\mathrm{O}(|\pi|)$—up to the current instruction (assuming that expressions are memoized/hash-consed [5] and reused during substitution—Chapter 7). Thus, the cost of simplifying an expression during the execution of an instruction may depend on previously

executed instructions. Worse, performance may degrade as the execution path and the symbolic expressions become deeper. This is one of the reasons simplification rules are heavily memoized (Chapter 7), and most executors apply slicing.

3.2.1.2 Slicing & the Purpose of the Context

Symbolic executors operate on an execution context Γ, mapping each variable to a value. A simpler alternative would be to keep track of a single variable, the machine state, and every statement would operate on that directly (similar to $BIL_{\mathcal{M}}(QF\_ABV)$). Why keep track of a variable context[4]?

The reason is slicing [53], and specifically dynamic slicing [54]. By keeping track of the value of each variable separately, symbolic execution is implicitly slicing the current execution path for each variable in the context. Instructions that do not operate on a variable have no effect on the variable's value. For example, we may execute millions of instructions drawing a GUI on the screen and not modify the variable holding our input. Had we executed the same program without the GUI instructions (the slice), the value of the variable would be identical.

Thus, the context augmented with the path predicate provide us with a path-sensitive slice of the execution for each variable. The slicing effect allows us to ignore large parts of the program and symbolically execute only the parts that matter, leading to significant speedups in practice (of course, in the worst case the slice is the entire path for all variables). Components that benefit from the presence of the context include the expression evaluation cost (expressions are smaller), the instruction execution cost (taint analysis is meaningful), and solver cost (formulas capture only part of the path, thus are smaller).

[4]Perhaps the answer is immediately obvious to the reader, but it was not to the author.

3.2.1.3 Summary

Real Implementations. The cost of symbolically evaluating instructions is not negligible in practice. For example, the MAYHEM executor spends up to 80.8% of its time during instruction evaluation when running in concolic mode [3]. We will see the part of this cost related to concrete execution can be mitigated by using different scheduling algorithms. Similarly, slicing and the ability to use taint analysis to only symbolically execute instructions that operate on symbolic data reduces the number of instructions by up to 4 orders of magnitude (Chapter 6).

Time & Space Cost. The time cost of executing a single instruction at the end of a path π ($\mathcal{E}_I^S(i)$) varies from $\mathrm{O}(1)$ to $\mathrm{O}(|\pi|)$ or higher depending on the implementation. For BIL statements, the size of the expression that is added to the context per instruction is bounded and thus is constant $\mathrm{O}(1)$. Extrapolating from instructions to paths, time costs vary from $\mathrm{O}(|\pi|)$ to $\mathrm{O}\left(|\pi|^2\right)$ or higher and the context size is linear $\mathrm{O}(|\pi|)$.

3.2.2 Scheduling & Path Selection

Every branching instruction in the program, potentially doubles the number of states that need to be explored. For complex programs, symbolic execution cannot possibly explore all of them in a reasonable amount of time, and thus only part of the state space can be explored. Among a huge number of states, which one should be explored next? This is a scheduling problem and is known in the literature as the *path selection* or *prioritization* problem.

There is no generally accepted solution to path selection (pickNext() from Chapter 2). Different programs behave differently, and identifying which states need to be explored for an arbitrary program is hard. Nevertheless, finding well-tuned heuristics that work well for specific domains is an active area of research. In the sections below, we present popular

search heuristics (Section 3.2.2.1), and the two main techniques for context switching between states (Section 3.2.2.2).

3.2.2.1 Search Heuristics

We use the term search heuristics or strategies to describe algorithms for exploring the symbolic execution tree. We split strategies into two main categories: graph-based, and goal/domain specific.

Graph-Based Strategies. Graph-based strategies depend solely on the structure of the symbolic execution tree. The algorithms are agnostic to the state of the program during exploration, and could be applied to pure graphs. Below, we present some representative algorithms:

- **Depth- & Breadth-First Search.** Using the standard graph-based algorithms, DFS and BFS explore the symbolic execution tree as expected. The advantages of DFS are: 1) states deeper in the tree (and thus potentially the program) are explored first, and 2) the number of pending states at any time is equal to the number of branches in the current path (small memory footprint). The primary disadvantage is that DFS can get stuck in non-terminating loops or very specific parts of the code—we refer to this issue as the *locality* problem. The advantage of BFS is that it is exploring all paths at the same depth in a round-robin fashion, potentially obtaining quick code coverage at startup. The downside is that the number of states grows very quickly, program exploration does not favor deeper paths, and path completion may take very long[5]—since all paths are explored simultaneously.

  DFS is usually implemented with a configurable maximum depth parameter to avoid exploring at arbitrary depth and mitigate locality. Most symbolic executors come with

[5]These disadvantages make BFS a poor strategy for several applications, e.g., when deep exploration is required.

DFS and BFS built-in, and serve as a baseline for comparing with other strategies [21, 19].

- **Random Search.** Selecting a state at random is a simple strategy to implement. By modifying the probability of selecting each state, symbolic execution can favor different states, e.g., by giving higher weight to states that are higher in the tree. The randomness ensures that exploration can escape from locality. State-of-the-art executors such as KLEE [21] come with a random path selection strategy. An advantage of random search is that it is easily composable with other search strategies.

- **Concolic Testing.** Concolic testing [20, 30] uses a concrete input, also called a *seed* input, to generate a trace of a program execution ($tr(P, \iota)$). Using the initial execution path in the tree, the analysis proceeds by choosing potentially branching nodes in the path, negating the conditional expression, and expanding exploration from there. Unfortunately, such explorations suffer from locality, since the initial seed determines the neighborhood of the state space that will be explored. Hybrid concolic testing [55] is a variation of concolic testing combined with random search to mitigate locality. Instead of constantly searching with the same starting seed, search is restarted with a new random seed multiple times to explore new states.

- **Generational Search.** A special type of concolic testing, generational search [12] expands the *front* of the symbolic execution tree in a BFS manner starting from the initial concrete path (instead of starting from the root of the tree). The number of conditional branches followed after diverging from the original path determines the generation of the state (the initial path has a generation of 0). The initial seed ensures that deeper states are reached (the main disadvantage of BFS), while the BFS expansion weighs similarly shallow and deep states, making progress in both fronts. By generating concrete inputs for all feasible alternate conditions of a trace, generational search can

generate thousands of concrete inputs that cover different execution paths in a single symbolic execution.

Goal & Domain Specific Strategies. Goal and domain specific strategies can observe the program and the execution state of the program, and adjust scheduling towards a specific goal. Typical goals include maximizing code coverage, exploring subparts of the state space, guiding execution towards a line of code, etc. We briefly mention below specific strategies that have been used in previous (including ours) work:

- **Maximizing Coverage.** Symbolic execution is typically used for test case generation; generating test cases that achieve higher code coverage gives the analyst higher confidence about the correctness of the tested software. KLEE [21] was the first symbolic executor that demonstrated high code coverage on a diverse set of utilities written in C (`coreutils`). KLEE employed a number of heuristics to select states, including proximity to uncovered instructions. Symbolic executors on higher-level languages, such as PEX [56], also heavily utilize heuristics to maximize code coverage. For example, Xie *et al.* [57] introduced a fitness function measuring path proximity to a test target (e.g., uncovered instructions) to guide path selection.

- **Line Reachability.** Related to the problem of maximizing code coverage, Ma *et al.* [45] proposed the line reachability problem. The line reachability problem is: given a line in a program, find an input that drives program execution to that line. Using the interprocedural control flow (ICFG) graph[6] of the program, Ma *et al.* [45] developed path selection techniques to direct symbolic execution towards a specific line, based on proximity metrics. Earlier, Zamfir *et al.* [58] also used a proximity heuristic based

[6]Having an ICFG of the program is invaluable, since it allows using search algorithms from the artificial intelligence community at the graph level (e.g., A* and beyond). Unfortunately, for low-level languages—including assembly—a useful ICFG is often unavailable.

on the ICFG to direct symbolic execution towards a specific line of code. Note, that a practical solution to line reachability would provide a fundamental primitive for maximizing coverage.

- **Pruning the state space.** The strategies presented above assign values to prioritize execution states. A different line of research, is based on pruning (or reducing) states, which is semantically similar to giving very low priority to such states. Pruning examples include selective symbolic execution [29], which saves time by performing symbolic execution only on selected parts of the programm; RWSet [59], which drops states that cannot provide new code coverage; and preconditioned symbolic execution [2], which uses heuristic preconditions for finding buffer overflows. Preconditioned symbolic execution will be discussed further in Chapter 5.

The list of strategies presented above is not exhaustive; there is a wealth of work in the area that—in the interest of time and space—we did not mention. We refer the reader to a number of reviews for exploring search strategies further [60, 61, 25].

Heuristics. Path selection algorithms are based on heuristics; there is no guarantee about the behavior of these algorithms on arbitrary programs. However, a domain-specific fine-tuned heuristic can be useful, practical, and efficient on many concrete program instances; thus making search heuristics one of the (currently) most productive research areas in symbolic execution.

3.2.2.2 Offline and Online Symbolic Execution

In the symbolic execution semantics presented in Chapter 2, the selection of the next state to be executed via pickNext() seems straightforward: using a heuristic, we extract a state from a priority queue; a constant time operation (e.g., with fibonacci heaps) if the heuristic does not dynamically update state values on extraction. However, actual implementations

suffer from an extra cost: replacing the environment of the previously executed state with the environment of the newly selected state. Similar to threads or processes running on a CPU, we refer to this cost as the *context-switching* cost. Whether a context-switch is required and selecting the new state depends on the path selection strategy (discussed in Section 3.2.2.1).

In this section, we discuss the two most popular ways of context-switching between symbolic execution states: offline and online symbolic execution. Last, we briefly discuss hybrid symbolic execution, which combines the above two approaches.

Offline Execution. Offline symbolic executors [32, 12] execute a single path at a time. When the execution of the path completes (timeouts are used for non-termination), a new path is selected for execution. The process repeats until there are no more paths. The execution is called offline, because the concrete execution of the path is first recorded in a trace, and then the trace is executed symbolically *offline*—not during the actual execution—to generate new test inputs (trace-based execution). Since concrete inputs are driving the exploration of the program, offline executors are also concolic executors. We will use the term offline executor to refer to any executor processing a single path at a time, not only trace-based executors.

Offline execution is attractive because of its simplicity. At any time, the symbolic executor nees to handle the execution of a single path; no need to keep multiple states in memory and context-switch at the instruction level. The design simplicity of offline execution requires fewer resources such as memory (only one state needs to be kept), thus allowing symbolic execution to run on larger programs. Using offline execution, SAGE [12], a symbolic executor developed by Microsoft, is capable of running applications the size of MS Word [62].

The disadvantages of offline execution also stem from its simplicity. By executing every path individually, the shared prefix of different paths will be executed repeatedly. Conceptually, offline execution explores the symbolic execution tree of the program in paths starting from the root and ending at a leaf node, thus repeating the first higher level nodes. Figure 3.2

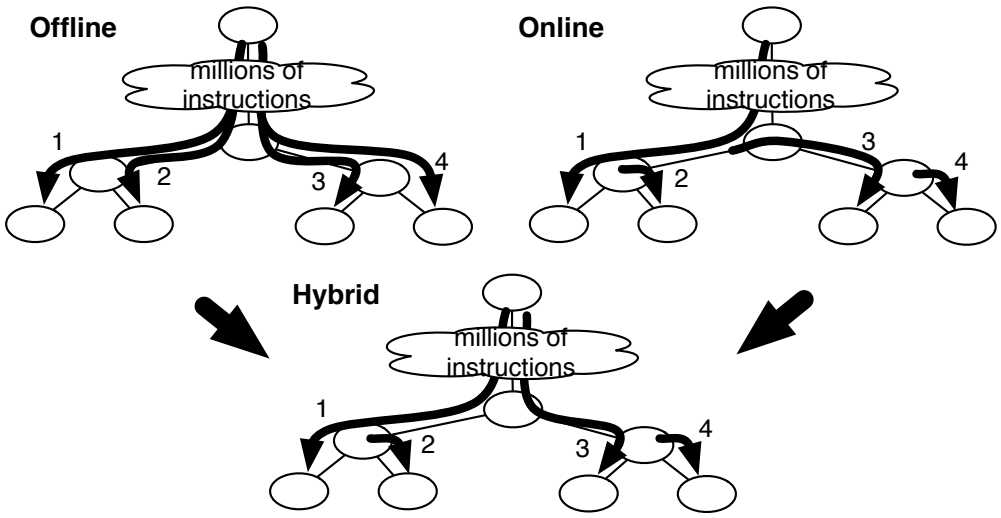


Figure 3.2: Hybrid execution combines the context-switching speed of online execution with the ability of offline execution to swap states to disk.

shows an example tree where the first nodes in the tree contain millions of instructions and thus repetition may become a problem.

For a symbolic execution tree of size $\mathrm{O}\left(|\mathcal{T}_P(\iota)|_t\right)$, the total number of instructions executed with offline execution is bounded by $\mathrm{O}\left(|\mathcal{T}_P(\iota)|_t^2\right)$—repeating every instruction (running a path) for every instruction encountered. We should mention that this is the worst case complexity, the total number of instructions executed depends on the structure of the tree and may be as low as $|\mathcal{T}_P(\iota)|_t$ (for a single path).

The extra cost of instruction re-execution is part of context-switching ($\mathcal{S}_I^S(i)$), the process required to restore the state where the last path forked execution. Other context-switching costs in offline execution include initialization steps, e.g., spawning a new process, which are considered constant for a path. However, this constant cost may be very high, especially when compared with the in-memory context switches of online symbolic execution (see next paragraph). The high-cost of context-switching is also the main reason offline execution explores paths to completion. More frequent switches would be prohibitively expensive.

Online Execution. Online symbolic executors [21, 29] keep all pending states in memory, and execution switches between them following a search strategy. The method is called online, because all decisions—including forking, scheduling, etc—are made as the program executes. Online symbolic execution explores each node in the symbolic execution tree once, thus making exploration $\mathrm{O}\left(|\mathcal{T}_P(\iota)|_t\right)$—much faster than offline execution.

Context-switching between states is implementation dependent, but typically faster than offline executors (no process invocation required). State-of-the-art tools make heavy use of immutable datastructures and copy-on-write optimizations [21, 29] to ensure that forking new states is fast (no need to duplicate states, just record the modifications) and that access to every state is immediate. Any part of the symbolic execution state that does not allow data sharing immediately penalizes symbolic execution. For example, modern symbolic executors make use incremental SMT solvers, i.e., solvers that keep internal state while solving constraints and can use previously asserted clauses to resolve new queries faster. Unfortunately, current implementations do not allow sharing of this state at the symbolic execution tree level, meaning that either solvers need to be replicated (too expensive memory-wise), or the solver's state needs to be reset, which requires a sequence of `push-pop` directives. Such operations are not constant time, and may depend on the distance of the two states in the symbolic execution tree.

Keeping all states in memory has advantages. Sharing between states is easier, data lookup is faster, and search strategies are much more flexible when compared to offline executors—online executors perform context switches at the instruction level, while offline executors at the path level.

Disadvantages are split in two categories: memory resources, and preserving side-effects. Exceeding memory resources is common in symbolic execution. No matter how much sharing exists, most programs can fork enough states that will exhaust the executor's memory (typically 4GB on a 32-bit process). To address the issue, executors such as KLEE [21]

have options to stop forking when a threshold is reached. Side-effects impose extra cost on context-switching and complicate the implementation. The incremental solver above was just one example, practical implementations have to deal with a large number of other side-effects at the operating system [3] (e.g., file descriptors) or lower levels [29] (e.g., video memory).

Hybrid Execution. MAYHEM [3] is a hybrid symbolic execution system. Instead of running in pure online or offline execution mode, MAYHEM alternates between modes to obtain the best of both worlds. Hybrid symbolic execution can context-switch quickly between states during the online exploration, while extraneous states are swapped to disk and explored in a new exploration. The startup and path re-execution costs are amortized among many paths (instead of a single path as in offline execution). To make re-execution faster, hybrid execution concretely executes the statements up to the fork point; symbolic evaluation is not needed since the symbolic state is preserved. Figure 3.2 shows the intuition behind hybrid symbolic execution. For more details on hybrid symbolic execution, we refer the reader to Chapter 6.

3.2.2.3 Summary

Real Implementations. The cost of context-switching and scheduling between states is difficult to measure and very often ignored. However, especially for offline executors the re-execution cost may be very substantial. Figure 3.3 shows the exploration time for the MAYHEM symbolic executor running on `/bin/echo` using different limits on the maximum number of online execution states (with hybrid execution). MAYHEM spent more than 25% of the time re-executing concrete previous paths in the offline scheme. For the online case, only 2% of the time was spent context-switching. By keeping everything in memory, online execution also makes more efficient use of the PIN code cache [63], since instructions do not need to be reinstrumented. As a result, the code cache makes online execution 40× faster than offline execution for this benchmark.

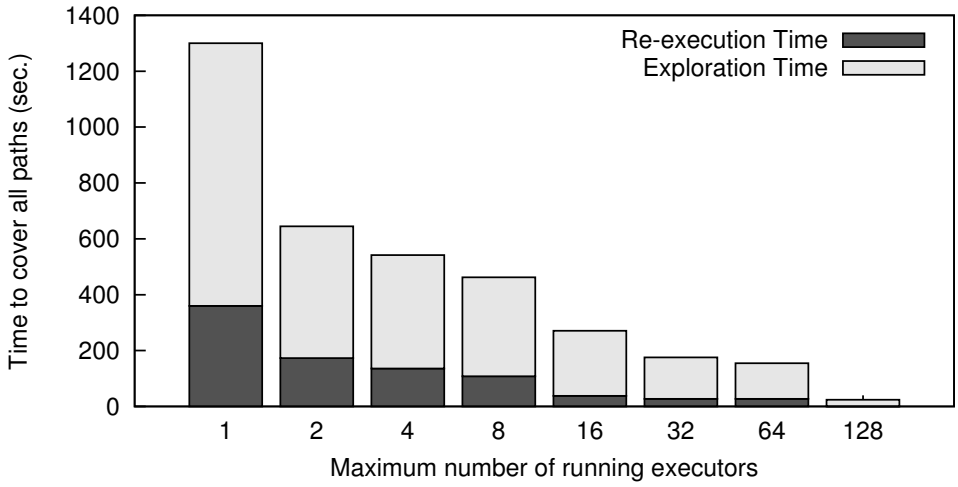


Figure 3.3: Exploration times for different limits on the maximum number of running executors.

Scheduling is equally important. Using the right search strategy can give vastly different results, e.g., we may go from 1 to 16 different exploits in our experiments (Chapter 5). Adaptive search strategies that constantly update the priority of pending execution states also carry overhead. Running an extra scheduling thread performing coverage-based analysis may increase exploration time by up to 37% in the MAYHEM executor.

Time & Space Cost. The cost for context-switching between states varies from constant $\mathrm{O}(1)$ to linear in the size of the path $\mathrm{O}(|\pi|)$. Online execution explores $\mathrm{O}(|\mathcal{T}_P(\iota)|_t)$ states, while offline execution may explore up to $\mathrm{O}\left(|\mathcal{T}_P(\iota)|_t^2\right)$ states. Hybrid execution is between these two bounds. The size of states is linear in the size of the tree $\mathrm{O}(|\mathcal{T}_P(\iota)|_t)$, assuming that state sharing is possible. We will refrain from labeling all search strategies with specific costs, since performance varies a lot depending on parameters. We mention however, that scheduling with graph-based search heuristics can be done in constant time.

3.2.3 Number and Cost of Queries

Symbolic execution transforms program fragments to formulas. The ability to query and solve these formulas allows symbolic execution to reason about program properties. In this section, we discuss the number of queries performed by symbolic execution (Section 3.2.3.1), and their solving cost in the context of SMT solvers (Section 3.2.3.2).

3.2.3.1 Number of Queries

On every conditional jump instruction, symbolic execution checks whether both branches are feasible; on every indirect jump, symbolic execution enumerates all possible jump values; on every assertion evaluation, validity is checked. Each one of these actions requires solving a formula with symbolic variables, a potentially expensive process—an NP-hard problem, see Section 3.2.3.2 for more details. Reducing or even avoiding queries can dramatically increase symbolic execution performance [21].

However, before discussing query elimination, we need to answer the more immediate question: how many queries are performed by symbolic execution? The answer is implementation- and application-dependent; here we present well-known bounds for the number of queries at each level:

- **Per Instruction.** Every $\mathsf{assert}\ e$ performs a validity query, so at least a constant number of queries $\mathrm{O}(1)$ is required per instruction ($\mathcal{N}_I^S(i)$). Instructions manipulating memory ($\mathsf{load}(\mu, e_i)$, $\mathsf{store}(\mu, e_i, e_v)$), may require finding the bounds of a symbolic pointer ptr, typically done by performing a binary search in the solver [29, 3], thus resulting in $\mathrm{O}(|ptr|_b)$ (or equivalently $\mathrm{O}(\log |\mu|_b)$) queries[7]. Finally, for fully symbolic jumps ($\mathsf{if}\ true\ \mathsf{jump}\ e$), the symbolic executor may need to enumerate $\mathsf{values}(\Pi, e)$,

[7]Executors that concretize symbolic memory, e.g., SAGE [12], do not have this cost.

including all possible valid jump targets in the program $\mathrm{O}(|P|)$—or $\mathrm{O}(|\mu|_b)$ if the program resides in memory—each one requiring a separate query.

- **Per Path.** Each path is forked because of a single query: the one that determined the target address of the last `jump`. Thus, despite performing $\mathrm{O}(|P|)$ queries in a single statement, each query corresponds to a path, effectively amorting the cost per path. For example, a branching statement may perform 2^{32} queries and fork 2^{32} new paths; the cost is still 1 query per path. With a constant startup cost, and a constant number of queries per instruction, every path will perform linear $\mathrm{O}(|\pi|)$ queries per path π. Note that we cannot have a constant cost per path due to assertions and infeasible branches. In systems with symbolic pointer resolution may increase this limit to $\mathrm{O}(n \cdot |\pi|)$, for pointers of bitwidth n.

- **Per Program.** Following the same instruction-level reasoning as above, the number of queries performed will be linear in the size of the symbolic execution tree $\mathrm{O}(|\mathcal{T}_P(\iota)|_t)$. Again, for systems with symbolic pointers of bitwidth n, the number of queries will be $\mathrm{O}(n \cdot |\mathcal{T}_P(\iota)|_t)$.

Longer traces are expected to have more queries, a trend that seems to hold in Figure 3.4, which shows a sample from symbolically executing 1,023 programs with the MAYHEM symbolic executor. Thus, eliminating queries should allow symbolic execution to explore longer traces within the same amount of time.

We now move to two heavily used approaches for eliminating queries: caching, and simplifications/approximations.

Caching. Memoizing previously resolved queries is by far the most widely used technique for eliminating unnecessary solver invocations; a hashtable lookup will—in most cases—outperform a solver query. A multitude of caching schemes have been suggested in the literature; we mention below three examples:

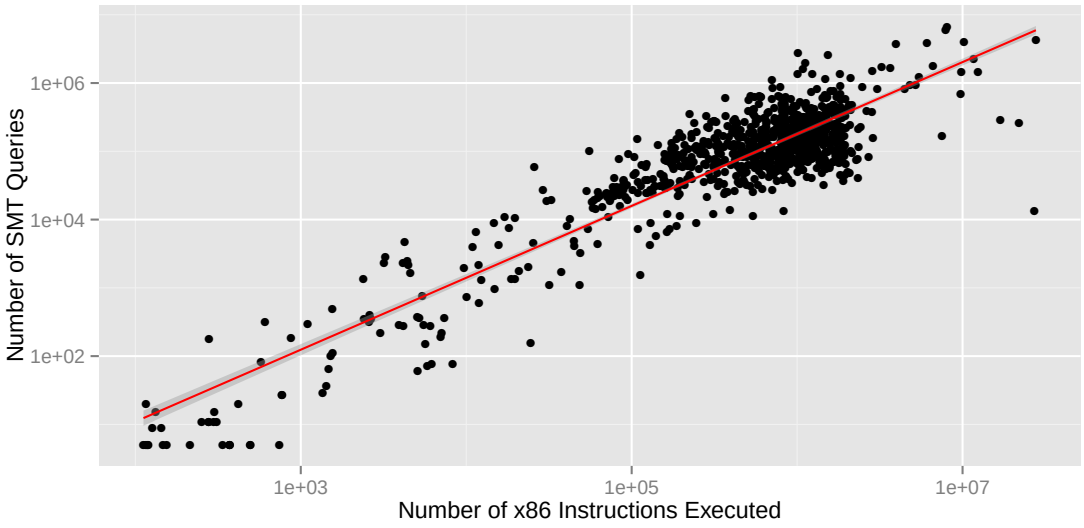


Figure 3.4: Number of symbolic x86 instructions executed with number of SMT queries resolved from our BIN suite of 1,023 programs (Chapter 7).

- **Lemma Cache.** The lemma cache stores the results of previously resolved queries. For solvers running in incremental mode, the cache needs to store the solver context (previously queried formulas) along with the formula. Every modern symbolic executor we are aware of [21, 12, 3], has a lemma cache implementation. Unsatisfiability checks [30] capture negations of already known formulas. Normalizing formulas using De Bruijn (α-equivalent) indices [3] improve cache performance (Chapter 6).

- **Counter-example Cache.** Proposed by Cadar *et al.* [21], the counter-example cache keeps variable assignments from previous solutions, detects whether the current set of constraints is a subset or superset of known formulas, and returns a solution whenever possible. For example, if the constraints are a subset of a satisfiable formula, the subset is also satisfiable.

- **Refinement Cache.** Finding the bounds of a symbolic pointer *ptr* requires requires $O(|ptr|_b)$ solver invocations for the binary search. The refinement cache [3] stores

previously detected bounds based on the symbolic index expression. Checking whether the bounds are correct requires fewer checks than resolving the bounds again (Chapter 6).

A cache without hits is just overhead [19]. Depending on the application, extra heuristics can be utilized for improving the hit rate. Further, keeping every satisfying assignment from every query performed results in substantial memory overhead. To handle this overhead, cache eviction strategies are employed, e.g., the lemma cache in MAYHEM employs a Least Recently Used (LRU) policy.

Simplifications & Approximations. In some cases, queries can be entirely eliminated by sacrifing accuracy or performing simplifications. We present three examples below:

- **Concretization.** During evaluation, symbolic execution has access to the concrete state of the program, i.e., has a concrete assignment to input variables. Using this assignment, satisfiable constraints can be identified (simply through expression evaluation). For example, during the evaluation of an $\mathsf{if}\ e\ \mathsf{jump}\ e_1$ statement, e will be either *true* or $false$ with the concrete input—thus one of the two $\mathsf{isSat}()$ queries of FORKCOND can be eliminated [3]. Similarly, the concrete state can be used to concretize symbolic expressions, i.e., substitute them with concrete values. Concretization, especially for symbolic pointers may reduce the number and complexity of queries [12], but may make formulas too constraining [3].

- **Approximation.** Enumerating all possible values of a symbolic pointer may be too expensive, since it requires $\mathrm{O}\left(|\mu|_b\right)$ (the size of memory) queries in the worst case. Depending on the precision required, we can avoid using the solver and use approximations for finding the range of a symbolic pointer expression such as Value Set Analysis (VSA) [64]. For example, the range of a symbolic pointer that may point to 250 bytes within a 256-bytes memory region, may be approximated by the 256-byte region without including too many extraneous values. MAYHEM [3] and

MergePoint [5] make use of both over- and under-approximations to make pointer resolution faster (Chapter 6).

- **Algebraic Simplifications.** As mentioned in Section 3.2.1, symbolic executors make heavy use of rewriting and simplification rules. Simplifications may reduce a symbolic expression to a constant, effectively eliminating the need for a query.

Real Implementations. Figure 3.4 shows that symbolic instructions executed and queries performed are positively correlated, with a 0.49 correlation coefficient. The average (per program) number of queries per symbolic assembly instruction executed was 0.024 (24 queries for every 1,000 instructions). A single 30 minute experiment with our BIN suite (Chapter 7) resolves approximately 200 million SMT queries, with 172 million being caught by the lemma cache (86% hit rate). With the lemma cache, the average (per program) number of queries reaching the solver per instruction goes down to 0.0003 (3 queries on every 10,000 instructions). Similarly, by layering caches with approximations, pointer resolution queries may be reduced by up to 99.9% (Chapter 6).

Time & Space Cost. Cache lookups are fast: typically $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$ for n cache entries, depending on the implementation. Note that to get the above bounds the cache requires constant time hashing of the formula expression, which is offered by hash-consing [65]—Chapter 7 discusses an approach for integrating hash-consing in the expression language. The more serious cost in caching is the memory overhead, which is linear in the number of queries ($\mathrm{O}(|\pi|)$ or higher if we consider symbolic pointers). Approximations and simplifications are usually linear (depending on the application it can be higher) in the size of the expression, and thus the path $\mathrm{O}(\pi)$.

3.2.3.2 Cost of Queries

Given a quantifier-free boolean bitvector expression exp with input domain Σ^n, a query—within the scope of the thesis—checks the validity or satisfiability of exp. SMT solvers [66] are typically used to perform such checks, and queries are called SMT queries (the two terms will be used interchangeably). The general satisfiability/validity problem is NP-hard. Despite research efforts to reduce the number of queries (via caching etc.) current symbolic executors still spend a substantial amount of time in the SMT solver [30, 21, 12, 62, 3, 5].

Numerous approaches have been proposed for optimizing solving times in symbolic execution, e.g., removing redundant constraints, reducing the formula size, and so on. Below, we present a sample of these techniques:

- **Independent Formulas.** Splitting a formula to independent smaller subformulas with disjoint sets of variables can significantly reduce formula size. For example, consider a formula $f(x, y)$ that is the conjuction of two independent clauses $f_1(x) \wedge f_2(y)$. Reasoning about each conjunct can be done independently, and can lead to smaller and faster queries. The effectiveness of the technique has been demonstrated multiple times [19, 21, 30, 12, 3]. Also, having smaller independent formulas improves cache performance [21].

  We mention two potential caveats with independent formulas. First, a single checksum-like clause that involves all input variables disables the optimization, since there will be no independent variables. Note that the operation does not need to be a cryptographic checksum; it can be a simple `strlen` on a symbolic buffer and returning the length $(\mathsf{ite}(buf[0], 0, \mathsf{ite}(buf[1], 1, \dots)))$. Second, independent formulas is not very compatible with incremental solving. For example, previously pushed constraints related to $f_1(x)$ need to popped to reason about $f_2(y)$ independently. Keeping multiple solver instances is possible, but has high memory overhead [3].

- **Constraint Subsumption.** Removing redundant constraints can simplify all follow-up queries. Constraint subsumption checks if the newly added constrained is subsumed by previous constraints, and if so ignores it. For example, consider adding the constraint $c = x > 42$ to the path predicate $\Pi = x > 40$. The constraint is subsumed by the path predicate ($\Pi \implies c$), and thus does not need to be added. Constraint subsumption is used widely [21, 12, 3]. Implied value concretization [21] is an instance of the opposite process where a newly added constraint is too constraining, e.g., $c = (x = 42)$, and can simplify away the existing predicate (to $\Pi = (x = 42)$ or even $\Pi = \mathit{true}$ if the context variable is concretized).

- **Incremental Solving.** Advances in SMT capabilities can directly improve performance. For instance, modern solvers support incremental solving [66], meaning that learned lemmas during previous invocations until the last pushed constraint persist and reused in new queries. This way, the solver does not need to reason again and again about the same formula prefixes, only the new part of the formula that is added. Pushing constraints and incremental solving can be especially useful during query bursts, such as when enumerating the values ($\mathsf{values}(P, e)$) of a pointer (Chapter 6).

- **Timeouts.** A straightforward way of avoiding hard queries is to give the SMT solver a hard timeout. Combined with a search strategy that lowers the priority of states that timed out, the path throughput of symbolic execution can be increased. Different strategies have different guarantees. For example, doubling the timeout for every state that times out, ensures a $2\times$ worst case complexity. Every practical symbolic executor implementation comes with a customizable solver timeout [21, 29, 3].

- **Aliasing.** Queries on symbolic arrays, and reasoning about memory aliasing in the solver can be very expensive (Chapter 6). At the source code level, where types and abstractions are available, object tracking can help reduce the number of symbolic

pointers [21] and avoid part of the aliasing. However, at the binary level, such abstractions are typically missing[8], and introducing an array of 2^{32} bytes (for a 32-bit system) in the formula is not appealing for performance. Removing memory expressions from formulas and making aliasing explicit can boost performance (Chapter 6).

- **Scheduling Heuristics.** Kuznetsov *et al.* [26] used heuristics to predict the difficulty of formulas, based on the number of occurrences of variables in future queries. They used the difficulty heuristic to decide when and if to merge symbolic execution states during exploration. We will discuss merging opportunities further in Chapter 7.

- **Custom Simplifications.** Rewriting rules targeted at simplifying expressions that the underlying solver does not handle efficiently are very common and can significantly boost performance. In Section 3.2.1.1 above, we saw one such simplification, where undoing a compiler simplification sped up solving by orders of magnitude. We believe the place for such rewriting rules is in the solver, and that enhancing solvers with a plugin architecture for rewriting rules would be invaluable for symbolic executor implementations.

We stress again, that the above list is not exhaustive. We refer the reader to the bibliography for more inforation on optimizing solving times [60, 61, 25].

Real Implementations. Using the MAYHEM executor in 2013, we resolved more than 16 billion SMT queries, spanning across 797 experiments on more than 37,391 programs. Figure 3.5 shows the empirical query time distribution from 4 different experiments ran on 558 programs from our BIN suite (Chapter 7) for a total of more than 100 million queries. MAYHEM was configured for full logging (writing all formulas to disk) with 14 bytes of symbolic arguments, and varied the symbolic file size (32 and 64 bytes), and technique (with

[8]There is an abundance of work on recovering such abstractions [67, 68].

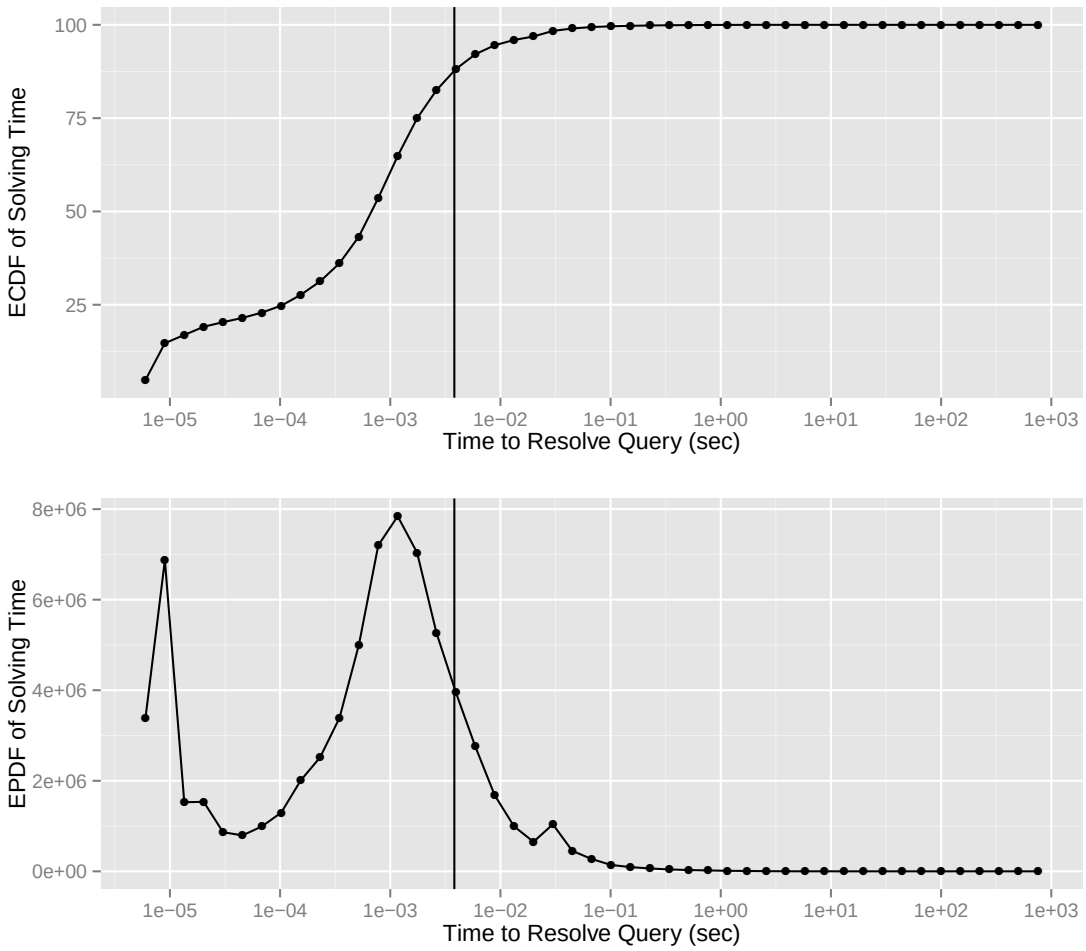


Figure 3.5: Empirical probability density function (EPDF) and cumulative density function (ECDF) of formula solving time on a sample dataset.

and without veritesting—see Chapter 7). The average query time is very low at 3.84ms with a standard deviation of 115ms. We see that the vast majority of queries takes less than 100ms to solve (99%), and almost all (99.9%) require less than a second. Our results are not unique, other symbolic executors have reported similar statistics—e.g., in SAGE 90% of all queries take less than 100ms and 99% take less than 1 second [23].

The moderate variance indicates the distribution is weakly heavy tailed. The time spent resolving queries more than 100ms accounts for 38% of the total time, while the percentage goes down to 13.5% for queries above 1sec. Thus, the majority of the solving time (62%) is

spent resolving queries that take less than 100ms. 62% of our query database consists of true queries.

A large portion of the aforementioned techniques for optimizing solving time focused on formula size, with the expectation that smaller formulas will be easier to solve. Is that a reasonable expectation? Figure 3.6 shows formula solving time with the size of newly added formulas—the plot does not include the total formula size, which does not show a specific trend—with the solver was running in incremental mode. The plot is a 5000-point sample uniformly drawn from buckets from our dataset. There seems to be an upwards trend, but the relation is not clear. Also, note that the axes are both logarithmic, the LOESS curve does not necessarily correspond to a linear relation. We investigate further the connection between solving time and size in Chapter 7.

At the lower-level the solver blasts the problem to SAT, selects a satisfying assignment to perform unit propagation with DPLL [69], and finally uses Conflict-Driven Clause Learning (CDCL) [70] to learn from conflicts, i.e., from variable assignments that lead to contradiction. We elide details here, but the interested reader may refer to bibliography for more information on SMT solvers. The high-level idea is the SMT solver is solving a search problem, and a conflict denotes whether a previous choice was problematic.

Figure 3.7 shows the same sample as above, but shows the relation between conflicts found by the solver with solving time. The data points are better clustered, and the trend seems clearer here; more conflicts increase the solving time steadily. Above the main body of data-points, there appears to be a smaller cluster that does not follow the same trend, where conflicts are much more expensive. Manual inspection showed that the vast majority of the points in the cluster come from a single program: `whatis`, a Linux utility for displaying manual page descriptions. To find the right page, `whatis` performs a symbolic hashtable lookup, which creates formulas that require reversing the hash function of the hashtable.

Figure 3.6: Solving time with the size of a formula $||_s^e$, measured in AST QF_BV nodes.

Figure 3.7: Solving time with the number of solver conflicts.

Using the solver to reverse hash functions results in hard queries [71], and manual inspection showed that the hard queries indeed contained expressions with the hash computation.

The above observation, shows that the behavior of a single program may be entirely different from others. In Chapter 7 we will see that program dependent plots are better suited from identifying clear performance trends.

Time & Space Cost. Satisfiability for a query with bitvector arithmetic without uninterpreted functions in SMT-LIB format [72] is an NExpTime-complete problem [73][9], and currently there is not a tight bound even for fixed-size bitvector logics [73]. A trivial upper bound $\mathrm{O}\left(|exp|_s^e \cdot 2^{|\iota|_b}\right)$ can be obtained by a simple enumeration and test algorithm: 1) to enumerate all possible inputs $\iota \in \Sigma^n$ in $\mathrm{O}\left(2^{|\iota|_b}\right)$ steps, and 2) for each of the possible values we need to evaluate the queried formula exp, a process requiring $\mathrm{O}\left(|exp|_s^e\right)$ steps. Thankfully, as shown from the trend lines in Figures 3.6 and 3.7, modern SMT solvers can resolve real queries much more efficiently in practice.

3.3 Example: Acyclic Programs

To better understand the interplay between all the above components, we will now obtain an approximate upper bound of the symbolic execution cost for acyclic programs. Specifically, we will target $BIL_{\mathcal{A},\mathcal{K}}(QF\_ABV_{||_s^e \leq 1})$, i.e., acyclic BIL programs with known control flow and flat expressions. We will assume a DFS search strategy, constant time context-switching, and scheduling will be offline (SAGE [12]-style trace-based). The input domain will be Σ^n.

An acyclic program with n branches has at most $\mathrm{O}\left(2^n\right)$ paths (every branch potentially doubles the number of paths). Thus, the number of feasible paths is: $\left|\mathcal{F}\left(P, \Sigma^n\right)\right| = \mathrm{O}\left(2^{|P|}\right)$[10].

[9]Depending on the logic or the encoding the complexity class may be different. For example, QF_BV with unary encoding is NP-complete [74].

[10]Note this bound is true only for programs with known control flow. If we allow computed indirect jumps, every branch can potentially point to any statement in the program, suggesting a bound $\mathrm{O}\left(|P|^{|P|}\right)$. More

Let $\mathcal{I}_I^{max}(i) = \max_{\pi \in \mathcal{F}(P,\Sigma^n)} \max_{i \in \pi} \mathcal{I}_I^{\{s_i\}}(i)$ be the maximum instruction cost. It is possible to bound the instruction cost because scheduling, context-switching, and instruction evaluation (flat expressions) are constant time. Thus, have that $\mathcal{I}_p^S(\pi) \leq \mathcal{I}_I^{max}(i) \cdot |\pi|$, which means that $\mathcal{I}_p^S(\pi) = \mathrm{O}(|P|)$.

The number of queries per path depends on the implementation. In Section 3.2.3.1, we showed that the number of queries will be at least linear in the size of the path (thus $\mathrm{O}(|P|)$), when all instructions executed are assertions. For exposition, we will only consider queries that consist of the path predicate conjoined with a set of added constraints (their size will be bounded by a constant—similar to Chapter 4). The size of the path predicate is linear in the size of the path[11], and thus the cost of evaluating a formula with a given solution is $\mathrm{O}(|P|)$.

Thus, using equations Equations (3.1), (3.4) and (3.6) and the above bounds we get the following (non-tight) upper bound on the time complexity of a symbolic executor on $BIL_{\mathcal{A},\mathcal{K}}(QF\_ABV_{||_s^e \leq 1})$ programs:

$$\mathcal{C}^{\{s_{init}\}}(P) = \mathrm{O}\left(\underbrace{2^{|P|}}_{\#\text{ of Paths}} \cdot \left(\underbrace{|P|}_{\text{Interpreter Cost}} + \underbrace{|P|}_{\text{Queries Performed}} \cdot \underbrace{\mathcal{Q}\left(|P|, \Sigma^{|s_{init}(\iota)|_b}\right)}_{\text{Query Cost}}\right)\right) \tag{3.8}$$

where $\mathcal{Q}(|P|, \Sigma^n)$ is the cost of resolving a formula of size $|P|$ and inputs from Σ^n (as mentioned above (Section 3.2.3.2), a trivial bound would be $\mathrm{O}(|P| \cdot 2^n)$, with an enumeration and test algorithm). Excluding linear factors, Equation (3.8) has two main components: 1) a factor exponential in the size of the program (path explosion), and 2) an unpredictable cost factor (query solving), potentially exponential in the size of the input. Both factors pose important scalability challenges for symbolic execution and are the motivation for most of our work (Chapters 5 to 7).

accurately, for acyclic (no statement repetition) programs a better upper bound is $\mathrm{O}(|P|!)$—all statement permutations.

[11] A path is a straightline sequence of instructions and can be converted in a formula that is linear in the number of instructions [34]. Linearity is expected since the path predicate is just a symbolic expression built by executing a sequence of statements (Section 3.2.1.1)

To put Equation (3.8) into perspective, we compare it with efficient verification algorithms for acyclic programs (circuits), which yield formulas that are quadratic in the size of the program [34, 75, 76]. The quadratic increase comes from the translation of the program to Static Single Assignment (SSA), and is closer to linear in practice [77]. Such techniques convert the entire program into a single formula of size $\mathrm{O}\left(|P|^2\right)$, thus getting the following (non-tight) upper bound for the cost of verification:

$$\mathcal{C}_{VC}^{\{s_{init}\}}(P) = \mathrm{O}\left(|P|^2 + \mathcal{Q}\left(|P|^2, \Sigma^{|s_{init}(\iota)|_b}\right)\right) \tag{3.9}$$

Comparing Equation (3.8) and Equation (3.9), we can make two observations: 1) the path explosion factor is not explicit in Equation (3.9) (as we will see in Chapter 7 path explosion is really encoded in the formula), and 2) the formula size increased from linear to quadratic in the formula solving factor. The two equations already highlight a trade-off between path explosion in symbolic execution and larger formulas in verification. In Chapter 7, we explore this trade-off further, and use verification techniques in symbolic execution to reduce the path explosion factor, to find more bugs, obtain higher code and path coverage, and complete program exploration faster.

3.3.1 Loops and Undecidability.

Equation (3.8) is limited to acyclic programs. Nevertheless, it still provides the intuition about the cost breakdown of symbolically executing a program with loops within a finite amount of time. During a finite-time exploration, symbolic execution will analyze only a fragment of the symbolic execution tree $\mathcal{T}_P(\iota)$, corresponding to an unrolled version of the original program. We still expect the number of paths to be exponential in the number of symbolic branches, and formulas to be linear in the size of the executed path.

Reasoning about programs with loops is generally undecidable. An infinite loop is enough to make symbolic execution non-terminating (Chapter 2). Automatically detecting loop

termination [48], or finding loop invariants [78, 79] is an active area of research. However, undecidability is still a matter of expressivity. For example, if we limit our scope to $BIL_{\mathcal{M}}(QF\_ABV)$ programs[12], i.e., programs with a single finite-sized memory, the total number of possible execution states is still very large, but finite $O\left(2^{|\mu|_b}\right)$, and termination detection can be decidable. We believe that finding the right balance between expressivity and practicality is key for advancing our abilities to reason about real-world software in the future.

[12]Note this is not an unreasonable assumption, most computer systems today—with the exception of some well-advertised cloud storage solutions—have *finite* memory.

Chapter 4

Automatic Exploit Generation

Life is too short for non strongly typed languages.

— Alexandre Rebert, *Group meeting.*

Attackers only need to find a single exploitable bug in order to install malware, bots, and viruses on vulnerable computers. Unfortunately, developers rarely have the time or resources to fix all bugs. This raises a serious security question: which bugs are exploitable, and thus should be fixed first? To address this question, we have been developing techniques for Automatic Exploit Generation (AEG). AEG proves when a bug is security-critical by generating a working exploit. In this chapter, we present the concept behind AEG and show how symbolic execution can be used to model, find, and demonstrate control flow hijack vulnerabilities.

4.1 Introduction

Buggy programs are one of the leading causes of hacked computers. Security-critical bugs pave the way for attackers to install Trojans, propagate worms, and use victim computers to

send spam and launch denial-of-service attacks. Thus, a direct way to make computers more secure is to fix bugs before they are exploited.

Unfortunately, bugs are plentiful. For example, the Ubuntu Linux bug management database currently lists over 103,000 open bugs. Specific widely-used programs such as the Firefox web browser and the current Linux 3.x kernel have 7,597 and 1,293 open bugs respectively[1]. Other projects, including those that are closed-source, likely have similar statistics. These are just the bugs that we already know about—there is always the persistent threat of zero-day exploits where attackers discover and craft an attack for previously unknown bugs. Thus, the question is not whether an attacker can find a security-critical bug, but *which* bug an attacker will find and exploit first.

Among the thousands of known bugs, which would you fix first? Which bugs are exploitable? How would you go about finding unknown exploitable bugs that still lurk?

In this chapter we introduce the Automatic Exploit Generation (AEG) challenge. Given a program, the AEG research challenge is to automatically find bugs *and* generate working exploits. The generated exploits unambiguously demonstrate that a bug is security-critical. As a result, AEG provides concrete, actionable information to decide which bugs to fix first.

Automatic exploit generation is cast as a program verification task, but with a twist. Traditional verification takes a program and a specification of safety as inputs, and verifies that the program satisfies the safety specification. The twist is we replace typical safety properties with an "exploitability" property, and the "verification" process becomes finding a program path where the exploitability property holds. Casting AEG as a verification task ensures that AEG techniques are based on a firm theoretic foundation. In our setting, *sound* means that if our analysis says a bug is exploitable, it really is exploitable. The working exploit guarantees soundness because it proves that a bug is exploitable.

[1]Numbers reported from respective bug tracking database as of January 27, 2013. Excludes bugs with severity wishlist, unknown, undecided, or trivial tag.

Verification has many well-known scalability challenges, and these challenges are exacerbated in AEG. Each new branch potentially doubles the number of possible program states, which can lead to a combinatorial explosion of states to check for exploitability. Traditional verification takes advantage of source code, models, and other abstractions to help tackle the state explosion to scale. Unfortunately, abstractions often leak by not perfectly encapsulating all security-relevant details, and the leaky points tend to affect the quality of security analysis. For example, writing one byte past a declared 11-byte array in C is wrong, but unlikely to be exploitable because most compilers will pad the array with extra bytes to word-align memory operations.

In order to provide high fidelity, most AEG work analyzes raw executable code. Executable code analysis is needed because many exploits rely on low-level details such as memory layout and CPU semantics, which are explicitly represented in executable code. Executable analysis is also widely applicable because everyone typically has access to executable code of the programs, thus can audit the code for security-critical bugs.

Throughout the thesis we focus on AEG as a defensive tool for prioritizing exploitable bugs. We are cognizant, however, that there are obvious offensive computing implications and applications. We believe that being aware of the offensive capabilities of an attacker is necessary for setting up proper defenses. AEG is such a capability, and a defender should be aware of AEG-like techniques before releasing their software.

Further, we describe current research in AEG, current successes, as well as limitations. We primarily focus on control flow hijack exploits that give an attacker the ability to run arbitrary code. Control flow hijacks are among the most serious threats to defenders and the most coveted exploits by attackers [80, 81]. Although most current researchers focus on control flow hijacks because of their immediate danger, AEG is not limited to only this class of attacks. Exploitable bugs can be found in programs in all languages, and the verification-based approach to AEG still applies.

AEG promotes two general research insights. First, that AEG is a type of verification. The better we get at verifying programs are safe, the better we will get at automatically generating exploits. Second, bugs are plentiful, and we need effective techniques like AEG to identify and prioritize security-critical bugs so that they are fixed first.

4.2 Exploiting Programs

Suppose you are interested in finding exploitable bugs in the `/usr/bin` directory of the latest Debian OS. We downloaded Debian 6.0.5 (the latest stable release at the time), and in our installation there are 1168 binary executables in that directory.

One simple way to find bugs is to perform black-box fuzzing, which is a program testing technique that runs the program on inputs from a fixed alphabet. Fuzzers typically try extreme values, such as 0 and the native maximum integer. The "black-box" is the program itself, whose content is not analyzed at all. The fuzzer chooses the inputs and observes the program, looking for hangs, crashes, buggy outputs, or other indications of a bug.

We fuzzed our 1168 programs using the following script:

```
1 for letter in `/bin/echo {a..z} {A..Z}`; do
2   timeout -k 1 -s 9 1s
3       <program> -letter <path>
4 done
```

The script tries all command line options from a to Z, followed by a 6676-byte path name we created on the file system. The `timeout` command limits total execution to 1 second, after which the program is killed.

The script took about 13 minutes to fuzz all programs on our test machine, yielding 756 total crashes. With a little analysis, we found that many of the crashes are due to the same

```
1  int main(int argc, char **argv){
2    char *name; int i;
3    for(;;){
4      i = getopt(argc, argv, ``c:s:t:vh'');
5      if(i == -1) break;
6      switch(i) {
7        case 'c': ...; break;
8        case 's': name = optarg; break;
9        ...
10     }
11   }
12   sock_fd = ud_connect(name);
13   ...
14 }
15 int ud_connect(const char *name){
16   int fd;
17   struct sockaddr_un {
18     sa_family_t sun_family;
19     char sun_path[108];
20   } addr;
21   ...
22   sprintf(addr.sun_path, "%s", name);
23   ...
24   return fd;
25 }
```

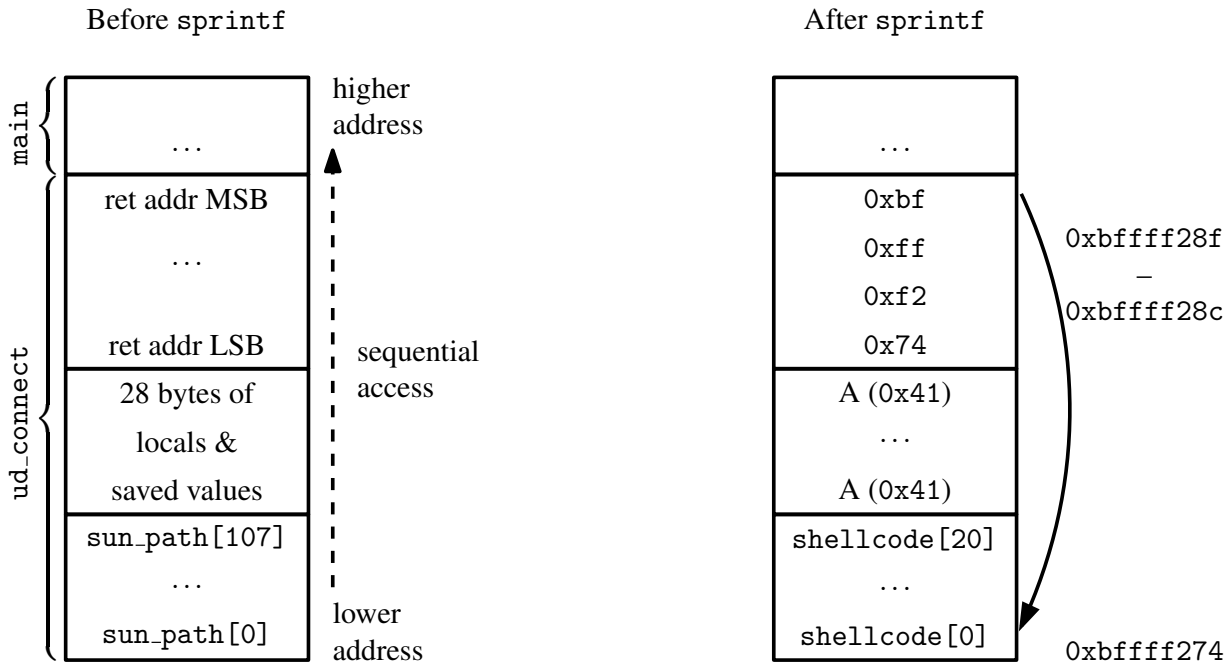


Figure 4.1: Our running example: a buffer overflow in `acpi_listen`.

bug (e.g., different command line options triggered the same buggy code) and we were able to pared down the list to 52 distinct bugs in 29 programs. Now, which bug would you fix first?

We argue the exploitable ones should be first, but the problem is determining which of the 52 are exploitable. We first describe *manual* exploit generation in order to introduce terminology, and to give a flavor of how exploits work in practice. In particular, we describe control flow hijack exploits, which have been a staple class of exploits in the computer security industry for decades. Well-known examples of control flow hijacks include exploits in the Morris worm in 1988, and Stuxnet in 2010. For now, we forgo several important issues relevant in practice, such as whether the buggy program is a realistic attack target, and whether additional OS defenses would protect the otherwise exploitable program from attack. We tackle these issues in later sections.

Figure 4.1 shows a bug discovered in `acpi_listen`, which we use as our running example. `acpi_listen` listens for events from the Advanced Configuration and Power Interface daemon (acpid). A buffer overflow occurs on line 22. The program reads in a command line argument, and if it is `-s` (line 8), assigns the following argument string to the `name` variable. On line 22, the `sprintf` function copies `name` into `sun_path`, a field in a local instance of the networking `sockaddr_un` data type (a standard data structure in UNIX for sockets).

The bug is that `sun_path` is a fixed-size buffer of 108 bytes, while the command line argument copied through `name` into `sun_path` can be of any length. The C standard doesn't specify what happens if `name` is more than 108 bytes, but only notes that the resulting execution is undefined by the standard itself. Of course, when run on real hardware something will happen, and with our fuzzing script the program crashed. Worse, this crashing bug can be turned into a control flow hijack.

All control flow hijack exploits have two goals: hijack the control and run an attacker computation. The mental model of an attacker is to first determine if any of the executed instructions can be subverted, and then to figure out how to execute their own attacker-

supplied computation. For `acpi_listen`, some of the details an attacker must understand in-depth include: i) the basic execution model for compiled programs, ii) how function calls are implemented, iii) how writing outside allocated space can hijack control, and iv) how we can direct the hijacked control to run the attacker's code. Since any discussion of creating exploits against vulnerable C programs assumes a basic understanding of these facts, we give a brief overview here.

During runtime, computer hardware implements a fetch-decode-execute loop to run a program. The hardware maintains an instruction pointer register (IP), which contains the memory address of the next instruction to be executed. During the fetch phase, the hardware loads the data pointed to by the IP register. The data is decoded as an instruction, which is then executed. The IP is then set to the next instruction to be executed. Control is hijacked by taking control of the IP, which is redirected to run the attacker's computation.

A straightforward exploit for `acpi_listen` hijacks control by overwriting data used to implement function returns. Function calls, returns, and local variables are not supported directly by hardware. Instead, the semantics of these abstractions are implemented by the compiler using low-level assembly instructions and memory. There are many details a real attacker must be proficient in, such as where arguments are passed, how values in registers are shared between the caller and callee. Without going into detail, we assume a standard C calling convention known as "cdecl". The compiled assembly implements a stack abstraction in memory where function calls push space for local variables, arguments to future calls, and control data onto the stack. A function call return pops the allocated space off the stack. Thus, the stack will grow a bit for each call, and shrink a bit on each return.

A call from a function f to a function g will first push onto the stack the address of the next instruction to execute in f when g returns. The pushed instruction address is called the *return address* because control flow returns to the instruction at this address when g returns. Next, space is created for g's local variables, and then f will hand control over to g, letting g

execute. When g returns, the hardware pops off the saved return address into the IP register, and continues execution. An important detail is that whatever address is popped into IP will be executed next, regardless of whether its the same as the original pushed value.

The stack frame just before `sprintf` is called on line 22 is shown in Figure 4.1. The flow of execution for creating the stack is:

1. When `main` called `ud_connect`, `main` pushed the next instruction to be executed (the instruction corresponding to line 13) onto the stack.

2. `main` transfered control to `ud_connect`.

3. `ud_connect` allocated space for its local variables. On our computer, 108 bytes were allocated for `sun_path`, and an additional 28 bytes for other data such as other local variables, saved register values, and other runtime data.

4. The body of `ud_connect` ran. When `sprintf` is called, a similar flow will push a new return address on the stack, push new space onto the stack for `sprintf`'s local variables, and so on.

5. When `ud_connect` returns, it will first deallocate the local variable space, and then pop off the saved return address into the IP register.

6. Under normal operation, the return address will point to the instruction for line 13, and `main` will resume execution.

The crux of a control flow hijack is that memory is used to store both control data and program variable values, but that the control data isn't protected from being overwritten in a variable update. Control flow hijacks are an instance of a channeling vulnerability, which arise when the control and data planes are not rigorously separated. For this particular example, an out-of-bound write can clobber the return address. When `sprintf` executes, it

will copy data sequentially from `name` up the stack starting from the address for `sun_path` as shown. The copy only stops when a NULL character is found, which is not necessarily when `sun_path` runs out of space. A long `name` will clobber the saved local variables, and eventually the saved return address. Since we are assuming an attacker specifies `name`, they can ultimately overwrite the return address with almost any value of their choosing.

Attackers need to analyze the runtime behavior of a program in order to figure out exactly how many bytes to write, what constraints there may be on the bytes, and what would be a good value with which to overwrite the return address. For `acpi_listen`, a string of length 140 will overwrite the return address. The first 108 bytes will be copied into space allocated for `sun_path`. The next 28 bytes on the stack are intended to hold local variables and saved register values. The final 4 bytes will overwrite the saved return address.

When `ud_connect` returns, the overwritten return address will be popped off the stack into the IP register. The machine will continue executing the instruction starting at the overwritten address. While this example overwrites the return address, there are a variety of other control data structures that can be used to seize control. Examples include function pointers, heap meta-data, and C++ virtual function tables.

Once an attacker can hijack control, they want to divert control to their own computation. Attackers have a number of techniques for specifying a computation, with names like code injection, return-to-libc, and return-oriented programming. The most basic attack injects executable code into the vulnerable process.

Suppose an attacker wants to execute the command line interpreter `/bin/sh`. This is a natural choice because it allows the attacker to subsequently enter any additional commands they want. In fact, executing `/bin/sh` is so popular that colloquially any attacker code is called "shellcode". A classic approach is to give executable code as input to the program, and redirect control flow to the given executable code. The executable code itself can roughly be created by first compiling a C program that executes `/bin/sh`, e.g., via the

`execve("/bin/sh", args, NULL)` system call. The resulting binary code is simply a string, which for a control flow hijack attack, we treat both as executable code and data.

The above description of shellcode, memory and stack layout, and low-level execution behavior illustrates just some of the large number of details an attacker must contend with to craft an exploit manually. By putting all the details together, the attacker can get `acpi_listen` to execute `/bin/sh`. This is because the bug in `acpi_listen` allows the attacker to redirect control to an arbitrary address and inject new code that will execute. The final step is to make sure the return address is overwritten with the address of the shellcode. On our machine, `sun_path` is at memory address 0xbffff274. The complete exploit for `acpi_listen` is an input where:

- The first bytes of the command line argument are the shellcode above. The shellcode we generated (not shown) is 21 bytes, and in this case the first 21 bytes will be copied into bytes 0-20 of `sun_path`.

- The next 115 bytes of input are any non-NULL ASCII value. These bytes are copied into bytes 21-107 of `sun_path` and the additional space for other locals.

- The last 4 bytes of input are the hex string `0x74 0xf2 0xff 0xbf`. These bytes overwrite the return address. When the return address is popped, the bytes become the address 0xbffff274 (because x86 is little endian), which is the address of the shellcode stored in `sun_path`.

The stack frame after supplying the above string as a command line argument following `-s` is shown in Figure 4.1. When `ud_connect` returns, the address 0xbffff274 will be popped into IP, and the hardware will fetch, decode, and execute the bytes in `sun_path`, which are executable code that launches `/bin/sh`. Once this shellcode runs, the machine is owned.

4.3 Automatic Exploit Generation

Manual exploit generation requires a developer to keep track of an enormous number of details. The size of the stack, the precise semantics of each instruction, and the exact addresses of control data are but a few example details. Our vision for AEG is to program a computer to take over the reasoning of exploiting bugs.

AEG uses verification techniques to transform the process of finding and deciding exploitability to reasoning in logic. At a high level, AEG first encodes what it means to exploit a program as a logical property. Second, AEG checks whether the exploitability property holds on a program. Third, for each path that the property holds, AEG produces a satisfying input that executes the path and exploits the program.

The three steps form the cornerstones of AEG research questions. First, what exploitability properties do we encode, and how? In industry, an exploit may mean control flow hijack, while an intelligence agency may also include information disclosures, and a safety board may include denial of service for critical services. Any single property may have many encodings, where some encodings are more efficient to check than others. Second, what techniques and algorithms should be employed to check a program? The general problem of checking programs for properties is called software model checking [82], and encompasses techniques such as bounded model checking, symbolic execution, and abstract interpretation. Third, what does it take to implement real systems, and how do those systems perform on real software? The theory of AEG can be succinctly described with a small number of operations on a well-defined programming language that interacts with its environment in a few predicable and easy-to-model ways. A real system, however, must contend with hundreds of different CPU instructions and the tricky and complex ways programs interact with its environment. Sometimes even pedestrian yet necessary details are hard to get right. For example, it took almost a year to stop finding bugs in our internal semantics for the x86

shift instructions (e.g., `shl`). Microsoft's SAGE tool reported a similar story for the same instructions [23].

Current AEG research primarily uses symbolic execution [16] to check program paths for exploitability properties. At a high level, symbolic execution picks a program path via a predefined path selection algorithm. The path is then "executed", except instead of providing a real, concrete input we supply a symbolic input that stands in for any possible concrete value. Symbolic execution builds up a path formula based upon the instructions executed. The path formula is satisfied (i.e., made true) by any concrete input that executes the desired path. If the path formula is unsatisfiable, then there is no input that executes the path and the path is called infeasible. The satisfiability check itself is done using SMT solvers [83]. By construction, free variables correspond to inputs, and any satisfying assignment of values to free variables (called a model) is an input that executes the selected path. SMT solvers can enumerate satisfying answers when needed.

In `acpi_listen`, the symbolic inputs are the first two arguments `argv[1]` and `argv[2]`. (Although we have shown source code for `acpi_listen` for clarity, our tools only require the program executable). Executing the `-s` option program path generates the constraint that the first 3 bytes of `argv[1]` correspond to the NULL-terminated string "`-s`". At each subsequent branch point, symbolic execution adds more constraints to the formula. Next, `acpi_listen` calls `sprintf`, which copies bytes from `name` to `addr.sun_path` until a NULL character is found. Symbolic execution captures this logic by adding the constraint that each copied byte is non-NULL. Symbolically executing the `-s` program path where `argv[1]` is 3 symbolic bytes and `argv[2]` is 140 non-NULL symbolic bytes generates the constraints (we

use the universal quantifier $\forall$ as a shorthand; our formulas are quantifier free):

$$\begin{aligned} &\text{argv}[1][0:2] = \text{“-s”} \wedge \\ &\forall i \in [0, 139].\, \text{argv}[2][i] \neq 0 \wedge \text{argv}[2][140] = 0 \end{aligned} \tag{4.1}$$

Note that a formula may have many satisfying answers, e.g., bytes 0–139 of `argv[2]` can be “A”, “B”, or any other non-NULL character.

Each feasible path is checked for exploitability by adding a set of constraints that are satisfied only by exploiting inputs. Most research tackles control flow hijack exploits, where the exploitability constraints specify: 1) the IP (Instruction Pointer) register holds a value that corresponds to some function of user input ι, and 2) the resulting IP points to shellcode. More specifically, let $tr(P, \iota)$ be a trace leading to a potential control flow hijack; and (Π, Γ) (the path predicate and context respectively; see Chapter 2 for notation) be the current symbolic execution state the exploitability constraints are:

$$\mathsf{load}(\Gamma[\mu], \Gamma[IP]) = \langle \text{shellcode} \rangle \tag{4.2}$$

Using the above formula in conjunction with the current path predicate, we get the exploitability check:

$$\Pi \wedge \mathsf{load}(\Gamma[\mu], \Gamma[IP]) = \langle \text{shellcode} \rangle \tag{4.3}$$

An assignment to input variables that satisfies the above constraints is—by construction—a control flow hijack exploit. The generated exploit can be automatically run and tested to ensure soundness.

Note that $\Gamma[IP]$ may be symbolic, meaning that the shellcode could potentially be positioned anywhere in memory. In our experiment, our AEG tool Mayhem [3] found the exploitable path and solved the exploitability formula in 0.5 seconds. Mayhem also has an option to enumerate satisfying answers to generate multiple exploits.

State Pruning with Preconditioned Symbolic Execution. The modeling above suffices to allow symbolic execution to detect control flow hijacks. However, augmenting a state-of-the-art symbolic executor (KLEE [21]) with that model was insufficient for identifying known exploitable bugs in the vast majority of programs we tested [2]. The state explosion problem was prohibiting the symbolic executor from exploring potentially exploitable paths (it is unlikely to find an exploitable state among the vast number of possibly irrelevant states). This observation was the motivation for preconditioned symbolic execution [2].

Preconditioned symbolic execution first performs lightweight analysis to determine the necessary conditions to exploit any lurking bugs, and then prunes the search space of paths that do not meet these conditions. For example, a lightweight program analysis may determine the minimum length to trigger any buffer overflow, which can then be used to prune symbolic execution paths corresponding to string inputs smaller than the minimum length. More concretely, to ensure that we only explore inputs longer than a threshold size $s_{\text{threshold}}$ we start symbolic execution with:

$$\Pi_{\text{init}} = (strlen(\iota) > s_{\text{threshold}})$$

as a precondition instead of $\Pi_{\text{init}} = true$. We also described how preconditioned symbolic execution can be used for fuzzing specific portions of the input, as well as how it can be integrated with existing fuzzers (how to check whether the crash found by fuzzer X is exploitable?). In our experiments [2] we found that pruning the state space had significant impact on exploitable bug detection, going from 1 to 16 exploitable bugs in 14 open source applications. Two of the identified exploitable bugs were against previously unknown vulnerabilities. We present our AEG system in depth in a follow up chapter (Chapter 5).

Exploit Generation, Defenses, and Attacks. One of the goals of our exploit generation research was to demonstrate attackers' capabilities. A bug marked as exploitable by AEG is under most environments exploitable. Even if global OS defenses are deployed to protect

against control hijacks such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), the bug remains likely exploitable due to return-oriented programming [84]. As a proof-of-concept we showed in Q [49] that it is possible to automatically take broken exploits that do not work against defenses and automatically upgrade them with symbolic execution so that they bypass both ASLR and DEP as currently implemented in modern operating systems (Windows 7, and Ubuntu 10.04).

Finally, ASLR and DEP only defend against memory overwrite attacks. Other vulnerabilities, such as information disclosure, denial of service, and command injection are also critical in practice. For example, DEP and ASLR offer no protection against the zero-day command injection exploit we found in a streaming audio player (`ezstream`). Since our original work on control flow hijack attacks, we have extended our system to check and report more classes of attacks (including crashes and command injections). While we present our work through the prism of control hijack attacks, we believe that our techniques are extensible to other areas and analyses.

4.3.1 Exploit Generation on Binaries and Memory Modeling

Our initial work on AEG was based on source code (Chapter 5). Using source code has benefits: analyses tend to scale better, types and datastructures are available, etc; but also drawbacks: we are unable to analyze programs without access to the source. To test arbitrary programs, we needed a tool that analyzes raw executable code. Executable analysis is widely applicable because everyone typically has access to the executable code of programs, thus can audit the code for security-critical bugs. Further, executable code analysis gives immediate access to low-level details necessary for exploits, such as memory layout and CPU semantics (obtaining such details was a non-trivial technical challenge at the source level [2]).

In 2010, we started designing a new symbolic executor called MAYHEM—presented in Chapter 6—for in-vivo analysis of binary programs. Among the many new technical

challenges at the binary level, a recurring one in AEG—that was exacerbated at the binary level—was the satisfiability of formulas that operate on memory with symbolic memory addresses. A symbolic memory address occurs when an index into an array or memory is based on user input, e.g.,

```
...;
y = memory[i % 256];
if(y == 2) vuln();
...
```

Without source code information the `memory` array could be 2^{32} cells long, and the SMT solver must do a case split to reason about all possible values of i that may reach downstream statements, e.g., `vuln`. The case splits can quickly cause an SMT solver to walk over an exponential cliff[2]. Symbolic memory references often crop up in commonly occurring library calls, e.g., conversion functions (e.g., `tolower`, `toascii`) and parsing functions (e.g., `sscanf`). Symbolic executors mitigate the case split either by concretizing symbolic addresses to an arbitrary value [12], e.g., by picking $i = 42$, or by forking a new state for every possible concretization [12, 29].

Unfortunately, concretization overconstrains formulas, and forking for every possible concretization leads to immediate state explosion. Thus, our initial AEG techniques missed 40% of known exploitable bugs in our test suite [3]. For example, AEG may need to craft an input that becomes valid shellcode after being processed by `tolower` (e.g., the mapping of a 32-bit instruction pointer in Equation 4.3 may be: $\Gamma[IP] =$ `tolower`$(\iota[0])$ :: `tolower`$(\iota[1])$ :: `tolower`$(\iota[2])$ :: `tolower`$(\iota[3])$, where $\iota[i]$ denotes the ith byte of input ι and `::` denotes concatenation). In MAYHEM, we proposed a number of optimizations for modeling symbolic

[2]There exist binary symbolic executors that handle memory fully symbolically, e.g., McVeto [31], but they are currently restricted to programs up to a few thousands of lines of code.

memory [3] that allow us to mitigate state explosion while keeping formulas concise. For example, one optimization performs a type of strength reduction where structured sequential symbolic memory accesses are encoded as piecewise linear equations [3]. We explore our memory modeling technique in depth in a follow up chapter (Chapter 6).

When we first started using symbolic execution and SMT solvers, we treated the SMT solver as a black box and focused only on the symbolic executor. In hindsight, that approach was naïve. We now believe it is more fruitful to view the SMT solver as a search procedure and use optimizations to guide the search and reduce/rearrange the size of the problem. This observation—along with a few others (Chapter 2)—led to the idea of veritesting (Chapter 7). In the original paper, we used MAYHEM to find and demonstrate 29 exploitable vulnerabilities in Windows and Linux programs, again exposing 2 new unknown vulnerabilities [3]. Since then, we have been working on extending and improving the MAYHEM executor, both in terms of technique, and scale (§7).

4.3.2 Example Application: Exploiting /usr/bin

Recall from §4.2 that we fuzzed `/usr/bin` on Debian and found 52 distinct bugs in 29 programs, including `acpi_listen`. One goal is to determine which bugs are exploitable.

We ran our binary-only AEG tool called Mayhem [3] on each crash to determine if we could automatically generate an exploit from the crashing path. We also manually checked whether it was possible to exploit the bug. 5 of the bugs are vulnerable to a control flow hijack, and Mayhem generated exploits for 4 of them. The exploit for `acpi_listen` took 0.5 seconds to generate, and the remaining three took 8, 12, and 28 seconds.

The above results on `/usr/bin` illustrate three points. First, current AEG tools like Mayhem are sound, but incomplete. A sound AEG technique only says a bug is exploitable if it really is exploitable, while a complete technique reports all exploitable bugs. Unfortunately, Rice's theorem states that checking any non-trivial program property in general is undecidable,

thus a sound and complete analysis is impossible in general. Second, AEG can be very fast when it succeeds. Third, there is ample room for improving AEG in particular, and symbolic execution and software model checking in general. For example, we analyzed why Mayhem failed on the last vulnerability, and found the problem was a single constraint that pushed the SMT solver we use (Z3) off an exponential cliff. Perhaps comically, manual analysis showed that the constraint was superfluous, but it was not recognized as such by the automatic formula optimizer. Once the constraint was removed from the formula, exploit generation succeeded almost immediately.

4.4 Real World Considerations

Security practitioners often only focus on exploits of programs on the attack surface of a system [85]. Roughly speaking, the attack surface consists of the set of programs, files, protocols, services, and other channels that are available to an attacker. Typical examples include network daemons, programs called from web servers on untrusted inputs, setuid programs, and media players. Our example `acpi_listen` is not on the attack surface. We chose `acpi_listen` because it highlights the steps of AEG, but disclosing the exploit will do little damage.

Overall, AEG techniques are independent of whether a program is on the attack surface or not. For example, over the course of writing this article, we ran Mayhem on additional examples that are on the attack surface. We found zero-day exploits for media applications (e.g., `ezstream` and `imview`) and network applications (e.g., `latd` and `ndtpd`).

Another consideration is additional layers of defense that may protect otherwise exploitable programs. Two popular OS-level defenses against control flow hijacks are Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP).

DEP marks memory pages either writable or executable, but forbids a memory page from being both. DEP prevents an attacker from writing and then executing shellcode, e.g., as in §4.3. Unfortunately, attackers have developed techniques for bypassing DEP. One technique is called a return-to-libc attack, where instead of an attacker writing new code to memory, they make use of code already present in memory, e.g., by running `system("/bin/sh")` in `libc` directly. Return-oriented programming (ROP) is a generalization of return-to-libc that uses instruction sequences already present in memory, called *gadgets*. Shacham *et al.* showed that it is possible to find a Turing-complete set of gadgets in `libc` [84].

ASLR prevents control flow hijacks by randomizing the location of objects in memory. Recall that in `acpi_listen` the attacker needed to know the address of shellcode. ASLR randomizes addresses so that vulnerable programs likely crash instead of successfully redirecting control to the shellcode. ASLR as currently deployed in Windows and Linux has several limitations that affect security. First, 32-bit architectures provide insufficient randomness for security [86]. 64-bit architectures will address this problem. Second, ASLR may not randomize all memory addresses, e.g., the code section of Linux executables is often not randomized. Third, an information disclosure vulnerability in a program may reveal post-randomization memory addresses, which can subsequently be used in an exploit.

Schwartz *et al.* proposed *exploit hardening*, which takes an exploit that works against an undefended system and hardens it to bypass defenses [49]. One aspect is to automatically generate ROP payloads (to bypass DEP) that take advantage of small portions of unrandomized memory (to bypass ASLR on the tested implementations of ASLR on Windows 7 and Linux). In particular, they showed (with high probability) ROP payloads can be generated given unrandomized code larger than `/bin/true`. Exploit hardening can be paired with AEG to check the end-to-end security of a program running on a specific system.

Finally, ASLR and DEP only defend against memory overwrite attacks. Other vulnerabilities, such as information disclosure, denial of service, and command injection are also critical

in practice. For example, DEP and ASLR do not protect against the zero-day command injection exploit found by Mayhem in `ezstream`.

4.5 Related Work

First Tools. Modern research in AEG itself dates back at least to 2005 with Ganapathy *et al.* [87], who explicitly connected verification to exploit generation. They modeled how format string specifiers are parsed by variadic functions like `printf`, and used the model to automatically generate exploits. They also demonstrated automatically generating an exploit against a key integrity property for a cryptographic co-processor [87]. However, they only considered API-level exploits, which does not include running shellcode nor the conditions necessary to reach a vulnerable API call site. In 2007, Medeiros [88] and Grenier *et al.* [89] proposed techniques based on pattern matching for AEG.

Patch-Based Exploit Generation. In 2008, Brumley *et al.* developed automatic *patch-based* exploit generation (APEG) [90]. The APEG challenge is: given a buggy program P and a patched version P', generate an exploit for the bug present in P but not present in P'. The idea is that the difference between P and P' reflects i) where the original bug occurs, and ii) under what conditions it may be triggered. Attackers have long known this, and routinely analyze patches to find non-public bugs. For example, attackers often joke Microsoft's "patch Tuesday" is followed by "exploit Wednesday". Our techniques automatically found the differences between P and P' and generated inputs that triggered the bugs in P using symbolic execution. One main security implication is that attackers can potentially use APEG to exploit bugs before patches can be distributed to a large number of users. We generated exploits for 5 Microsoft security patches, which included triggering an infinite loop in the TCP/IP driver and stealing files on Microsoft webservers. One limitation was our

work only proposed, but did not implement, techniques for executing shellcode for memory safety bugs [90, §6].

Generating Control Hijacks. Heelan's 2009 thesis work was the first to comprehensively describe and implement techniques for automatically generating control flow hijack exploits that execute shellcode [91]. In Heelan's problem setting, the attacker is given an input that executes an exploitable program path, and the goal is to output a working control flow hijack exploit. This setting is the same as in our running example where we first fuzzed to find bugs, and then checked exploitability. Heelan proposed using symbolic execution and taint analysis to derive the conditions necessary to transfer control to shellcode, and demonstrated a tool that produced exploits for several synthetic and one real vulnerabilities. His work also used a technique called return-to-register to improve exploit robustness. Heelan's thesis also presents a comprehensive history of AEG up through 2009.

Finding and Demonstrating Exploitability. In 2011, we proposed AEG techniques that both find bugs and generate exploits, and demonstrated our techniques on 16 vulnerabilities [2]. The initial work performed symbolic execution on source code to find bugs, and then used dynamic binary analysis to generate control flow hijack exploits. The model for detecting exploitability was expressed as a standard enforceable security policy [92]. The work proposed a number of optimizations for searching the state space, including preconditioned symbolic execution discussed in Chapter 5.

Bug Prioritization on Binary Code. In 2012, we proposed MAYHEM, a tool and a set of techniques for AEG given only executable code [3]. MAYHEM also proposed techniques for actively managing symbolically executed program paths without exhausting memory, and reasoning about symbolic memory addresses efficiently. Both papers target control flow hijacks for buffer overflows and format string vulnerabilities. MAYHEM generated exploits for 7 Windows and 22 Linux vulnerabilities. Disregarding one long-running outlier, the average

exploit generation time was 165 seconds. MAYHEM can currently generate exploits for buffer overflows, format strings, command injection, and some information leak vulnerabilities.

Bypassing Defenses. AEG [2] and MAYHEM [3] are designed to demonstrate a bug is exploitable, but do not try to bypass defenses that may otherwise protect a system. In 2011 we proposed techniques for bypassing the ASLR and DEP defenses as were implemented in Windows 7 and Linux, as well as exploit hardening and maintenance [49].

Follow-up Work. Exploit generation is still an actively researched area with more papers appearing every year. STING [93] is a symbolic execution system targeted at finding name resolution vulnerabilities. Vanegue *et al.* summarize several applications of SMT solvers in security, including AEG [94]. They argue that current approaches to AEG face a potentially large gap before being generally applicable in some real world problem instances, such as when the heap layout is not deterministic from the attackers point of view. Caselden *et al.* [95] proposed transformation-aware exploit generation for increased scalability. Several other systems have been presented for analyzing crashes [96], generating web exploits [97], search heuristics for finding buffer overflows [98], and customizing memory models for exploit generation [99].

4.6 Conclusion and Open Problems

AEG is still a young topic and far from a solved problem. Scalability will always be an open and interesting problem. Current AEG tools scale to finding buffer overflow exploits in programs the size of common Linux utilities. We would ultimately like to check much larger programs such as Google Chrome and Microsoft Word. Any significant scaling improvements is an important win since if AEG can scale, so can many other verification tasks. One promising data point is that related symbolic execution tools like SAGE routinely find

security-critical bugs in large applications [13], though it remains to be seen if those same bugs can also be automatically exploited.

Although AEG in theory should help offense, current results do not yet adequately demonstrate AEG in realistic offensive scenarios. For example, AEG tools have not yet exclusively focused on the attack surface, but instead report any control flow hijack as a potential exploit. Further, most exploits generated by existing tools are known exploitable bugs. AEG research still needs to prove it can find a large number of zero-day exploits.

More fundamentally, AEG needs to expand the formalism for exploitability to find a wider variety of exploitable bugs.[3] Integer overflows, use-after-free, heap overflows, and information flow problems seem important targets. Heap overflows in particular pose many challenges, one being the internal heap state is hard to know and model exactly, and another being new heap allocators (e.g., as in Windows 7 and 8) have certain provable guarantees against exploitation. In our experience, often the problem isn't coming up with *some* formalism, it's coming up with the right formalism that lends itself to efficient and practical analysis.

Except for a few examples, most work in AEG has focused on exploits in type-unsafe languages. However, it would be wrong to say programs in type-safe languages aren't exploitable. One problem is that the runtime environments for type-safe languages may be exploited, e.g., in Java vulnerabilities are becoming common. More fundamentally, exploitable bugs based on information flow, timing, and logic errors can be written in any language, even those that are type-safe.

Our overall message is that AEG is a verification task, and therefore the better we get at software verification, the better we get at automatically generating exploits. Just 7 years ago AEG techniques were restricted to analyzing a single API call. Today, AEG can both

[3] Although we advocate expanding to new types of exploits, recent vulnerability reports indicating the death of the basic buffer overflow are greatly exaggerated.

automatically find and generate exploits in common binaries. Advancements will continue to be fueled by better tools, techniques, and improvements in verification and security.

Part III

State Space Management

Chapter 5

State Pruning & Prioritization

Everything in good measure.

— Cleobulus, the Lindian.

The automatic exploit generation challenge is given a program, automatically find vulnerabilities and generate exploits for them. In this chapter we present AEG, the first end-to-end system for fully automatic exploit generation on source code. We used AEG to analyze 14 open-source projects and successfully generated 16 control flow hijacking exploits. Two of the generated exploits (expect-5.43 and htget-0.93) are zero-day exploits against unknown vulnerabilities. Our contributions are: 1) we show how exploit generation for control flow hijack attacks can be modeled as a formal verification problem at the source code level (building on top of Chapter 4), 2) we propose preconditioned symbolic execution, a novel technique for targeting symbolic execution, 3) we present a general approach for generating working exploits once a bug is found, and 4) we build the first end-to-end system that automatically finds vulnerabilities and generates exploits that produce a shell.

5.1 Introduction

Control flow exploits allow an attacker to execute arbitrary code on a computer. Current state-of-the-art in control flow exploit generation is for a human to think very hard about whether a bug can be exploited. Until now, automated exploit generation where bugs are automatically found and exploits are generated has not been shown practical against real programs.

In this chapter, we develop novel techniques and an end-to-end system for automatic exploit generation (AEG) on real programs. In our setting, we are given the potentially buggy program in source form. Our AEG techniques find bugs, determine whether the bug is exploitable, and, if so, produce a working control flow hijack exploit string. The exploit string can be directly fed into the vulnerable application to get a shell. We have analyzed 14 open-source projects and successfully generated 16 control flow hijacking exploits, including two zero-day exploits for previously unknown vulnerabilities.

Our automatic exploit generation techniques have several immediate security implications. First, practical AEG fundamentally changes the perceived capabilities of attackers. For example, previously it has been believed that it is relatively difficult for untrained attackers to find novel vulnerabilities and create zero-day exploits. Our research shows this assumption is unfounded. Understanding the capabilities of attackers informs what defenses are appropriate. Second, practical AEG has applications to defense. For example, automated signature generation algorithms take as input a set of exploits, and output an IDS signature (*aka* an input filter) that recognizes subsequent exploits and exploit variants [22, 100]. Automated exploit generation can be fed into signature generation algorithms by defenders without requiring real-life attacks.

Challenges. There are several challenges we address to make AEG practical:

A. Source code analysis alone is inadequate and insufficient. Source code analysis is insufficient to report whether a potential bug is exploitable because errors are found with respect to source code level abstractions. Control flow exploits, however, must reason about binary and runtime-level details, such as stack frames, memory addresses, variable placement and allocation, and many other details unavailable at the source code level. For instance, consider the following code excerpt:

```
1    char src[12], dst[10];
2    strncpy(dst, src, sizeof(src));
```

In this example, we have a classic buffer overflow where a larger buffer (12 bytes) is copied into a smaller buffer (10 bytes). While such a statement is clearly wrong [1] and would be reported as a bug at the source code level, in practice this bug would likely not be exploitable. Modern compilers would page-align the declared buffers, resulting in both data structures getting 16 bytes. Since the destination buffer would be 16 bytes, the 12-byte copy would not be problematic and the bug not exploitable.

While source code analysis is insufficient, binary-level analysis is unscalable. Source code has abstractions, such as variables, buffers, functions, and user-constructed types that make automated reasoning easier and more scalable. No such abstractions exist at the binary-level; there only stack frames, registers, gotos and a globally addressed memory region.

In our approach, we combine source-code level analysis to improve scalability in finding bugs and binary and runtime information to exploit programs. To the best of our knowledge, we are the first to combine analysis from these two very different code abstraction levels.

B. Finding the exploitable paths among an infinite number of possible paths. Our techniques for AEG employ symbolic execution, a formal verification technique that explores program paths and checks if each path is exploitable. Programs have loops, which in turn means that

[1] Technically, the C99 standard would say the program exhibits undefined behavior at this point.

they have a potentially infinite number of paths. However, not all paths are equally likely to be exploitable. Which paths should we check first?

Our main focus is to detect exploitable bugs. Our results show (§ 5.8) that existing state-of-the-art solutions proved insufficient to detect such security-critical bugs in real-world programs.

To address the path selection challenge, we developed two novel contributions in AEG. First, we have developed *preconditioned symbolic execution*, a novel technique which targets paths that are more likely to be exploitable. For example, one choice is to explore only paths with the maximum input length, or paths related to HTTP GET requests. While preconditioned symbolic execution eliminates some paths, we still need to prioritize which paths we should explore first. To address this challenge, we have developed a priority queue *path prioritization* technique that uses heuristics to choose likely more exploitable paths first. For example, we have found that if a programmer makes a mistake—not necessarily exploitable—along a path, then it makes sense to prioritize further exploration of the path since it is more likely to eventually lead to an exploitable condition.

C. An end-to-end system. We provide the first practical end-to-end system for AEG on real programs. An end-to-end system requires not only addressing a tremendous number of scientific questions, e.g., binary program analysis and efficient formal verification, but also a tremendous number of engineering issues. Our AEG implementation is a single command line that analyzes source code programs, generates symbolic execution formulas, solves them, performs binary analysis, generates binary-level runtime constraints, and formats the output as an actual exploit string that can be fed directly into the vulnerable program.

Scope. While, in this chapter, we make exploits robust against local environment changes, our goal is not to make exploits robust against common security defenses, such as address space randomization [86] and $w \oplus x$ memory pages (e.g., Windows DEP). In this work, we always require source code. AEG on binary-only is left as future work. We also do not claim

```
1 int main(int argc, char **argv) {
2   int skfd;              /* generic raw socket desc.
      */
3   if(argc == 2)
4     print_info(skfd, argv[1], NULL, 0);
5 ...
6 static int print_info(int skfd, char *ifname, char *args
    [], int count) {
7   struct wireless_info  info;
8   int                   rc;
9   rc = get_info(skfd, ifname, &info);
10 ...
11 static int get_info(int skfd, char *ifname, struct
     wireless_info * info) {
12   struct iwreq          wrq;
13   if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0) {
14       struct ifreq ifr;
15       strcpy(ifr.ifr_name, ifname); /* buffer overflow */
16 ...
```

Figure 5.1: Code snippet from Wireless Tools' `iwconfig`.

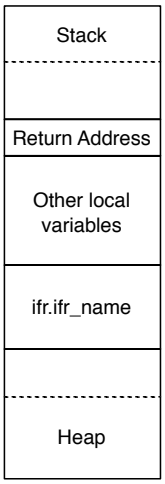


Figure 5.2: Memory Diagram

```
00000000  02 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000010  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000020  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000030  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000040  01 01 01 01 70 f3 ff bf  31 c0 50 68 2f 2f 73 68  |....p...1.Ph//sh|
00000050  68 2f 62 69 6e 89 e3 50  53 89 e1 31 d2 b0 0b cd  |h/bin..PS..1....|
00000060  80 01 01 01 00                                    |.....|
```

Figure 5.3: A generated exploit of `iwconfig` from AEG.

AEG is a "solved" problem; there is always opportunity to improve performance, scalability, to work on a larger variety of exploit classes, and to work in new application settings.

5.2 Overview of AEG

This section explains how AEG works by stepping through the entire process of bug-finding and exploit generation on a real world example. The target application is the setuid root `iwconfig` utility from the `Wireless Tools` package (version 26), a program consisting of about 3400 lines of C source code.

Before AEG starts the analysis, there are two necessary preprocessing steps: 1) We build the project with the GNU C Compiler (GCC) to create the binary we want to exploit, and 2) with the LLVM [101] compiler—to produce bytecode that our bug-finding infrastructure uses for analysis. After the build, we run our tool, AEG, and get a control flow hijacking exploit in less than 1 second. Providing the exploit string to the `iwconfig` binary, as the 1st argument, results in a root shell.

Figure 5.1 shows the code snippet that is relevant to the generated exploit. `iwconfig` has a classic `strcpy` buffer overflow vulnerability in the `get_info` function (line 15), which AEG spots and exploits automatically in less than 1 second. To do so, our system goes through the following analysis steps:

1. AEG searches for bugs at the source code level by exploring execution paths. Specifically, AEG executes `iwconfig` using symbolic arguments (`argv`) as the input sources. AEG considers a variety of input sources, such as files, arguments, etc., by default.

2. After following the path `main` → `print_info` → `get_info`, AEG reaches line 15, where it detects an out-of-bounds memory error on variable `ifr.ifr_name`. AEG solves the current path constraints and generates a concrete input that will trigger the detected bug, e.g., the first argument has to be over 32 bytes.

3. AEG performs dynamic analysis on the `iwconfig` binary using the concrete input generated in step 2. It extracts runtime information about the memory layout, such as the address of the overflowed buffer (`ifr.ifr_name`) and the address of the return address of the vulnerable function (`get_info`).

4. AEG generates the constraints describing the exploit using the runtime information generated from the previous step: 1) the vulnerable buffer (`ifr.ifr_name`) must contain our shellcode, and 2) the overwritten return address must contain the address of the

shellcode—available from runtime. Next, AEG appends the generated constraints to the path constraints and queries a constraint solver for a satisfying answer.

5. The satisfying answer gives us the exploit string, shown in Figure 5.3. Finally, AEG runs the program with the generated exploit and verifies that it works, i.e., spawns a shell. If the constraints were not solvable, AEG would resume searching the program for the next potential vulnerability.

Challenges. The above walkthrough illustrates a number of challenges that AEG has to address:

- The *State Space Explosion* problem (Steps 1-2). There are potentially an infinite number of paths that AEG has to explore until an exploitable path is detected. AEG utilizes preconditioned symbolic execution (see § 5.5.2) to target exploitable paths.
- The *Path Selection* problem (Steps 1-2). Amongst an infinite number of paths, AEG has to select which paths should be explored first. To do so, AEG uses path prioritization techniques (see § 5.5.3).
- The *Environment Modelling* problem (Steps 1-3). Real-world applications interact intensively with the underlying environment. To enable accurate analysis on such programs AEG has to model the environment IO behavior, including command-line arguments, files and network packets (see § 5.5.4).
- The *Mixed Analysis* challenge (Steps 1-4). AEG performs a mix of binary- and source-level analysis in order to scale to larger programs than could be handled with a binary-only approach. Combining the analyses' results of such fundamentally different levels of abstraction presents a challenge on its own (see § 5.6.2).
- The *Exploit Verification* problem (Step 5). Last, AEG has to verify that the generated exploit is a *working* exploit for a given system (see § 5.6.3).

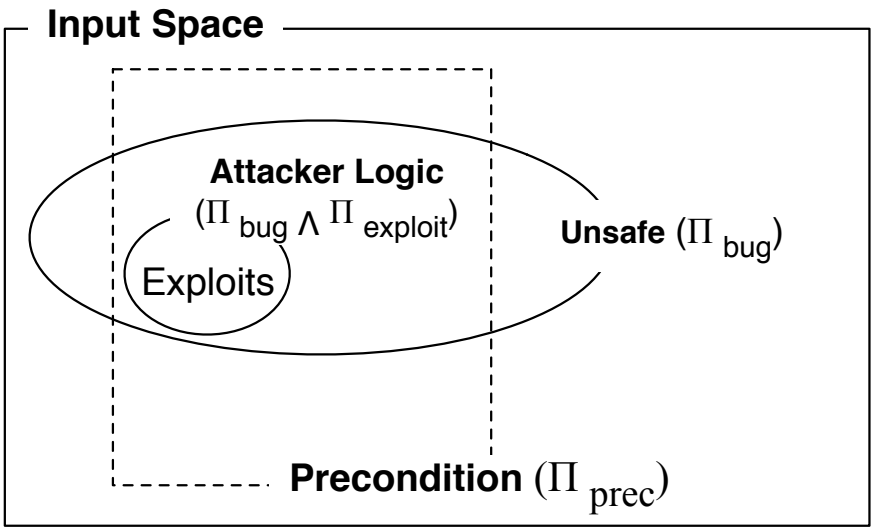


Figure 5.4: The input space diagram shows the relationship between unsafe inputs and exploits. Preconditioned symbolic execution narrows down the search space to inputs that satisfy the precondition (Π_{prec}).

5.3 The AEG Challenge

At its core, the automatic exploit generation (AEG) challenge is a problem of finding program inputs that result in a desired exploited execution state. In this section, we show how the AEG challenge can be phrased as a formal verification problem, as well as propose a new symbolic execution technique that allows AEG to scale to larger programs than previous techniques. As a result, this formulation: 1) enables formal verification techniques to produce exploits, and 2) allows AEG to directly benefit from any advances in formal verification.

5.3.1 Problem Definition

In this chapter, we focus on generating a control flow hijack exploit input that intuitively accomplishes two things. First, the exploit should violate program safety, e.g., cause the program to write to out-of-bound memory. Second, the exploit must redirect control flow to the attacker's logic, e.g., by executing injected shellcode, performing a return-to-libc attack, and others.

At a high level, our approach uses program verification techniques where we verify that the program is exploitable (as opposed to traditional verification that verifies the program is safe). The exploited state is characterized by two Boolean predicates: a buggy execution path predicate Π_{bug} and a control flow hijack exploit predicate Π_{exploit}, specifying the control hijack and the code injection attack. The Π_{bug} predicate is satisfied when a program violates the semantics of program safety. However, simply violating safety is typically not enough. In addition, Π_{exploit} captures the conditions needed to hijack control of the program.

An exploit in our approach is an input ϵ that satisfies the Boolean equation:

$$\Pi_{\text{bug}}(\epsilon) \ \wedge \ \Pi_{\text{exploit}}(\epsilon) \ = \ \texttt{true} \tag{5.1}$$

Using this formulation, the mechanics of AEG is to check at each step of the execution whether Equation 5.1 is satisfiable. Any satisfying answer is, by construction, a control flow hijack exploit. We discuss these two predicates in more detail below.

The Unsafe Path Predicate Π_{bug}. Π_{bug} represents the path predicate of an execution that violates the safety property ϕ. In our implementation, we use popular well-known safety properties for C programs, such as checking for out-of-bounds writes, unsafe format strings, etc. The unsafe path predicate Π_{bug} partitions the input space into inputs that satisfy the predicate (unsafe), and inputs that do not (safe). While path predicates are sufficient to describe bugs at the source-code level, in AEG they are necessary but insufficient to describe the very specific actions we wish to take, e.g., execute shellcode.

The Exploit Predicate Π_{exploit}. The exploit predicate specifies the attacker's logic that the attacker wants to do after hijacking `eip`. For example, if the attacker only wants to crash the program, the predicate can be as simple as "set `eip` to an invalid address after we gain control". In our experiments, the attacker's goal is to get a shell. Therefore, Π_{exploit} must specify that the shellcode is well-formed in memory, and that `eip` will transfer control to it. The conjunction of the exploit predicate (Π_{exploit}) will induce constraints on the final

solution. If the final constraints (from Equation 5.1) are not met, we consider the bug as non-exploitable (§5.6.2).

5.3.2 Scaling with Preconditioned Symbolic Execution

Our formulation allows us to use formal verification techniques to generate exploits. While this means formal verification can be used for AEG, existing techniques such as model checking, weakest preconditions, and forward symbolic verification unfortunately only scale to small programs. For example, KLEE is a state-of-the-art forward symbolic execution engine [21], but in practice is limited to small programs such as /bin/ls. In our experiments, KLEE was able to find only 1 of the bugs we exploited (§ 5.8).

We observe that one reason scalability is limited with existing verification techniques is that they prove the absence of bugs by considering the entire program state space. For example, when KLEE explores a program for buffer overflows it considers all possible input lengths up to some maximum size, i.e., inputs of length 0, inputs of length 1, and so on. We observe that we can scale AEG by restricting the state space to only include states that are likely exploitable, e.g., by considering only inputs of a minimum length needed to overwrite any buffer. We achieve this by performing low-cost analysis to determine the minimum length ahead of time, which allows us to prune off the state space search during the (more expensive) verification step.

We propose *preconditioned symbolic execution* as a verification technique for pruning off portions of the state space that are uninteresting. Preconditioned symbolic execution is similar to forward symbolic execution [16] in that it incrementally explores the state space to find bugs. However, preconditioned symbolic execution takes in an additional Π_{prec} parameter. Preconditioned symbolic execution only descends into program branches that satisfy Π_{prec},

with the net effect that subsequent steps of unsatisfied branches are pruned away. [2] In AEG, we use preconditioned symbolic execution to restrict exploration to only likely-exploitable regions of the state space. For example, for buffer overflows Π_{prec} is specified via lightweight program analysis that determines the minimum sized input to overflow any buffer.

Figure 5.4 depicts the differences visually. Typical verification explores the entire input state space, as represented by the overall box, with the goal of finding inputs that are unsafe and satisfy Π_{bug}. In AEG, we are only concerned with the subset of unsafe states that are exploitable, represented by the circle labeled $\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$. The intuition is that preconditioned symbolic execution limits the space searched to a smaller box.

Logically, we would be guaranteed to find all possible exploits when Π_{prec} is less restrictive than the exploitability condition:

$$\Pi_{\text{bug}}(x) \wedge \Pi_{\text{exploit}}(x) \Rightarrow \Pi_{\text{prec}}(x)$$

In practice, this restriction can be eased to narrow the search space even further, at the expense of possibly missing some exploits. We explore several possibilities in § 5.5.2, and empirically evaluate their effectiveness in § 5.8.

5.4 Our Approach

In this section, we give an overview of the components of AEG, our system for automatic exploit generation. Figure 5.5 shows the overall flow of generating an exploit in AEG. Our approach to the AEG challenge consists of six components: Pre-Process, Src-Analysis, Bug-Find, DBA [3], Exploit-Gen, and Verify.

[2] Note preconditioned forward symbolic execution is different than weakest preconditions. Weakest preconditions statically calculate the weakest precondition to achieve a desired post-condition. Here we dynamically check a not-necessarily weakest precondition for pruning.

[3] Dynamic Binary Analysis

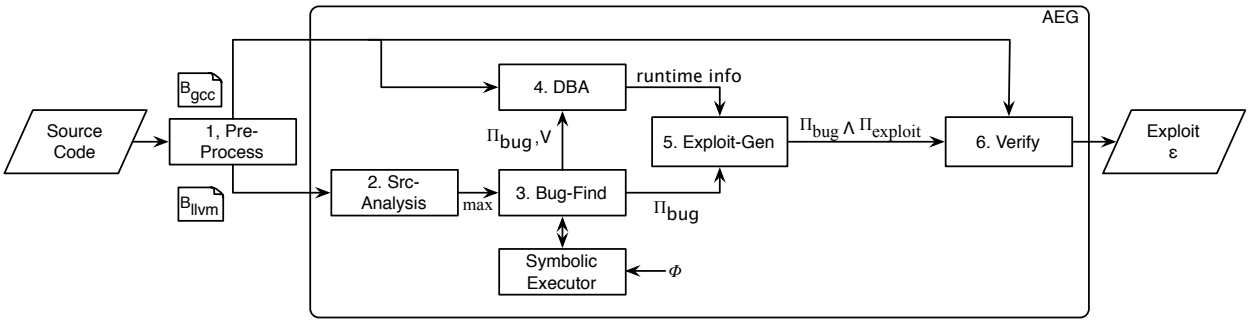


Figure 5.5: AEG design.

Pre-Process: src $\rightarrow (B_{\text{gcc}}, B_{\text{llvm}})$.

AEG is a two-input single-output system: the user provides the target binary and the LLVM bytecode of the same program, and—if AEG succeeds—we get back a working exploit for the given binary. Before the program analysis part begins, there is a necessary manual preprocessing step: the source program (*src*) is compiled down to 1) a binary B_{gcc}, for which AEG will try to generate a working exploit and 2) a LLVM bytecode file B_{llvm}, which will be used by our bug finding infrastructure.

Src-Analysis: $B_{\text{llvm}} \rightarrow max$.

AEG analyzes the source code to generate the maximum size of symbolic data *max* that should be provided to the program. AEG determines *max* by searching for the largest statically allocated buffers of the target program. AEG uses the heuristic that *max* should be at least 10% larger than the largest buffer size.

Bug-Find $(B_{\text{llvm}}, \phi, max) \rightarrow (\Pi_{\text{bug}}, V)$.

BUG-FIND takes in LLVM bytecode B_{llvm} and a safety property ϕ, and outputs a tuple $\langle \Pi_{\text{bug}}, V \rangle$ for each detected vulnerability. Π_{bug} contains the path predicate, i.e., the conjunction of all path constraints up to the violation of the safety property ϕ. V contains source-level information about the detected vulnerability, such as the name of the object being overwritten, and the vulnerable function. To generate the path

constraints, AEG uses a symbolic executor. The symbolic executor reports a bug to AEG whenever there is a violation of the ϕ property. AEG utilizes several novel bug-finding techniques to detect exploitable bugs (see § 5.5).

DBA: $(B_{\text{gcc}}, (\Pi_{\text{bug}}, V)) \rightarrow R$.

DBA performs dynamic binary analysis on the target binary B_{gcc} with a concrete buggy input and extracts runtime information R. The concrete input is generated by solving the path constraints Π_{bug}. While executing the vulnerable function (specified in V at the source-code level), DBA examines the binary to extract low-level runtime information (R), such as the vulnerable buffer's address on the stack, the address of the vulnerable function's return address, and the stack memory contents just before the vulnerability is triggered. DBA has to ensure that all the data gathered during this stage are accurate, since AEG relies on them to generate working exploits (see § 5.6.1).

Exploit-Gen: $(\Pi_{\text{bug}}, R) \rightarrow \Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$.

Exploit-Gen receives a tuple with the path predicate of the bug (Π_{bug}) and runtime information (R), and constructs a formula for a control flow hijack exploit. The output formula includes constraints ensuring that: 1) a possible program counter points to a user-determined location, and 2) the location contains shellcode (specifying the attacker's logic Π_{exploit}). The resulting exploit formula is the conjunction of the two predicates (see § 5.6.2).

Verify: $(B_{\text{gcc}}, \Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}) \rightarrow \{\epsilon, \perp\}$.

Verify takes in the target binary executable B_{gcc} and an exploit formula $\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$, and returns an exploit ϵ only if there is a satisfying answer. Otherwise, it returns $\perp$. In our implementation, AEG performs an additional step in Verify: runs the binary B_{gcc} with ϵ as an input, and checks if the adversarial goal is satisfied or not, i.e., if the program spawns a shell (see § 5.6.3).

Algorithm 1 shows our high-level algorithm for solving the AEG challenge.

Algorithm 1: Our AEG exploit generation algorithm

input : src: the program's source code
output : $\{\epsilon, \perp\}$: a working exploit or $\perp$

```
1 (B_gcc, B_llvm) = Pre-Process(src);
2 max = Src-Analysis(B_llvm);
3 while (Π_bug, V) = Bug-Find(B_llvm, φ, max) do
4     R = DBA(B_gcc, (Π_bug, V)) ;
5     Π_bug ∧ Π_exploit = Exploit-Gen(Π_bug, R) ;
6     ε = Verify(B_gcc, Π_bug ∧ Π_exploit);
7     if ε ≠ ⊥ then
8         return ε;
9 return ⊥;
```

5.5 Bug-Find: Program Analysis for Exploit Generation

BUG-FIND takes as input the target program in LLVM bytecode form, checks for bugs, and for each bug found attempts the remaining exploit generation steps until it succeeds. BUG-FIND finds bugs with symbolic program execution, which explores the program state space one path at a time. However, there are an infinite number of paths to potentially explore. AEG addresses this problem with two novel algorithms. First, we present a novel technique called preconditioned symbolic execution that constrains the paths considered to those that would most likely include exploitable bugs. Second, we propose novel path prioritization heuristics for choosing which paths to explore first with preconditioned symbolic execution.

5.5.1 Traditional Symbolic Execution for Bug Finding

At a high level, symbolic execution is conceptually similar to normal concrete execution except that we provide a fresh symbolic variable instead of providing a concrete value for inputs. As the program executes, each step of symbolic execution builds up an expression by substituting symbolic inputs for terms of the program. At program branches, the interpreter conceptually "forks off" two interpreters, adding the true branch guard to the conditions for the true branch interpreter, and similarly for the false branch. The conditions imposed as the interpreter executes are called the *path predicate* to execute the given path. After forking, the interpreter checks if the path predicate is satisfiable by querying a decision procedure. If not, the path is not realizable by any input, so the interpreter exits. If the path predicate can be satisfied, the interpreter continues executing and exploring the program state space. A more precise semantics can be found in Chapter 2.

Symbolic execution is used to find bugs by adding safety checks using ϕ. For example, whenever we access a buffer using a pointer, the interpreter needs to ensure the pointer is within the bounds of the buffer. The bounds-check returns either true, meaning the safety property holds, or false, meaning there is a violation, thus a bug. Whenever a safety violation is detected, symbolic execution stops and the current buggy path predicate (Π_{bug}) is reported.

5.5.2 Preconditioned Symbolic Execution

The main challenge with symbolic execution (and other verification techniques) is managing the state space explosion problem. Since symbolic execution forks off a new interpreter at every branch, the total number of interpreters is exponential in the number of branches.

We propose *preconditioned symbolic execution* as a novel method to target symbolic execution towards a certain subset of the input state space (shown in Figure 5.4). The state space subset is determined by the precondition predicate (Π_{prec}); inputs that do not satisfy

```
1 int process_input(char input[42])
2     char buf[20];
3     while(input[i] != '\0')
4         buf[i++] = input[i];
```

Figure 5.6: Tight symbolic loops. A common pattern for most buffer overflows.

Π_{prec} will not be explored. The intuition for preconditioned symbolic execution is that we can narrow down the state space we are exploring by specifying exploitability conditions as a precondition, e.g., all symbolic inputs should have the maximum size to trigger buffer overflow bugs. The main benefit from preconditioned symbolic execution is simple: by limiting the size of the input state space before symbolic execution begins, we can prune program paths and therefore explore the target program more efficiently.

Note that preconditions cannot be selected at random. If a precondition is too specific, we will detect no exploits (since exploitability will probably not imply the precondition); if it is too general, we will have to explore almost the entire state space. Thus, preconditions have to describe common characteristics among exploits (to capture as many as possible) and at the same time it should eliminate a significant portion of non-exploitable inputs.

Preconditioned symbolic execution enforces the precondition by adding the precondition constraints to the path predicate during initialization. Adding constraints may seem strange since there are more checks to perform at branch points during symbolic execution. However, the shrinking of the state space—imposed by the precondition constraints—outweighs the decision procedure overhead at branching points. When the precondition for a branch is unsatisfiable, we do no further checks and do not fork off an interpreter at all for the branch. We note that while we focus only on exploitable paths, the overall technique is more generally applicable.

The advantages of preconditioned symbolic execution are best demonstrated via example. Consider the program shown in Figure 5.6. Suppose that the *input* buffer contains 42

symbolic bytes. Lines 4-5 represent a tight symbolic loop—equivalent to a `strcpy`—that will eventually spawn 42 different interpreters with traditional symbolic execution, each one having a different path predicate. The 1st interpreter will not execute the loop and will assume that $(input[0] = 0)$, the 2nd interpreter will execute the loop once and assume that $(input[0] \neq 0) \wedge (input[1] = 0)$, and so on. Thus, each path predicate will describe a different condition about the *string length* of the symbolic *input* buffer. [4]

Preconditioned symbolic execution avoids examining the loop iterations that will not lead to a buffer overflow by imposing a *length precondition*:

$$L = \forall_{i=0}^{i<n}(input[i] \neq 0) \wedge (input[n] = 0)$$

This predicate is appended to the path predicate (Π) before we start the symbolic execution of the program, thus eliminating paths that do not satisfy the precondition. In our previous example (Figure 5.6), the executor performs the followings checks every time we reach the loop branch point:

$$\text{false branch:} \quad \Pi \wedge L \Rightarrow input[i] = 0, \text{ pruned } \forall i < n$$

$$\text{true branch:} \quad \Pi \wedge L \Rightarrow input[i] \neq 0, \text{ satisfiable } \forall i < n$$

Both checks are very fast to perform, since the validity (or invalidity) of the branch condition can be immediately determined by the precondition constraints L (in fact, in this specific example there is no need for a solver query, since validity or invalidity can be determined by a simple iteration through our assumption set $\Pi \wedge L$). Thus, by applying the length precondition we only need a single interpreter to explore the entire loop. In the rest of the section, we show how we can generate different types of preconditions to reduce the search space.

[4] The length precondition for strings is generated based on a null character, because all strings are null-terminated.

5.5.2.1 Preconditions

In AEG, we have developed and implemented 4 different preconditions for efficient exploit generation:

None There is no precondition and the state space is explored as normal.

Known Length The precondition is that inputs are of known maximum length, as in the previous example. We use static analysis to automatically determine this precondition.

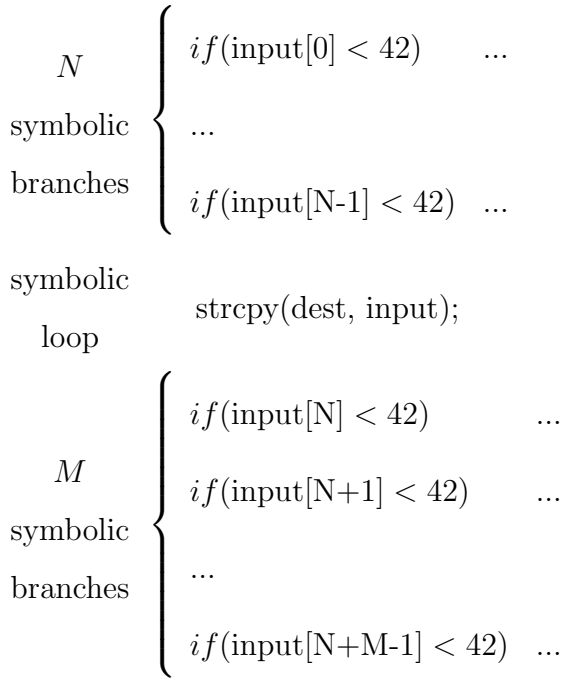
Known Prefix The precondition is that the symbolic inputs have a known prefix.

Concolic Execution Concolic execution [30, 12] can be viewed as a specific form of preconditioned symbolic execution where the precondition is specified by a single program path as realized by an example input. For example, we may already have an input that crashes the program, and we use it as a precondition to determine if the executed path is exploitable.

The above preconditions assume varying amounts of static analysis or user input. In the following, we further discuss these preconditions, and also describe the reduction in the state space that preconditioned symbolic execution offers. A summary of the preconditions' effect on branching is shown in Figure 5.7.

None. Preconditioned symbolic execution is equivalent to standard symbolic execution. The input precondition is `true` (the entire state space). *Input Space*: For S symbolic input bytes, the size of the input space is 256^S. The example in Figure 5.7 contains $N+M$ symbolic branches and a symbolic loop with S maximum iterations, thus in the worst case (without pruning), we need $2^N \cdot S \cdot 2^M$ interpreters to explore the state space.

Known Length. The precondition is that all inputs should be of maximum length. For example, if the input data is of type string, we add the precondition that each byte of input



(a) An example that illustrates the advantages of preconditioned symbolic execution.

Precondition	Input Space	# of Interpreters
None	256^S	$2^N \cdot S \cdot 2^M$
Known Length	255^S	$2^N \cdot 2^M$
Known Prefix	256^{S-P}	$2^{N-P}(S-P)2^M$
Concolic	1	1

(b) The size of the input space and the number of interpreters required to explore the state space of the example program at the left, for each of the 4 preconditions supported by AEG. We use S to denote the number of symbolic input bytes and P for the length of the known prefix ($P < N < S$).

Figure 5.7: A preconditioned symbolic execution example.

up to the maximum input length is not NULL, i.e., $(\texttt{strlen}(input) = len)$ or equivalently in logic $(input[0] \neq 0) \wedge (input[1] \neq 0) \wedge \ldots \wedge (input[len-1] \neq 0) \wedge (input[len] = 0)$. *Input space*: The input space of a string of length len will be 255^{len}. Note that for $len = S$, this means a 0.4% decrease of the input space for each byte. *Savings*: The length precondition does not affect the $N + M$ symbolic branches of the example in Figure 5.7. However, the symbolic `strcpy` will be converted into a straight-line concrete copy —since we know the length and pruning is enabled, we need not consider copying strings of all possible lengths. Thus, we need 2^{N+M} interpreters to explore the entire state space. Overall, the length precondition decreases the input space slightly, but can concretize `strcpy`-like loops—a common pattern for detecting buffer overflows.

Known Prefix. The precondition constrains a prefix on input bytes, e.g., an HTTP GET request always starts with "GET", or that a specific header field needs to be within a certain range of values, e.g., the protocol field in the IP header. We use a prefix precondition to target our search towards inputs that start with that specific prefix. For example, suppose that we wish to explore only PNG images on an image-processing utility. The PNG standard specifies that all images must start with a standard 8-byte header PNG_H, thus simply by specifying a prefix precondition $(input[0] = PNG\_H[0]) \wedge \ \ldots \ \wedge (input[7] = PNG\_H[7])$, we can focus our search to PNG images alone. Note that prefix preconditions need not only consist of exact equalities; they can also specify a range or an enumeration of values for the symbolic bytes.

Input space: For S symbolic bytes and an exact prefix of length P ($P < N < S$), the size of the input space will be 256^{S-P}. *Savings*: For the example shown in Figure 5.7, the prefix precondition effectively concretizes the first P branches as well as the first P iterations of the symbolic `strcpy`, thus reducing the number of required interpreters to $S \cdot 2^{N+M-P}$. A prefix precondition can have a radical effect on the state space, but is no panacea. For example, by considering only valid prefixes we are potentially missing exploits caused by malformed headers.

Concolic Execution. The dual of specifying no precondition is specifying the precondition that all input bytes have a specific value. Specifying all input bytes have a specific value is equivalent to concolic execution [30, 12]. Mathematically, we specify $\forall_i$: $\bigwedge (input[i] = concrete\_input[i])$.

Input Space: There is a single concrete input. *Savings*: A single interpreter is needed to explore the program, and because of state pruning, we are concretely executing the execution path for the given input. Thus, especially for concolic execution, it is much more useful to disable state pruning and drop the precondition constraints whenever we fork a new interpreter. Note that, in this case, AEG behaves as a concolic fuzzer, where the concrete

constraints describe the initial seed. Even though concolic execution seems to be the most constrained of all methods, it can be very useful in practice. For instance, an attacker may already have a proof-of-concept (PoC—an input that crashes the program) but cannot create a working exploit. AEG can take that PoC as a seed and generate an exploit—as long as the program is exploitable with any of the AEG-supported exploitation techniques.

5.5.3 Path Prioritization: Search Heuristics

Preconditioned symbolic execution limits the search space. However, within the search space, there is still the question of *path prioritization*: which paths should be explored first? AEG addresses this problem with path-ranking heuristics. All pending paths are inserted into a priority queue based on their ranking, and the next path to explore is always drawn out of the priority queue. In this section, we present two new path prioritization heuristics we have developed: *buggy-path-first* and *loop exhaustion*.

Buggy-Path-First. Exploitable bugs are often preceded by small but unexploitable mistakes. For example, in our experiments we found errors where a program first has an off-by-one error in the amount of memory allocated for a `strcpy`. While the off-by-one error could not directly be exploited, it demonstrated that the programmer did not have a good grasp of buffer bounds. Eventually, the length misunderstanding was used in another statement further down the path that was exploitable. The observation that one bug on a path means subsequent statements are also likely to be buggy (and hopefully exploitable) led us to the buggy-path-first heuristic. Instead of simply reporting the first bug and stopping like other tools such as KLEE [21], the buggy-path-first heuristic prioritizes buggy paths higher and continues exploration.

Loop Exhaustion. Loops whose exit condition depends on symbolic input may spawn a tremendous amount of interpreters—even when using preconditioned symbolic execution

techniques such as specifying a maximum length. Most symbolic execution approaches mitigate this program by de-prioritizing subsequent loop executions or only considering loops a small finite number of times, e.g., up to 3 iterations. While traditional loop-handling strategies are excellent when the main goal is maximizing code coverage, they often miss exploitable states. For example, the perennial exploitable bug is a `strcpy` buffer overflow, where the `strcpy` is essentially a while loop that executes as long as the source buffer is not NULL. Typical buffer sizes are quite large, e.g., 512 bytes, which means we must execute the loops at least that many times to create an exploit. Traditional approaches that limit loops simply miss these bugs.

We propose and use a *loop exhaustion* search strategy. The loop-exhaustion strategy gives higher priority to an interpreter exploring the maximum number of loop iterations, hoping that computations involving more iterations are more promising to produce bugs like buffer overflows. Thus, whenever execution hits a symbolic loop, we try to *exhaust* the loop—execute it as many times as possible. Exhausting a symbolic loop has two immediate side effects: 1) on each loop iteration a new interpreter is spawned, effectively causing an explosion in the state space, and 2) execution might get "stuck" in a deep loop. To avoid getting stuck, we impose two additional heuristics during loop exhaustion: 1) we use preconditioned symbolic execution along with pruning to reduce the number of interpreters or 2) we give higher priority to only one interpreter that tries to fully exhaust the loop, while all other interpreters exploring the same loop have the lowest possible priority.

5.5.4 Environment Modelling: Vulnerability Detection in the Real World

AEG models most of the system environments that an attacker can possibly use as an input source. Therefore, AEG can detect most security relevant bugs in real programs. Our

support for environment modeling includes file systems, network sockets, standard input, program arguments, and environment variables. Additionally, AEG handles most common system and library function calls.

Symbolic Files. AEG employs an approach similar to KLEE's [21] for symbolic files: modeling the fundamental system call functions, such as open, read, and write. AEG simplifies KLEE's file system models to speedup the analysis, since our main focus is not on code coverage, but on efficient exploitable bug detection. For example, AEG ignores symbolic file properties such as permissions, in order to avoid producing additional paths.

Symbolic Sockets. To be able to produce remote exploits, AEG provides network support in order to analyze networking code. A symbolic socket descriptor is handled similarly to a symbolic file descriptor, and symbolic network packets and their payloads are handled similarly to symbolic files and their contents. AEG currently handles all network-related functions, including `socket, bind, accept, send`, etc.

Environment Variables. Several vulnerabilities are triggered because of specific environment variables. Thus, AEG supports a complete summary of `get_env`, representing all possible results (concrete values, fully symbolic and failures).

Library Function Calls and System Calls. AEG provides support for about 70 system calls. AEG supports all the basic network system calls, thread-related system calls, such as `fork`, and also all common formatting functions, including `printf` and `syslog`. Threads are handled in the standard way, i.e., we spawn a new symbolic interpreter for each process/thread creation function invocation. In addition, AEG reports a possibly exploitable bug whenever a (fully or partially) symbolic argument is passed to a formatting function. For instance, AEG will detect a format string vulnerability for "`fprintf`(stdout, user_input)".

5.6 DBA, Exploit-Gen and Verify: The Exploit Generation

At a high level, the three components of AEG (DBA, EXPLOIT-GEN and VERIFY) work together to convert the unsafe predicate (Π_{bug}) output by BUG-FIND into a working exploit ϵ.

5.6.1 DBA: Dynamic Binary Analysis

DBA is a dynamic binary analysis (instrumentation) step. It takes in three inputs: 1) the target executable (B_{gcc}) that we want to exploit; 2) the path constraints that lead up to the bug (Π_{bug}); and 3) the names of vulnerable functions and buffers, such as the buffer susceptible to overflow in a stack overflow attack or the buffer that holds the malicious format string in a format string attack. It then outputs a set of runtime information: 1) the address to overwrite (in our implementation, this is the address of the return address of a function, but we can easily extend this to include function pointers or entries in the GOT), 2) the starting address that we write to, and 3) the additional constraints that describe the stack memory contents just before the bug is triggered.

Once AEG finds a bug, it replays the same buggy execution path using a concrete input. The concrete input is generated by solving the path constraints Π_{bug}. During DBA, AEG performs instrumentation on the given executable binary B_{gcc}. When it detects the vulnerable function call, it stops execution and examines the stack. In particular, AEG obtains the address of the return address of the vulnerable function (&retaddr), the address of the vulnerable buffer where the overwrite starts (bufaddr) and the stack memory contents between them (μ).

In the case of format string vulnerabilities, the vulnerable function is a *variadic* formatting function that takes user input as the format argument. Thus, the address of the return

```
1  char *ptr = malloc(100);
2  char buf[100];
3  strcpy(buf, input); // overflow
4  strcpy(ptr, buf);   // ptr dereference
5  return;
```

Figure 5.8: When stack contents are garbled by stack overflow, a program can fail before the return instruction.

address (&retaddr) becomes the return address of the vulnerable formatting function. For example, if there is a vulnerable `printf` function in a program, AEG overwrites the return address of the `printf` function itself, exploiting the format string vulnerability. This way, an attacker can hijack control of the program right after the vulnerable function returns. It is straightforward to adapt additional format string attacks such as GOT hijacking, in AEG.

Stack Restoration. AEG examines the stack contents during DBA in order to generate an exploit predicate ($\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$) that does not corrupt the local stack variables in Exploit-Gen (§ 5.6.2). For example, if there is a dereference from the stack before the vulnerable function returns, simply overwriting the stack will not always produce a valid exploit. Suppose an attacker tries to exploit the program shown in Figure 5.8 using the `strcpy` buffer overflow vulnerability. In this case, `ptr` is located between the return address and the `buf` buffer. Note that `ptr` is dereferenced after the stack overflow attack. Since `ptr` is also on the stack, the contents of `ptr` are garbled by the stack overflow, and might cause the program to crash before the return instruction. Thus, a sophisticated attack must consider the above case by overwriting a valid memory pointer to the stack. AEG properly handles this situation by examining the entire stack space during DBA, and passing the information (μ) to Exploit-Gen.

Algorithm 2: Stack-Overflow Return-to-Stack Exploit Predicate Generation Algorithm

input : (bufaddr, &retaddr, μ) = R
output : Π_{exploit}

for $i = 1$ *to len(μ)* **do**
  $exp\_str[i] \leftarrow \mu[i]$; `// stack restoration`
offset $\leftarrow$ &retaddr - bufaddr;
jmp_target $\leftarrow$ offset + 8 ; `// old ebp + retaddr = 8`
exp_str[offset] $\leftarrow$ jmp_target ; `// eip hijack`
for $i = 1$ *to len(shellcode)* **do**
  exp_str[offset + i] $\leftarrow$ shellcode[i];
return $(Mem[bufaddr] == exp\_str[1]) \wedge \ldots \wedge (Mem[bufaddr + len(\mu) - 1] == exp\_str[len(\mu)])$; // $\Pi_{\texttt{exploit}}$

5.6.2 Exploit-Gen

Exploit-Gen takes in two inputs to produce an exploit: the unsafe program state containing the path constraints (Π_{bug}) and low-level runtime information R, i.e., the vulnerable buffer's address (bufaddr), the address of the vulnerable function's return address (&retaddr), and the runtime stack memory contents (μ). Using that information, Exploit-Gen generates exploit formulas ($\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$) for four types of exploits: 1) stack-overflow return-to-stack, 2) stack-overflow return-to-libc, 3) format-string return-to-stack, 4) format-string return-to-libc. In this chapter, we present the full algorithm only for 1.

In order to generate exploits, AEG performs two major steps. First, AEG determines the class of attack to perform and formulates Π_{exploit} for control hijack. For example, in a stack-overflow return-to-stack attack, Π_{exploit} must have the constraint that the address of the return address (&retaddr) should be overwritten to contain the address of the shellcode—as provided by DBA. Further, the exploit predicate Π_{exploit} must also contain constraints that shellcode must be written on the target buffer. The generated predicate is used in conjunction with Π_{bug} to produce the final constraints (the exploit formula $\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$) that can be solved to produce an exploit. Algorithm 2 shows how the exploit predicate (Π_{exploit}) is generated for stack-overflow return-to-stack attacks.

5.6.2.1 Exploits

AEG produces two types of exploits: return-to-stack and return-to-libc, both of which are the most popular classic control hijack attack techniques. AEG currently cannot handle state-of-the-art protection schemes, but we discuss possible directions in Section 7.7. Additionally, our return-to-libc attack is different from the classic one in that we do not need to know the address of a "/bin/sh" string in the binary. This technique allows bypassing stack randomization (but not libc randomization).

Return-to-stack Exploit. The return-to-stack exploit overwrites the return address of a function so that the program counter points back to the injected input, e.g., user-provided shellcode. To generate the exploit, AEG finds the address of the vulnerable buffer (bufaddr) into which an input string can be copied, and the address where the return address of a vulnerable function is located at. Using the two addresses, AEG calculates the jump target address where the shellcode is located. Algorithm 2 describes how to generate an exploit predicate for a stack overflow vulnerability in the case of a return-to-stack exploit where the shellcode is placed after the return address.

Return-to-libc Exploit. In the classic return-to-libc attack, an attacker usually changes the return address to point to the `execve` function in libc. However, to spawn a shell, the attacker must know the address of a "/bin/sh" string in the binary, which is not common in most programs. In our return-to-libc attack, we create a symbolic link to `/bin/sh` and for the link name we use an arbitrary string which resides in libc. For example, a 5 byte string pattern $e8..00...._{16}$ [5] is very common in libc, because it represents a call instruction on x86. Thus, AEG finds a certain string pattern in libc, and generates a symbolic link to `/bin/sh` in the same directory as the target program. The address of the string is passed as the first argument of `execve` (the file to execute), and the address of a null string 00000000_{16} is used

[5] A dot (.) represents a 4-bit string in hexadecimal notation.

for the second and third arguments. The attack is valid only for local attack scenarios, but is more reliable since it bypasses stack address randomization.

Note that the above exploitation techniques (return-to-stack and return-to-libc) determine how to spawn a shell for a control hijack attack, but not how to hijack the control flow. Thus, the above techniques can be applied by different types of control hijack attacks, e.g., format string attacks and stack overflows. For instance, a format string attack can use either of the above techniques to spawn a shell. AEG currently handles all possible combinations of the above attack-exploit patterns.

5.6.2.2 Exploitation Techniques

Various Shellcode. The return-to-stack exploit requires shellcode to be injected on the stack. To support different types of exploits, AEG has a shellcode database with two shellcode classes: standard shellcodes for local exploits, and binding and reverse binding shellcodes for remote exploits. In addition, this attack restores the stack contents by using the runtime information μ (§ 5.6.1).

Types of Exploits. AEG currently supports four types of exploits: stack-overflow return-to-stack, stack-overflow return-to-libc, format-string return-to-stack, and format-string return-to-libc exploit. The algorithms to generate the *exp_str* for each of the above exploits are simple extensions of Algorithm 2.

Shellcode Format & Positioning. In code-injection attack scenarios, there are two parameters that we must always consider: 1) the format, e.g., size and allowed characters and 2) the positioning of the injected shellcode. Both are important because advanced attacks have complex requirements on the injected payload, e.g., that the exploit string fits within a limited number of bytes or that it only contains alphanumeric characters. To find positioning, AEG applies a brute-force approach: tries every possible user-controlled memory location to place the shellcode. For example, AEG can place the shellcode either below or above the

overwritten return address. To address the special formatting challenge, AEG has a shellcode database containing about 20 different shellcodes, including standard and alphanumeric. Again, AEG tries all possible shellcodes in order to increase reliability. Since AEG has a VERIFY step, all the generated control hijacks are verified to become actual exploits.

5.6.2.3 Reliability of Exploits

Exploits are delicate, especially those that perform control flow hijacking. Even a small change, e.g., the way a program executes either via ./a.out or via ../../../a.out, will result in a different memory layout of the process. This problem persists even when ASLR is turned off. For the same reason, most of the proof-of-concept exploits in popular advisories do not work in practice without some (minor or major) modification. In this subsection, we discuss the techniques employed by AEG to generate reliable exploits for a given system configuration: a) offsetting the difference in environment variables, and b) using NOP-sleds.

Offsetting the Difference in Environment Variables. Environment variables are different for different terminals, program arguments of different length, etc. When a program is first loaded, environment variables will be copied onto the program's stack. Since the stack grows towards lower memory addresses, the more environment variables there are, the lower the addresses of the actual program data on the stack are going to be. Environment variables such as `OLDPWD` and `_` (underscore) change even across same system, since the way the program is invoked matters. Furthermore, the arguments (`argv`) are also copied onto the stack. Thus, the length of the command line arguments affects the memory layout. Thus, AEG calculates the addresses of local variables on the stack based upon the difference in the size of the environment variables between the binary analysis and the normal run. This technique is commonly used if we have to craft the exploit on a machine and execute the exploit on another machine.

NOP-Sled. AEG optionally uses NOP-sleds. For simplicity, Algorithm 2 does not take the NOP-sled option into account. In general, a large NOP-sled can make an exploit more reliable, especially against ASLR protection. On the other hand, the NOP-sled increases the size of the payload, potentially making the exploit more difficult or impossible. In AEG's case, the NOP-sled option can be either turned on or off by a command line option.

5.6.3 Verify

Verify takes in two inputs: 1) the exploit constraints $\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$, and 2) the target binary. It outputs either a concrete working exploit, i.e., an exploit that spawns a shell, or $\perp$, if AEG fails to generate the exploit. Verify first solves the exploit constraints to get a concrete exploit. If the exploit is a local attack, it runs the executable with the exploit as the input and checks if a shell has been spawned. If the exploit is a remote attack, AEG spawns three processes. The first process runs the executable. The second process runs `nc` to send the exploit to the executable. The third process checks that a remote shell has been spawned at port 31337.

Note that, in Figure 5.5, we have shown a straight-line flow from Pre-Process to Verify for simplicity. However, in the actual system, Verify provides feedback to Exploit-Gen if the constraints cannot be solved. This is a cue for Exploit-Gen to select a different shellcode.

5.7 Implementation

AEG is written in a mixture of C++ and Python and consists of 4 major components: symbolic executor (Bug-Find), dynamic binary evaluator (DBA), exploit generator (Exploit-Gen), and constraint solver (Verify). We chose KLEE [21] as our backend symbolic executor, and added about 5000 lines of code to implement our techniques and heuristics as

well as to add in support for other input sources (such as sockets and symbolic environment variables). Our dynamic binary evaluator was written in Python, using a wrapper for the GNU debugger. We used STP for constraint solving [102].

5.8 Evaluation

The following sections present our experimental work on the AEG challenge. We first describe the environment in which we conducted our experiments. Then, we show the effectiveness of AEG by presenting 16 exploits generated by AEG for 14 real-world applications. Next, we highlight the importance of our search heuristics—including preconditioned symbolic execution—in identifying exploitable bugs. In addition, we present several examples illustrating the exploitation techniques already implemented in AEG. Last, we evaluate the reliability of the generated exploits.

5.8.1 Experimental Setup

We evaluated our algorithms and AEG on a machine with a 2.4 GHz Intel(R) Core 2 Duo CPU and 4GB of RAM with 4MB L2 Cache. All experiments were performed under Debian Linux 2.6.26-2. We used LLVM-GCC 2.7 to compile programs to run in our source-based AEG and GCC 4.2.4 to build binary executables. All programs presented in the chapter are unmodified open-source applications that people use and can be downloaded from the Internet. Time measurements are performed with the Unix `time` command. The buggy-path-first and loop exhaustion search heuristics elaborated in § 5.5.3 were turned on by default for all the experiments.

	Program	Ver.	Exploit Type	Vulnerable Input Source	Generation Time (s)	Executable Lines of Code	Advisory ID.
None	aeon	0.2a	Local Stack	Env. Var.	3.8	3392	CVE-2005-1019
	iwconfig	V.26	Local Stack	Arguments	1.5	11314	CVE-2003-0947
	glftpd	1.24	Local Stack	Arguments	2.3	6893	OSVDB#16373
	ncompress	4.2.4	Local Stack	Arguments	12.3	3198	CVE-2001-1413
Length	$htget_1$	0.93	Local Stack	Arguments	57.2	3832	CVE-2004-0852
	$htget_2$	0.93	Local Stack	Env. Var	1.2	3832	Zero-day
	$expect_1$	5.43	Local Stack	Env. Var	187.6	458404	Zero-day
	$expect_2$	5.43	Local Stack	Env. Var	186.7	458404	OSVDB#60979
	socat	1.4	Local Format	Arguments	3.2	35799	CVE-2004-1484
	tipxd	1.1.1	Local Format	Arguments	1.5	7244	OSVDB#12346
Prefix	aspell	0.50	Local Stack	Local File	15.2	550	CVE-2004-0548
	exim	4.41	Local Stack	Arguments	33.8	241856	EDB-ID#796
	xserver	0.1a	Remote Stack	Sockets	31.9	1077	CVE-2007-3957
	rsync	2.5.7	Local Stack	Env. Var	19.7	67744	CVE-2004-2093
	xmail	1.21	Local Stack	Local File	1276.0	1766	CVE-2005-2943
Concolic	corehttp	0.5.3	Remote Stack	Sockets	83.6	4873	CVE-2007-4060
Average Generation Time & Executable Lines of Code					114.6	56784	

Table 5.1: List of open-source programs successfully exploited by AEG. Generation time was measured with the GNU Linux `time` command. Executable lines of code was measured by counting LLVM instructions.

5.8.2 Exploits by AEG

Table 5.1 shows the list of vulnerabilities that AEG successfully exploits. We found these 14 programs from a variety of popular advisories: Common Vulnerabilities and Exposures (CVE), Open Source Vulnerability Database (OSVDB), and Exploit-DB (EDB) and downloaded them to test on AEG. Not only did AEG reproduce the exploits provided in the CVEs, it found and generated working exploits for 2 additional vulnerabilities — 1 for `expect-5.43` and 1 for `htget-0.93`.

We order the table by the kind of path exploration technique used to find the bug, ordered from the least to most amount of information given to the algorithm itself. 4 exploits required no precondition at all and paths were explored using only our path prioritization techniques (§ 5.5.3). We note that although we build on top of KLEE [21], in our experiments KLEE only detected the `iwconfig` exploitable bug.

6 of the exploits were generated only after inferring the possible maximum lengths of symbolic inputs using our static analysis (the Length rows). Without the maximum input length AEG failed most often because symbolic execution would end up considering all possible input lengths up to some maximum buffer size, which was usually very large (e.g., 512 bytes). Since length is automatically inferred, these 6 combined with the previous 4 mean that 10 total exploits were produced automatically with no additional user information.

5 exploits required that the user specify a prefix on the input space to explore. For example, xmail's vulnerable program path is only triggered with valid a email address. Therefore, we needed to specify to AEG that the input included an "@" sign to trigger the vulnerable path.

Corehttp is the only vulnerability that required concolic execution. The input we provided was "A"x (repeats 880 times) + \r\n\r\n. Without specifying the complete GET request, symbolic execution got stuck on exploring where to place white-spaces and EOL (end-of-line) characters.

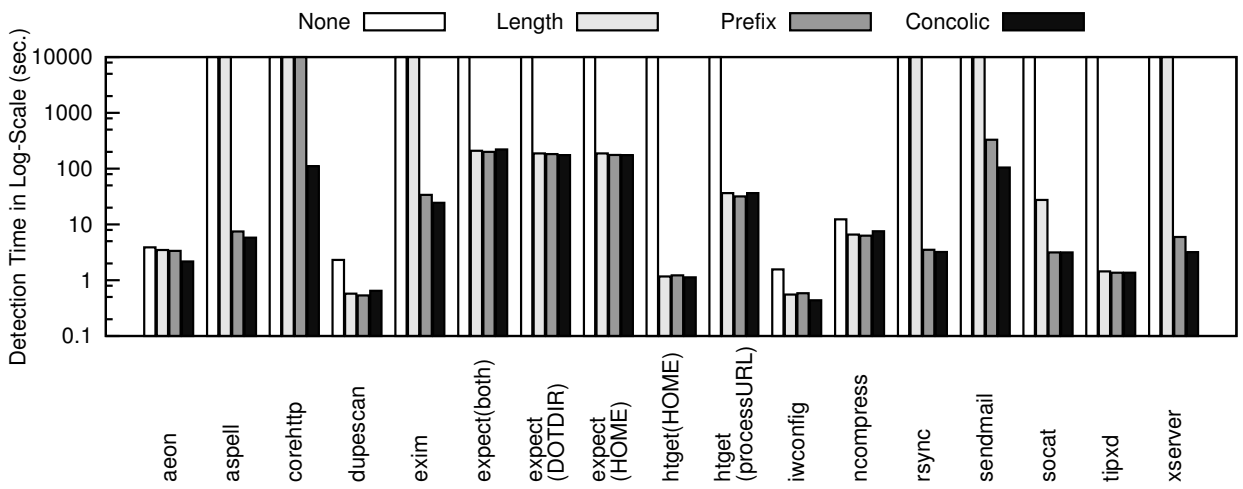


Figure 5.9: Comparison of preconditioned symbolic execution techniques.

Generation Time. Column 5 in Table 5.1 shows the total time to generate working exploits. The quickest we generated an exploit was 0.5s for iwconfig (with a length precondition), which required exploring a single path. The longest was xmail at 1276s (a little over 21 minutes), and required exploring the most paths. On average exploit generation took 114.6s for our test suite. Thus, when AEG works, it tends to be very fast.

Variety of Environment Modeling. Recall from § 5.5.4, AEG handles a large variety of input sources including files, network packets, etc. In order to present the effectiveness of AEG in environment modeling, we grouped the examples by exploit type (Table 5.1 column 4), which is either local stack (for a local stack overflow), local format (for a local format string attack) or remote stack (for a remote stack overflow) and input source (column 5), which shows the source where we provide the exploit string. Possible sources of user input are environment variables, network sockets, files, command line arguments and `stdin`.

The two zero-day exploits, expect and htget, are both environment variable exploits. While most attack scenarios for environment variable vulnerabilities such as these are not terribly exciting, the main point is that AEG found new vulnerabilities and exploited them automatically.

```
1 if(!(sysinfo.config_filename = malloc(strlen(optarg)))) {
2     fprintf(stderr, "Could not allocate memory for filename storage\n")
          ;
3     exit(1);
4 }
5 strcpy((char *)sysinfo.config_filename, optarg);
6 tipxd_log(LOG_INFO, "Config file is %s\n", sysinfo.config_filename);
7 ...
8 void tipxd_log(int priority, char *format, ...) {
9     vsnprintf(log_entry, LOG_ENTRY_SIZE-1, format, ap);
10    syslog(priority, log_entry);
```

Figure 5.10: Code snippet of `tipxd`.

5.8.3 Preconditioned Symbolic Execution and Path Prioritization Heuristics

5.8.3.1 Preconditioned Symbolic Execution

We also performed experiments to show how well preconditioned symbolic execution performs on specific vulnerabilities when different preconditions are used. Figure 5.9 shows the result. We set the maximum analysis time to 10,000 seconds, after which we terminate the program. The preconditioned techniques that failed to detect an exploitable bug within the time limit are shown as a bar of maximum length in Figure 5.9.

Our experiments show that increasing the amount of information supplied to the symbolic executor via the precondition significantly improves bug detection times and thus the effectiveness of AEG. For example, by providing a length precondition we almost tripled the number of exploitable bugs that AEG could detect within the time limit. However, the amount of information supplied did not tremendously change how quickly an exploit is generated, when it succeeds at all.

```
1 int ProcessURL(char *TheURL, char *Hostname, char *Filename, char *
    ActualFilename, unsigned *Port) {
2   char BufferURL[MAXLEN];
3   char NormalURL[MAXLEN];
4   strcpy(BufferURL, TheURL);
5   ...
6   strncpy(Hostname, NormalURL, I);
```

Figure 5.11: Code snippet of `htget`

5.8.3.2 Buggy-Path-First: Consecutive Bug Detection

Recall from § 5.5.3 the path prioritization heuristic to check buggy paths first. `tipxd` and `htget` are example applications where this prioritization heuristic pays off. In both cases there is a non-exploitable bug followed by an exploitable bug in the same path. Figure 5.10 shows a snippet from tipxd, where there is an initial non-exploitable bug on line 1 (it should be "`malloc(strlen(optarg) + 1)`" for the NULL byte). AEG recognizes that the bug is non-exploitable and prioritizes that path higher for continued exploration.

Later on the path, AEG detects a format string vulnerability on line 10. Since the `config_filename` is set from the command line argument `optarg` in line 5, we can pass an arbitrary format string to the `syslog` function in line 10 via the variable `log_entry`. AEG recognizes the format string vulnerability and generates a format string attack by crafting a suitable command line argument.

5.8.4 Mixed Binary and Source Analysis

In § 6.1, we argue that source code analysis alone is insufficient for exploit generation because low-level runtime details like stack layout matter. The `aspell, htget, corehttp, xserver` are examples of this axiom.

For example, Figure 5.11 shows a code snippet from `htget`. The stack frame when invoking this function has the function arguments at the top of the stack, then the return

address and saved ebp, followed by the local buffers BufferURL and NormalURL. The strcpy on line 4 is exploitable where TheURL can be much longer than BufferURL. However, we must be careful in the exploit to *only* overwrite up to the return address, e.g., if we overwrite the return address and Hostname, the program will simply crash when Hostname is dereferenced (before returning) on line 6.

Since our technique performs dynamic analysis, we can reason about runtime details such as the exact stack layout, exactly how many bytes the compiler allocated to a buffer, etc, very precisely. For the above programs this precision is essential, e.g., in htget the predicate asserts that we overwrite up to the return address but no further. If there is not enough space to place the payload before the return address, AEG can still generate an exploit by applying stack restoration (presented in § 5.6.1), where the local variables and function arguments are overwritten, but we impose constraints that their values should remain unchanged. To do so, AEG again relies on our dynamic analysis component to retrieve the runtime values of the local variables and arguments.

5.8.5 Exploit Variants

Whenever an exploitable bug is found, AEG generates an exploit formula ($\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$) and produces an exploit by finding a satisfying answer. However, this does not mean that there is a *single* satisfying answer (exploit). In fact, we expected that there is huge number of inputs that satisfy the formula. To verify our expectations, we performed an additional experiment where we configured AEG to generate *exploit variants*—different exploits produced by the same exploit formula. Table 5.2 shows the number of exploit variants generated by AEG within an hour for 5 sample programs.

Program	# of exploits
`iwconfig`	3265
`ncompress`	576
`aeon`	612
`htget`	939
`glftpd`	2201

Table 5.2: Number of exploit variants generated by AEG within an hour.

5.8.6 Additional Success

AEG also had an anecdotal success. Our research group entered smpCTF 2010, a time-limited international competition where teams compete against each other by solving security challenges. One of the challenges was to exploit a given binary. Our team ran the Hex-rays decompiler to produce source, which was then fed into AEG (with a few tweaks to fix some incorrect decompilation from the Hex-rays tool). AEG returned an exploit in under 60 seconds.

5.9 Discussion and Future Work

Advanced Exploits. In our experiments we focused on stack buffer overflows and format string vulnerabilities. In order to extend AEG to handle heap-based overflows we would likely need to extend the control flow reasoning to also consider heap management structures. Integer overflows are more complicated however, as typically an integer overflow is not problematic by itself. Security-critical problems usually appear when the overflowed integer is used to index or allocate memory. We leave adding support for these types of vulnerabilities as future work.

Other Exploit Classes. While our definition includes the most popular bugs exploited today, e.g., input validation bugs, such as information disclosure, buffer overflows, heap overflows, and so on, it does not capture all security-critical vulnerabilities. For example, our formulation leaves out-of-scope timing attacks against crypto, which are not readily characterized as safety problems. We leave extending AEG to these types of vulnerabilities as future work.

Symbolic Input Size. Our current approach performs simple static analysis and determines that symbolic input variables should be 10% larger in size than the largest statically allocated buffer. While this is an improvement over KLEE (KLEE required a user specify the size), and was sufficient for our examples, it is somewhat simplistic. More sophisticated analysis would provide greater precision for exactly what may be exploitable, e.g., by considering stack layout, and may be necessary for more advanced exploits, e.g., heap overflows where buffers are dynamically allocated.

Portable Exploits. In our approach, AEG produces an exploit for a given environment, i.e., OS, compiler, etc. For example, if AEG generates an exploit for a GNU compiled binary, the same exploit might not work for a binary compiled with the Intel compiler. This is to be expected since exploits are dependent upon run-time layout that may change from compiler to compiler. However, given an exploit that works when compiled with A, we can run AEG on the binary produced from compiler B to check if we can create a new exploit. Also, our current prototype only handles Linux-compatible exploits. Crafting platform-independent and portable exploits is addressed in other work [103] and falls outside the scope of this thesis.

5.10 Related Work

Automatic Exploit Generation. Brumley *et al.* [90] introduced the automatic *patch-based* exploit generation (APEG) challenge. They also introduced the notion that exploits can be described as a predicate on the program state space, which we use and refine in this work. There are two significant differences between AEG and APEG. First, APEG requires access to a buggy program and a patch, while AEG only requires access to a potentially buggy program. Second, APEG defines an exploit as an input violating a new safety check introduced by a patch, e.g., only generating unsafe inputs in Figure 5.4. While Brumley *et al.* speculate generating root shells may be possible, they do not demonstrate it. We extend their notion of "exploit" to include specific actions, and demonstrate that we can produce specific actions such as launch a shell. Heelan *et al.* [91] automatically generated a control flow hijack when the bug is known, and the crashing input is given (similar to concolic execution). Heelan *et al.* can also generated exploits when given a trampoline register. A complete bibliography on AEG can found found in Section 4.5.

Bug-finding techniques. In blackbox fuzzing, we give random inputs to a program until it fails or crashes [10]. Blackbox fuzzing is easy and cheap to use, but it is hard to use in a complex program. Symbolic execution has been used extensively in several application domains, including vulnerability discovery and test case generation [19, 21], input filter generation [22, 100], and others. Symbolic execution is so popular because of its simplicity: it behaves just like regular execution but it also allows data (commonly input) to be symbolic. By performing computations on symbolic data instead of their concrete values, symbolic execution allows us to reason about multiple inputs with a single execution. Taint analysis is a type of information flow analysis for determining whether untrusted user input can flow into trusted sinks. There are both static [104, 105, 106] and dynamic [44, 107] taint analysis

tools. For a more extensive explanation of symbolic execution and taint analysis, we refer to a survey [1].

Symbolic Execution There is a rich variety of work in symbolic execution and formal methods that can be applied to our AEG setting. For example, Engler *et al.* [108] mentioned the idea of *exactly-constrained* symbolic execution, where equality constraints are imposed on symbolic data for concretization. Our problem definition enables any form of formal verification to be used, thus we believe working on formal verification is a good place to start when improving AEG.

5.11 Conclusion

In this chapter, we presented the first fully automatic end-to-end approach for exploit generation. We implemented our approach in AEG and analyzed 14 open-source projects. We successfully generated 16 control flow hijack exploits, two of which were against previously unknown vulnerabilities. In order to make AEG practical, we developed a novel preconditioned symbolic execution technique and path prioritization algorithms for finding and identifying exploitable bugs.

5.12 Acknowledgements

We would like to thank all the people that worked in the AEG project and especially JongHyup Lee, David Kohlbrenner and Lokesh Agarwal. We would also like to thank our anonymous reviewers for their useful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 0953751. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work is also

partially supported by grants from Northrop Grumman as part of the Cybersecurity Research Consortium, from Lockheed Martin, and from DARPA Grant No. N10AP20021.

Chapter 6

State Reduction & Query Elimination

Do not reinvent the wheel.

— My father, Ioannis, *On system architecture.*

In this chapter, we present MAYHEM, the first end-to-end system for automatically finding exploitable bugs in binary (i.e., executable) programs. Every bug reported by MAYHEM is accompanied by a working shell-spawning exploit. The working exploits ensure soundness and that each bug report is security-critical and actionable. MAYHEM works on raw binary code without debugging information. To make exploit generation possible at the binary-level, MAYHEM addresses two major technical challenges: actively managing execution paths without exhausting memory, and reasoning about *symbolic memory indices*, where a load or a store address depends on user input. To this end, we propose two novel techniques: 1) hybrid symbolic execution for combining online and offline (concolic) execution to maximize the benefits of both techniques, and 2) index-based memory modeling, a technique that allows MAYHEM to efficiently reason about symbolic memory at the binary level. We used MAYHEM to find and demonstrate 29 exploitable vulnerabilities in both Linux and Windows programs, 2 of which were previously undocumented.

6.1 Introduction

Bugs are plentiful. In 2014, the Ubuntu Linux bug management database currently listed over 100,000 open bugs. However, bugs that can be exploited by attackers are typically the most serious, and should be patched first. Thus, a central question is not whether a program has bugs, but which bugs are exploitable.

In this chapter we present MAYHEM, a sound system for automatically finding exploitable bugs in binary (i.e., executable) programs. MAYHEM produces a working control-hijack exploit for each bug it reports, thus guaranteeing each bug report is actionable and security-critical. By working with binary code MAYHEM enables even those without source code access to check the (in)security of their software.

MAYHEM detects and generates exploits based on the basic principles introduced in our previous work on AEG [2]. At a high-level, MAYHEM finds exploitable paths by augmenting symbolic execution [16] with additional constraints at potentially vulnerable program points. The constraints include details such as whether an instruction pointer can be redirected, whether we can position attack code in memory, and ultimately, whether we can execute attacker's code. If the resulting formula is satisfiable, then an exploit is possible.

A main challenge in exploit generation is exploring enough of the state space of an application to find exploitable paths. In order to tackle this problem, MAYHEM's design is based on four main principles: 1) the system should be able to make forward progress for arbitrarily long times—ideally run "forever"—without exceeding the given resources (especially memory), 2) in order to maximize performance, the system should not repeat work, 3) the system should not throw away any work—previous analysis results of the system should be reusable on subsequent runs, and 4) the system should be able to reason about symbolic memory where a load or store address depends on user input. Handling memory addresses is essential to exploit real-world bugs. Principle #1 is necessary for running complex

applications, since most non-trivial programs will contain a potentially infinite number of paths to explore.

Current approaches to symbolic execution, e.g., CUTE [30], BitBlaze [32], KLEE [21], SAGE [12], McVeto [31], AEG [2], S2E [29], and others [109], do not satisfy all the above design points. Conceptually, current executors can be divided into two main categories: offline executors — which concretely run a single execution path and then symbolically execute it (also known as trace-based or *concolic* executors, e.g., SAGE), and online executors — which try to execute all possible paths in a single run of the system (e.g., S2E). Neither online nor offline executors satisfy principles #1-#3. In addition, most symbolic execution engines do not reason about symbolic memory, thus do not meet principle #4.

Offline symbolic executors [32, 12] reason about a single execution path at a time. Principle #1 is satisfied by iteratively picking new paths to explore. Further, every run of the system is independent from the others and thus results of previous runs can be immediately reused, satisfying principle #3. However, offline does not satisfy principle #2. Every run of the system needs to restart execution of the program from the very beginning. Conceptually, the same instructions need to be executed repeatedly for every execution trace. Our experimental results show that this re-execution can be very expensive (see §6.7).

Online symbolic execution [21, 29] forks at each branch point. Previous instructions are never re-executed, but the continued forking puts a strain on memory, slowing down the execution engine as the number of branches increase. The result is no forward progress and thus principles #1 and #3 are not met. Some online executors such as KLEE stop forking to avoid being slowed down by their memory use. Such executors satisfy principle #1 but not principle #3 (interesting paths are potentially eliminated).

MAYHEM combines the best of both worlds by introducing *hybrid symbolic execution*, where execution alternates between online and offline symbolic execution runs. Hybrid execution acts like a memory manager in an OS, except that it is designed to *efficiently* swap

out symbolic execution engines. When memory is under pressure, the hybrid engine picks a running executor, and saves the current execution state, and path formula. The thread is restored by restoring the formula, concretely running the program up to the previous execution state, and then continuing. Caching the path formulas prevents the symbolic re-execution of instructions, which is the bottleneck in offline, while managing memory more efficiently than online execution.

MAYHEM also proposes techniques for efficiently reasoning about symbolic memory. A symbolic memory access occurs when a load or store address depends on input. Symbolic pointers are very common at the binary level, and being able to reason about them is necessary to generate control-hijack exploits. In fact, our experiments show that 40% of the generated exploits would have been impossible due to concretization constraints (§6.7). To overcome this problem, MAYHEM employs an index-based memory model (§6.4) to avoid constraining the index whenever possible.

Results are encouraging. While there is ample room for new research, MAYHEM currently generates exploits for several security vulnerabilities: buffer overflows, function pointer overwrites, and format string vulnerabilities for 29 different programs. MAYHEM also demonstrates 2-10× speedup over offline symbolic execution without having the memory constraints of online symbolic execution.

Overall, MAYHEM makes the following contributions:

1) Hybrid execution. We introduce a new scheme for symbolic execution—which we call *hybrid* symbolic execution—that allows us to find a better balance between speed and memory requirements. Hybrid execution enables MAYHEM to explore multiple paths faster than existing approaches (see §6.3).

2) Index-based memory modeling. We propose index-based memory model as a practical approach to dealing with symbolic indices at the binary-level. (see §6.4).

3) Binary-only exploit generation. We present the first end-to-end binary-only exploitable bug finding system that demonstrates exploitability by outputting working control hijack exploits.

6.2 Overview of Mayhem

In this section we describe the overall architecture, usage scenario, and challenges for finding exploitable bugs. We use an HTTP server, `orzHttpd`—shown in Figure 6.1a—as an example to highlight the main challenges and present how MAYHEM works. Note that we show source for clarity and simplicity; MAYHEM runs on binary code.

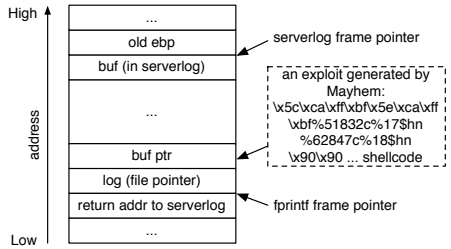
In `orzHttpd`, each HTTP connection is passed to `http_read_request`. This routine in turn calls `static_buffer_read` as part of the loop on line 29 to get the user request string. The user input is placed into the 4096-byte buffer conn-¿read_buf.buf on line 30. Each read increments the variable conn-¿read_buf.used by the number of bytes read so far in order to prevent a buffer overflow. The read loop continues until \r\n\r\n is found, checked on line 34. If the user passes in more than 4096 bytes without an HTTP end-of-line character, the read loop aborts and the server returns a 400 error status message on line 41. Each non-error request gets logged via the `serverlog` function.

The vulnerability itself is in `serverlog`, which calls `fprintf` with a user specified format string (an HTTP request). Variadic functions such as `fprintf` use a format string specifier to determine how to walk the stack looking for arguments. An exploit for this vulnerability works by supplying format strings that cause `fprintf` to walk the stack to user-controlled data. The exploit then uses additional format specifiers to write to the desired location. Figure 6.1b shows the stack layout of `orzHttpd` when the format string vulnerability is detected. There is a call to `fprintf` and the formatting argument is a string of user-controlled bytes.

We highlight several key points for finding exploitable bugs:

```
1 #define BUFSIZE 4096
2
3 typedef struct {
4   char buf[BUFSIZE];
5   int used;
6 } STATIC_BUFFER_t;
7
8 typedef struct conn {
9   STATIC_BUFFER_t read_buf;
10  ... // omitted
11 } CONN_t;
12
13 static void serverlog(LOG_TYPE_t type,
14                       const char *format, ...)
15 {
16   ... // omitted
17   if(format != NULL) {
18     ... // omitted
19   }
20   fprintf(log, buf); // vulnerable point
21   fflush(log);
22 }
23
24 HTTP_STATE_t http_read_request(CONN_t *conn)
25 {
26   ... // omitted
27   while(conn->read_buf.used < BUFSIZE) {
28     sz = static_buffer_read(conn, &conn->read_buf);
29     if(sz < 0) {
30       ...
31       conn->read_buf.used += sz;
32       if(memcmp(&conn->read_buf.buf[conn->read_buf.used] - 4, "\r\n\r\n", 4) == 0)
33       {
34         break;
35       }
36   }
37   if(conn->read_buf.used >= BUFSIZE) {
38     conn->status.st = HTTP_STATUS_400;
39     return HTTP_STATE_ERROR;
40   }
41   ...
42   serverlog(ERROR_LOG,
43             "%s\n",
44             conn->read_buf.buf);
45   ...
46 }
```

(a) Code Snippet



(b) Stack diagram of the vulnerable program.

Figure 6.1: `orzHttpd` vulnerability

Low-level details matter: Determining exploitability requires that we reason about low-level details like return addresses and stack pointers. This is our motivation for focusing on binary-level techniques.

There are an enormous number of paths: In the example, there is a new path on every encounter of an `if` statement, which can lead to an exponential path explosion. Additionally, the number of paths in many portions of the code is related to the size of the input. For example, `memcmp` unfolds a loop, creating a new path for symbolic execution on each iteration. Longer inputs mean more conditions, more forks, and harder scalability challenges. Unfortunately most exploits are not short strings, e.g., in a buffer overflow typical exploits are hundreds or thousands of bytes long.

The more checked paths, the better: To reach the exploitable `fprintf` bug in the example, MAYHEM needs to reason through the loop, read input, fork a new interpreter for every possible path and check for errors. Without careful resource management, an engine can get bogged down with too many symbolic execution threads because of the huge number of possible execution paths.

Execute as much natively as possible: Symbolic execution is slow compared to concrete execution since the semantics of an instruction are simulated in software. In `orzHttpd`, millions of instructions set up the basic server before an attacker can even connect to a socket. We want to execute these instructions concretely and then switch to symbolic execution.

The MAYHEM architecture for finding exploitable bugs is shown in Figure 6.2. The user starts MAYHEM by running:

```
mayhem -sym-net 80 400 ./orzhttpd
```

The command-line tells MAYHEM to symbolically execute `orzHttpd`, and open sockets on port 80 to receive symbolic 400-byte long packets. All remaining steps to create an exploit are performed automatically.

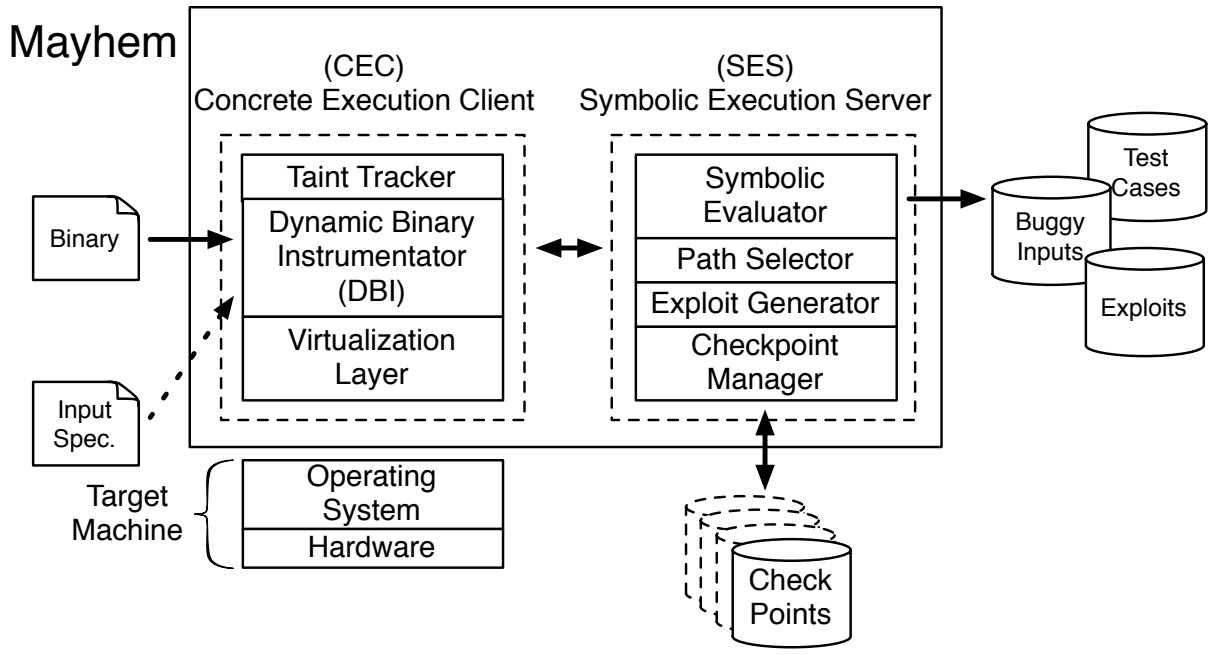


Figure 6.2: MAYHEM architecture

MAYHEM consists of two concurrently running processes: a *Concrete Executor Client* (CEC), which executes code natively on a CPU, and a *Symbolic Executor Server* (SES). Both are shown in Figure 6.2. At a high level, the CEC runs on a target system, and the SES runs on any platform, waiting for connections from the CEC. The CEC takes in a binary program along with the potential symbolic sources (input specification) as an input, and begins communication with the SES. The SES then symbolically executes blocks that the CEC sends, and outputs several types of test cases including normal test cases, crashes, and exploits. The steps followed by MAYHEM to find the vulnerable code and generate an exploit are:

1. The --sym-net 80 400 argument tells MAYHEM to perform symbolic execution on data read in from a socket on port 80. Effectively this is specifying which input sources are potentially under attacker control. MAYHEM can handle attacker input from environment variables, files, and the network.

2. The CEC loads the vulnerable program and connects to the SES to initialize all symbolic input sources. After the initialization, MAYHEM executes the binary concretely on the CPU in the CEC. During execution, the CEC instruments the code and performs dynamic taint analysis [44]. Our taint tracking engine checks if a block contains tainted instructions, where a block is a sequence of instructions that ends with a conditional jump or a call instruction.

3. When the CEC encounters a tainted branch condition or jump target, it suspends concrete execution. A tainted jump means that the target may be dependent on attacker input. The CEC sends the instructions to the SES and the SES determines which branches are feasible. The CEC will later receive the next branch target to explore from the SES.

4. The SES, running in parallel with the CEC, receives a stream of tainted instructions from the CEC. The SES jits the instructions to an intermediate language (§7.2), and symbolically executes the corresponding IL. The CEC provides any concrete values whenever needed, e.g., when an instruction operates on a symbolic operand and a concrete operand. The SES maintains two types of formulas:

   Path Formula The path formula reflects the constraints to reach a particular line of code. Each conditional jump adds a new constraint on the input. For example, lines 32-33 create two new paths: one which is constrained so that the read input ends in an \r\n\r\n and line 35 is executed, and one where the input does not end in \r\n\r\n and line 28 will be executed.

   Exploitability Formula The exploitability formula determines whether i) the attacker can gain control of the instruction pointer, and ii) execute a payload.

5. When MAYHEM hits a tainted branch point, the SES decides whether we need to fork execution by querying the SMT solver. If we need to fork execution, all the new

forks are sent to the path selector to be prioritized. Upon picking a path, the SES notifies the CEC about the change and the corresponding execution state is restored. If the system resource cap is reached, then the checkpoint manager starts generating checkpoints instead of forking new executors (§6.3). At the end of the process, test cases are generated for the terminated executors and the SES informs the CEC about which checkpoint should continue execution next.

6. During the execution, the SES switches context between executors and the CEC checkpoints/restores the provided execution state and continues execution. To do so, the CEC maintains a virtualization layer to handle the program interaction with the underlying system and checkpoint/restore between multiple program execution states (§6.3.3).

7. When MAYHEM detects a tainted jump instruction, it builds an exploitability formula, and queries an SMT solver to see if it is satisfiable. A satisfying input will be, by construction, an exploit. If no exploit is found on the tainted branch instruction, the SES keeps exploring execution paths.

8. The above steps are performed at each branch until an exploitable bug is found, MAYHEM hits a user-specified maximum runtime, or all paths are exhausted.

6.3 Hybrid Symbolic Execution

MAYHEM is a hybrid symbolic execution system. Instead of running in pure online or offline execution mode, MAYHEM can alternate between modes. In this section we present the motivation and mechanics of hybrid execution.

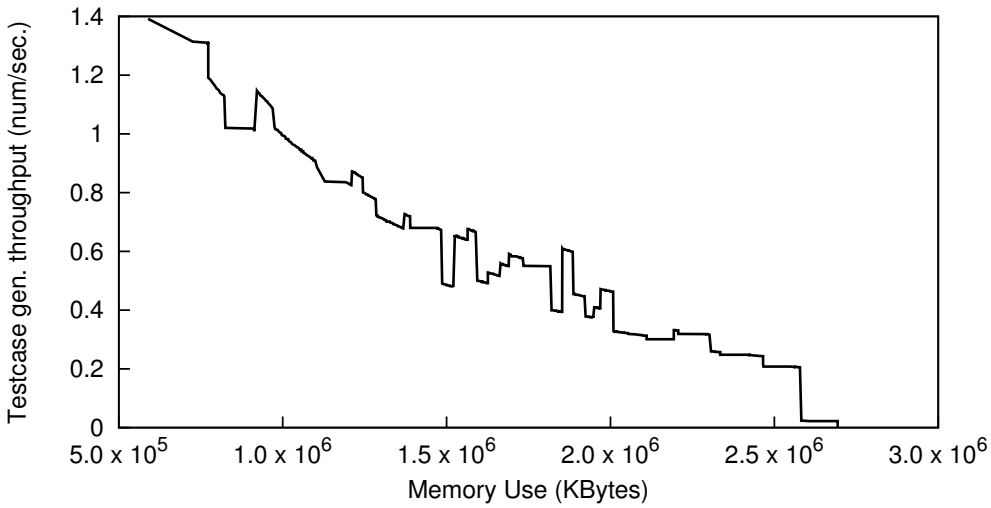


Figure 6.3: Online execution throughput versus memory use.

6.3.1 Previous Symbolic Execution Systems

Offline symbolic execution—as found in systems such as SAGE [12]—requires two inputs: the target program and an initial seed input. In the first step, offline systems concretely execute the program on the seed input and record a trace. In the second step, they symbolically execute the instructions in the recorded trace. This approach is called *concolic* execution, a juxtaposition of concrete and symbolic execution. Offline execution is attractive because of its simplicity and low resource requirements; we only need to handle a single execution path at a time.

The top-left diagram of Figure 3.2 highlights an immediate drawback of this approach. For every explored execution path, we need to first re-execute a (potentially) very large number of instructions until we reach the symbolic condition where execution forked, and then begin to explore new instructions.

Online symbolic execution avoids this re-execution cost by forking two interpreters at branch points, each one having a copy of the current execution state. Thus, to explore a different path, online execution simply needs to perform a *context switch* to the execution

state of a suspended interpreter. S2E [29], KLEE [21] and AEG [2] follow this approach by performing online symbolic execution on LLVM bytecode.

However, forking off a new executor at each branch can quickly strain the memory, causing the entire system to grind to a halt. State-of-the-art online executors try to address this problem with aggressive copy-on-write optimizations. For example, KLEE has an immutable state representation and S2E shares common state between snapshots of physical memory and disks. Nonetheless, since all execution states are kept in memory simultaneously, eventually all online executors will reach the memory cap. The problem can be mitigated by using DFS (Depth-First-Search)—however, this is not a very useful strategy in practice. To demonstrate the problem, we downloaded S2E [29] and ran it on a coreutils application (`echo`) with 2 symbolic arguments, each one 10 bytes long. Figure 6.3 shows how the symbolic execution throughput (number of test cases generated per second) is slowed down as the memory use increases.

6.3.2 Hybrid Symbolic Execution

MAYHEM introduces *hybrid symbolic execution* to actively manage memory without constantly re-executing the same instructions. Hybrid symbolic execution alternates between online and offline modes to maximize the effectiveness of each mode. MAYHEM starts analysis in online mode. When the system reaches a memory cap, it switches to offline mode and does not fork any more executors. Instead, it produces checkpoints to start new online executions later on. The crux of the system is to distribute the online execution tasks into subtasks without losing potentially interesting paths. The hybrid execution algorithm employed by MAYHEM is split into four main phases:

1. Initialization: The first time MAYHEM is invoked for a program, it initializes the checkpoint manager, the checkpoint database, and test case directories. It then starts online execution of the program and moves to the next phase.

2. Online Exploration: During the online phase, MAYHEM symbolically executes the program in an online fashion, context-switching between current active execution states, and generating test cases.

3. Checkpointing: The checkpoint manager monitors online execution. Whenever the memory utilization reaches a cap, or the number of running executors exceeds a threshold, it will select and generate a checkpoint for an active executor. A checkpoint contains the symbolic execution state of the suspended executor (path predicate, statistics, etc.) and replay information[1]. The concrete execution state is discarded. When the online execution eventually finishes all active execution paths, MAYHEM moves to the next phase.

4. Checkpoint Restoration: The checkpoint manager selects a checkpoint based on a ranking heuristic 6.3.4 and restores it in memory. Since the symbolic execution state was saved in the checkpoint, MAYHEM only needs to re-construct the concrete execution state. To do so, MAYHEM concretely executes the program using one satisfiable assignment of the path predicate as input, until the program reaches the instruction when the execution state was suspended. At that point, the concrete state is restored and the online exploration (phase 2) restarts. Note that phase 4 avoids symbolically re-executing instructions during the checkpoint restoration phase (unlike standard concolic execution), and the re-execution happens concretely. In Figure 3.2 we showed the intuition behind hybrid execution. We provide a detailed comparison between online, offline, and hybrid execution in Section 6.7.3.

6.3.3 Design and Implementation of the CEC

The CEC takes in the binary program, a list of input sources to be considered symbolic, and an optional checkpoint input that contains execution state information from a previous run. The CEC concretely executes the program, hooks input sources and performs taint analysis

[1]Note that the term "checkpoint" differs from an offline execution "seed", which is just a concrete input.

on input variables. Every basic block that contains tainted instructions is sent to the SES for symbolic execution. As a response, the CEC receives the address of the next basic block to be executed and whether to save the current state as a restoration point. Whenever an execution path is complete, the CEC context-switches to an unexplored path selected by the SES and continues execution. The CEC terminates only if all possible execution paths have been explored or a threshold is reached. If we provide a checkpoint, the CEC first executes the program concretely until the checkpoint and then continues execution as before.

Virtualization Layer. During an online execution run, the CEC handles multiple concrete execution states of the analyzed program simultaneously. Each concrete execution state includes the current register context, memory and OS state (the OS state contains a snapshot of the virtual filesystem, network and kernel state). Under the guidance of the SES and the path selector, the CEC context switches between different concrete execution states depending on the symbolic executor that is currently active. The virtualization layer mediates all system calls to the host OS and emulates them. Keeping separate copies of the OS state ensures there are no side-effects across different executions. For instance, if one executor writes a value to a file, this modification will only be visible to the current execution state—all other executors will have a separate instance of the same file.

Efficient State Snapshot. Taking a full snapshot of the concrete execution state at every fork is very expensive. To mitigate the problem, CEC shares state across execution states–similar to other systems [21, 29]. Whenever execution forks, the new execution state reuses the state of the parent execution. Subsequent modifications to the state are recorded in the current execution.

6.3.4 Design and Implementation of the SES

The SES manages the symbolic execution environment and decides which paths are executed by the CEC. The environment consists of a symbolic executor for each path, a path selector which determines which feasible path to run next, and a checkpoint manager.

The SES caps the number of symbolic executors to keep in memory. When the cap is reached, MAYHEM stops generating new interpreters and produces *checkpoints*; execution states that will explore program paths that MAYHEM was unable to explore in the first run due to the memory cap. Each checkpoint is prioritized and used by MAYHEM to continue exploration of these paths at a subsequent run. Thus, when all pending execution paths terminate, MAYHEM selects a new checkpoint and continues execution—until all checkpoints are consumed and MAYHEM exits.

Each symbolic executor maintains two contexts (as state): a variable context containing all symbolic register values and temporaries, and a memory context keeping track of all symbolic data in memory. Whenever execution forks, the SES clones the current symbolic state (to keep memory low, we keep the execution state immutable to take advantage of copy-on-write optimizations—similar to previous work [21, 29]) and adds a new symbolic executor to a priority queue. This priority queue is regularly updated by our path selector to include the latest changes (e.g., which paths were explored, instructions covered, and so on).

Preconditioned Symbolic Execution: MAYHEM implements preconditioned symbolic execution as in AEG [2]. In preconditioned symbolic execution, a user can optionally give a partial specification of the input, such as a prefix or length of the input, to reduce the range of search space. If a user does not provide a precondition, then SES tries to explore all feasible paths. This corresponds to the user providing the minimum amount of information to the system.

Path Selection: MAYHEM applies path prioritization heuristics—as found in systems such as SAGE [12] and KLEE [21]—to decide which path should be explored next. Currently,

MAYHEM uses three heuristic ranking rules: a) executors exploring new code (e.g., instead of executing known code more times) have high priority, b) executors that identify symbolic memory accesses have higher priority, and c) execution paths where symbolic instruction pointers are detected have the highest priority. The heuristics are designed to prioritize paths that are most likely to contain a bug. For instance, the first heuristic relies on the assumption that previously explored code is less likely to contain a bug than new code.

6.3.5 Performance Tuning

MAYHEM employs several optimizations to speed-up symbolic execution. We present three optimizations that were most effective: 1) independent formula, 2) algebraic simplifications, and 3) taint analysis.

Similar to KLEE [21], MAYHEM splits the path predicate to independent formulas to optimize solver queries. A small implementation difference compared to KLEE is that MAYHEM keeps a map from input variables to formulas at all times. It is not constructed only for querying the solver (this representation allows more optimizations §6.4). MAYHEM also applies other standard optimizations as proposed by previous systems such as the constraint subsumption optimization [12], a counter-example cache [21] and others. MAYHEM also simplifies symbolic expressions and formulas by applying algebraic simplifications, e.g. `x` $\oplus$ `x = 0`, `x & 0 = 0`, and so on.

Recall from §6.3.3, MAYHEM uses taint analysis [44] to selectively execute instruction blocks that deal with symbolic data. This optimization gives a 8× speedup on average over executing all instruction blocks (see §6.7.7).

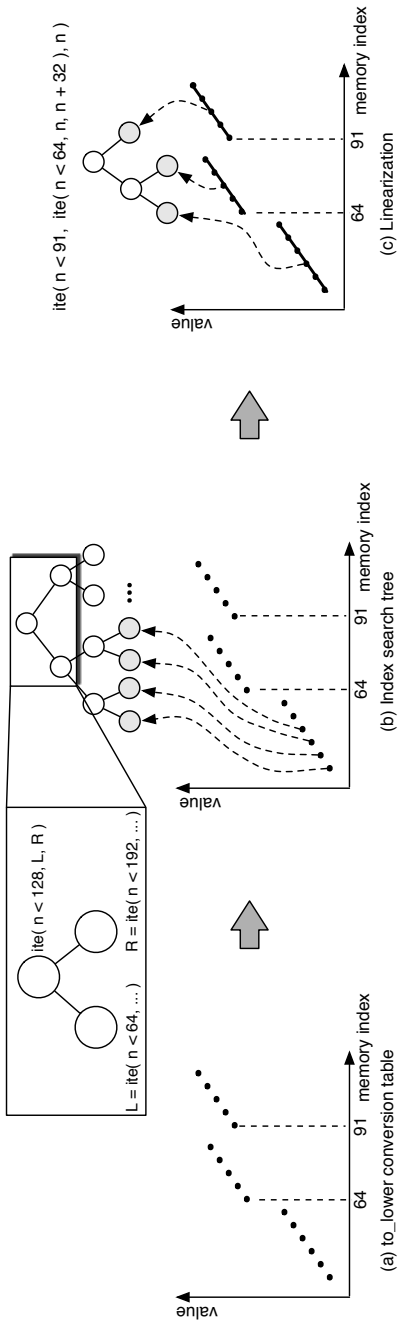


Figure 6.4: Figure (a) shows the `to_lower` conversion table, (b) shows the generated IST, and (c) the IST after linearization.

6.4 Index-based Memory Modeling

MAYHEM introduces an *index-based memory model* as a practical approach to handling symbolic memory loads. The index-based model allows MAYHEM to adapt its treatment of symbolic memory based on the value of the index. In this section we present the entire memory model of MAYHEM.

MAYHEM models memory as a map $\mu : I \rightarrow E$ from 32-bit indices (i) to expressions (e). In a `load(`μ`,`i`)` expression, we say that index i *indexes* memory μ, and the loaded value e represents the *contents* of the i^{th} memory cell. A load with a concrete index i is directly translated by MAYHEM into an appropriate lookup in μ (i.e., $\mu[i]$). A `store(`μ`,` i`,` e`)` instruction results in a new memory $\mu[i \leftarrow e]$ where i is mapped to e.

6.4.1 Previous Work & Symbolic Index Modeling

A symbolic index occurs when the index used in a memory lookup is not a number, but an expression—a pattern that appears very frequently in binary code. For example, a C `switch(c)` statement is compiled down to a jump-table lookup where the input character `c` is used as the index. Standard string conversion functions (such as ASCII to Unicode and vice versa, `to_lower`, `to_upper`, etc.) are all in this category.

Handling arbitrary symbolic indices is notoriously hard, since a symbolic index may (in the worst case) reference *any* cell in memory. Previous research and state-of-the-art tools indicate that there are two main approaches for handling a symbolic index: a) concretizing the index and b) allowing memory to be fully symbolic.

First, concretizing means instead of reasoning about all possible values that could be indexed in memory, we *concretize* the index to a single specific address. This concretization can reduce the complexity of the produced formulas and improve solving/exploration times. However, constraining the index to a single value may cause us to miss paths—for instance,

if they depend on the value of the index. Concretization is the natural choice for offline executors, such as SAGE [12] or BitBlaze [32], since only a single memory address is accessed during concrete execution.

Reasoning about all possible indices is also possible by treating memory as fully symbolic. For example, tools such as McVeto [31], BAP [41] and BitBlaze [32] offer capabilities to handle symbolic memory. The main trade-off—when compared with the concretization approach—is performance. Formulas involving symbolic memory are more expressive, thus solving/exploration times are usually higher.

6.4.2 Memory Modeling in Mayhem

The first implementation of MAYHEM followed the simple concretization approach and concretized all memory indices. This decision proved to be severely limiting in that selecting a single address for the index usually did not allow us to satisfy the exploit payload constraints. Our experiments show that 40% of the examples require us to handle symbolic memory—simple concretization was insufficient (see §6.7).

The alternative approach was symbolic memory. To avoid the scalability problems associated with fully symbolic memory, MAYHEM models memory *partially*, where writes are always concretized, but symbolic reads are allowed to be modeled symbolically. In the rest of this section we describe the index-based memory model of MAYHEM in detail, as well as some of the key optimizations.

Memory Objects. To model symbolic reads, MAYHEM introduces *memory objects*. Similar to the global memory μ, a memory object $\mathcal{M}$ is also a map from 32-bit indices to expressions. Unlike the global memory however, a memory object is immutable. Whenever a symbolic index is used to read memory, MAYHEM generates a fresh memory object $\mathcal{M}$ that contains all values that could be accessed by the index—$\mathcal{M}$ is a partial snapshot of the global memory.

Using the memory object, MAYHEM can reduce the evaluation of a `load(`μ`, `i`)` expression to $\mathcal{M}[i]$. Note, that this is semantically equivalent to returning $\mu[i]$. The key difference is in the size of the symbolic array we introduce in the formula. In most cases, the memory object $\mathcal{M}$ will be orders of magnitude smaller than the entire memory μ.

Memory Object Bounds Resolution. Instantiating the memory object requires MAYHEM to find all possible values of a symbolic index i. In the worst case, this may require up to 2^{32} queries to the solver (for 32-bit memory addresses). To tackle this problem MAYHEM exchanges some accuracy for scalability by resolving the bounds $[\mathcal{L},\mathcal{U}]$ of the memory region—where $\mathcal{L}$ is the lower and $\mathcal{U}$ is the upper bound of the index. The bounds need to be conservative, i.e., all possible values of the index should be within the $[\mathcal{L},\mathcal{U}]$ interval. Note that the memory region does not need to be continuous, for example i might have only two realizable values ($\mathcal{L}$ and $\mathcal{U}$).

To obtain these bounds MAYHEM uses the solver to perform binary search on the value of the index in the context of the current path predicate. For example, initially for the lowest bound of a 32-bit i: $\mathcal{L} \in [0, 2^{32}-1]$. If $i < \frac{2^{32}-1}{2}$ is satisfiable then $\mathcal{L} \in [0, \frac{2^{32}-1}{2} - 1]$ while unsatisfiability indicates that $\mathcal{L} \in [\frac{2^{32}-1}{2}, 2^{32}-1]$. We repeat the process until we recover both bounds. Using the bounds we can now instantiate the memory object (using a fresh symbolic array $\mathcal{M}$) as follows: $\forall i \in [\mathcal{L},\mathcal{U}] : \mathcal{M}[i] = \mu[i]$.

The bounds resolution algorithm described above is sufficient to generate a conservative representation of memory objects and allow MAYHEM to reason about symbolic memory reads. In the rest of the section we detail the main optimization techniques MAYHEM includes to tackle some of the caveats of the original algorithm:

- Querying the solver on every symbolic memory dereference is expensive. Even with binary search, identifying both bounds of a 32-bit index required $\sim$ 54 queries on average (§6.7) (§6.4.2.1,§6.4.2.2,§6.4.2.3).

- The memory region may not be continuous. Even though many values between the bounds may be infeasible, they are still included in the memory object, and consequently, in the formula (§6.4.2.2).

- The values within the memory object might have structure. By modeling the object as a single byte array we are missing opportunities to optimize our formulas based on the structure. (§6.4.2.4,§6.4.2.5).

- In the worst case, a symbolic index may access any possible location in memory (§6.4.3).

6.4.2.1 Value Set Analysis (VSA)

MAYHEM employs an online version of VSA [64] to reduce the solver load when resolving the bounds of a symbolic index (i). VSA returns a strided interval for the given symbolic index. A strided interval represents a set of values in the form $\mathcal{S}[\mathcal{L},\mathcal{U}]$, where $\mathcal{S}$ is the stride and $\mathcal{L}$, $\mathcal{U}$ are the bounds. For example, the interval $2[1,5]$ represents the set $\{1,3,5\}$. The strided interval output by VSA will be an over-approximation of all possible values the index might have. For instance, $i = (1 + byte) << 1$ — where $byte$ is a symbolic byte with an interval $1[0,255]$ — results in an interval: $VSA(i) = 2[2,512]$.

The strided interval produced by VSA is then refined by the solver (using the same binary-search strategy) to get the tight lower and upper bounds of the memory object. For instance, if the path predicate asserts that $byte < 32$, then the interval for the index $(1 + byte) << 1$ can be refined to $2[2,64]$. Using VSA as a preprocessing step has a cascading effect on our memory modeling: a) we perform 70% less queries to resolve the exact bounds of the memory object (§6.7), b) the strided interval can be used to eliminate impossible values in the $[\mathcal{L},\mathcal{U}]$ region, thus making formulas simpler, and c) the elimination can trigger other optimizations (see §6.4.2.5).

6.4.2.2 Refinement Cache

Every VSA interval is refined using solver queries. The refinement process can still be expensive (for instance, the over-approximation returned by VSA might be too coarse). To avoid repeating the process for the same intervals, MAYHEM keeps a cache mapping intervals to potential refinements. Whenever we get a cache hit, we query the solver to check whether the cached refinement is accurate for the current symbolic index, before resorting to binary-search for refinement. The refinement cache can reduce the number of bounds-resolution queries by 82% (§6.7).

6.4.2.3 Lemma Cache

Checking an entry of the refinement cache still requires solver queries. MAYHEM uses another level of caching to avoid repeatedly querying α-equivalent formulas, i.e., formulas that are structurally equivalent up to variable renaming. To do so, MAYHEM converts queried formulas to a canonical representation (F) and caches the query results (Q) in the form of a *lemma*: $F \rightarrow Q$. The answer for any formula mapping to the same canonical representation is retrieved immediately from the cache. The lemma cache can reduce the number of bounds-resolution queries by up to 96% (§6.7). The effectiveness of this cache depends on the independent formulas optimization §6.3.5. The path predicate has to be represented as a set of independent formulas, otherwise any new formula addition to the current path predicate would invalidate all previous entries of the lemma cache.

6.4.2.4 Index Search Trees (ISTs)

Any value loaded from a memory object $\mathcal{M}$ is symbolic. To resolve constraints involving a loaded value ($\mathcal{M}[i]$), the solver needs to both find an entry in the object that satisfies the constraints *and* ensure that the index to the object entry is realizable. To lighten the burden on the solver, MAYHEM replaces memory object lookup expressions with *index search trees*

(ISTs). An IST is a binary search tree where the symbolic index is the key and the leaf nodes contain the entries of the object. The entire tree is encoded in the formula representation of the load expression.

More concretely, given a (sorted by address) list of entries E within a memory object $\mathcal{M}$, a balanced IST for a symbolic index i is defined as: $IST(E) = ite(i < addr(E_{right}), E_{left}, E_{right}))$, where `ite` represents an if-then-else expression, E_{left} (E_{right}) represents the left (right) half of the initial entries E, and $addr(\cdot)$ returns the lowest address of the given entries. For a single entry the IST returns the entry without constructing any `ite` expressions.

Note that the above definition constructs a balanced IST. We could instead construct the IST with nested `ite` expressions—making the formula depth $O(n)$ in the number of object entries instead of $O(\log n)$. However, our experimental results show that a balanced IST is 4× faster than a nested IST (§6.7). Figure 6.4 shows how MAYHEM constructs the IST when given the entries of a memory object (the `to_lower` conversion table) with a single symbolic character as the index.

6.4.2.5 Bucketization with Linear Functions

The IST generation algorithm creates a leaf node for each entry in the memory object. To reduce the number of entries, MAYHEM performs an extra preprocessing step before passing the object to the IST. The idea is that we can use the memory object structure to combine multiple entries into a single *bucket*. A bucket is an index-parameterized expression that returns the value of the memory object for every index within a range.

MAYHEM uses linear functions to generate buckets. Specifically, MAYHEM sweeps all entries within a memory object and joins consecutive points ($\langle index, value \rangle$ tuples) into lines, a process we call *linearization*. Any two points can form a line $y = \alpha x + \beta$. Follow-up points $\langle i_i, v_i \rangle$ will be included in the same line if $u_i = \alpha i_i + \beta$. At the end of linearization, the memory object is split into a list of buckets, where each bucket is either a line or an isolated

```
1 typedef struct {
2   int value;
3   char * bar;
4 } foo;
5 int vulnerable (char * input)
6 {
7   foo * ptr = init;
8   buffer[100];
9   strcpy(buffer, input);
10  buffer[0] = ptr->bar[0];
11  return 0;
12 }
```

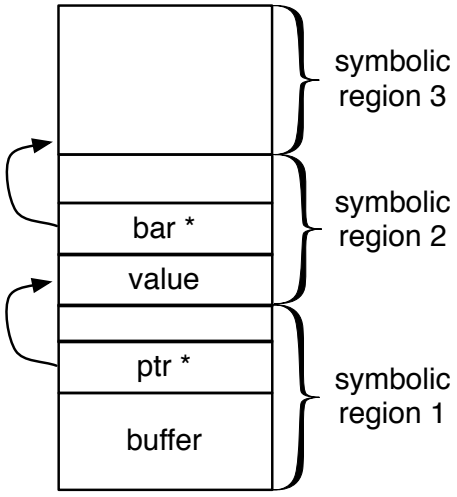


Figure 6.5: MAYHEM reconstructing symbolic data structures.

point. The list of buckets can now be passed to the IST algorithm. Figure 6.4 shows the `to_lower` IST after applying linearization. Linearization effectively reduces the number of leaf nodes from 256 to 3.

The idea of using linear functions to simplify memory lookups comes from a simple observation: linear-like patterns appear frequently for several operations at the binary level. For example, jump tables generated by switch statements, conversion and translation tables (e.g., ASCII to Unicode and vice versa) all contain values that are scaling linearly with the index.

6.4.3 Prioritized Concretization.

Modeling a symbolic load using a memory object is beneficial when the size of the memory object is significantly smaller than the entire memory ($|\mathcal{M}| \ll |\mu|$). Thus, the above optimizations are only activated when the size of the memory object, approximated by the range, is below a threshold ($|\mathcal{M}| < 1024$ in our experiments).

Whenever the memory object size exceeds the threshold, MAYHEM will concretize the index used to access it. However, instead of picking a satisfying value at random, MAYHEM attempts to *prioritize* the possible concretization values. Specifically, for every symbolic pointer, MAYHEM performs three checks:

1. Check if it is possible to redirect the pointer to unmapped memory under the context of the current path predicate. If true, MAYHEM will generate a crash test case for the satisfying value.

2. Check if it is possible to redirect the symbolic pointer to symbolic data. If it is, MAYHEM will redirect (and concretize) the pointer to the least constrained region of the symbolic data. By redirecting the pointer towards the least constrained region, MAYHEM tries to avoid loading overconstrained values, thus eliminating potentially interesting paths that depend on these values. To identify the least constrained region, MAYHEM splits memory into symbolic regions, and sorts them based on the complexity of constraints associated with each region.

3. If all of the above checks fail, MAYHEM concretizes the index to a valid memory address and continues execution.

The above steps infer whether a symbolic expression is a pointer, and if so, whether it is valid or not (e.g., NULL). For example, Figure 6.5 contains a buffer overflow at line 9. However, an attacker is not guaranteed to hijack control even if `strcpy` overwrites the return address. The program needs to reach the return instruction to actually transfer control. However, at line 10 the program performs two dereferences both of which need to succeed (i.e., avoid crashing the program) to reach line 11 (note that pointer *ptr* is already overwritten with user data). MAYHEM augmented with prioritized concretization will generate 3 distinct test cases: 1) a crash test case for an invalid dereference of pointer *ptr*, 2) a crash test case where dereferencing pointer *bar* fails after successfully redirecting *ptr* to symbolic data, and

3) an exploit test case, where both dereferences succeed and user input hijacks control of the program. Figure 6.5 shows the memory layout for the third test case.

6.5 Exploit Generation

MAYHEM checks for two exploitable properties: a symbolic (tainted) instruction pointer, and a symbolic format string. Each property corresponds to a buffer overflow and format string attack respectively. Whenever any of the two exploitable policies are violated, MAYHEM generates an exploitability formula and tries to find a satisfying answer, i.e., an exploit.

MAYHEM can generate both local and remote attacks. Our generic design allows us to handle both types of attacks similarly. For Windows, MAYHEM detects overwritten Structured Exception Handler (SEH) on the stack when an exception occurs, and tries to create an SEH-based exploit.

Buffer Overflows: MAYHEM generates exploits for any possible instruction-pointer overwrite, commonly triggered by a buffer overflow. When MAYHEM finds a symbolic instruction pointer, it first tries to generate jump-to-register exploits, similar to previous work [91]. For this type of exploit, the instruction pointer should point to a trampoline, e.g. `jmp %eax`, and the register, e.g. `%eax`, should point to a place in memory where we can place our shellcode. By encoding those constraints into the formula, MAYHEM is able to query the solver for a satisfying answer. If an answer exists, we proved that the bug is exploitable. If we can't generate a jump-to-register exploit, we try to generate a simpler exploit by making the instruction pointer point directly to a place in memory where we can place shellcode.

Format String Attacks: To identify and generate format string attacks, MAYHEM checks whether the format argument of format string functions, e.g., `printf`, contains any symbolic bytes. If any symbolic bytes are detected, it tries to place a format string payload within the argument that will overwrite the return address of the formatting function.

6.6 Implementation

MAYHEM consists of about 27,000 lines of C/C++ and OCaml code. Our binary instrumentation framework was built on Pin [63] and all the hooks for modeled system and API calls were written in C/C++. The symbolic execution engine is written solely in OCaml and consists of about 10,000 lines of code. We rely on BAP [41] to convert assembly instructions to the IL. We use Z3 [66] as our decision procedure, for which we built direct OCaml bindings. To allow for remote communication between the two components we implemented our own cross-platform, light-weight RPC protocol (both in C++ and OCaml). Additionally, to compare between different symbolic execution modes, we implemented all three: online, offline and hybrid.

6.7 Evaluation

6.7.1 Experimental Setup

We evaluated our system on 2 virtual machines running on a desktop with a 3.40GHz Intel(R) Core i7-2600 CPU and 16GB of RAM. Each VM had 4GB RAM and was running Debian Linux (Squeeze) VM and Windows XP SP3 respectively.

6.7.2 Exploitable Bug Detection

We downloaded 29 different vulnerable programs to check the effectiveness of MAYHEM. Table 6.1 summarizes our results. Experiments were performed on stripped unmodified binaries on both Linux and Windows. One of the Windows applications MAYHEM exploited (`Dizzy`) was a packed binary.

Column 3 shows the type of exploits that MAYHEM detected as we described in §6.5. Column 4 shows the symbolic sources that we considered for each program. There are

	Program	Exploit Type	Input Source	Input Size	Symb. Mem.	Precondition	Advisory ID.	Exploit Time (s)
Linux	**A2ps**	Stack Overflow	Env. Vars	550		crashing	EDB-ID-816	189
	Aeon	Stack Overflow	Env. Vars	1000		length	CVE-2005-1019	10
	Aspell	Stack Overflow	Stdin	750		crashing	CVE-2004-0548	82
	Atphttpd	Stack Overflow	Network	800	✓	crashing	CVE-2000-1816	209
	FreeRadius	Stack Overflow	Env.	9000		length	Zero-Day	133
	GhostScript	Stack Overflow	Arg.	2000		prefix	CVE-2010-2055	18
	Glftpd	Stack Overflow	Arg.	300		length	OSVDB-ID-16373	4
	Gnugol	Stack Overflow	Env.	3200		length	Zero-Day	22
	Htget	Stack Overflow	Env. vars	350	✓	length	N/A	7
	Htpasswd	Stack Overflow	Arg.	400	✓	prefix	OSVDB-ID-10068	4
	Iwconfig	Stack Overflow	Arg.	400		length	CVE-2003-0947	2
	Mbse-bbs	Stack Overflow	Env. vars	4200	✓	length	CVE-2007-0368	362
	nCompress	Stack Overflow	Arg.	1400		length	CVE-2001-1413	11
	OrzHttpd	Format String	Network	400		length	OSVDB-ID-60944	6
	PSUtils	Stack Overflow	Arg.	300		length	EDB-ID-890	46
	Rsync	Stack Overflow	Env. Vars	100	✓	length	CVE-2004-2093	8
	SharUtils	Format String	Arg.	300		prefix	OSVDB-ID-10255	17
	Socat	Format String	Arg.	600		prefix	CVE-2004-1484	47
	Squirrel Mail	Stack Overflow	Arg.	150		length	CVE-2004-0524	2
	Tipxd	Format String	Arg.	250		length	OSVDB-ID-12346	10
	xGalaga	Stack Overflow	Env. Vars	300		length	CVE-2003-0454	3
	Xtokkaetama	Stack Overflow	Arg.	100		crashing	OSVDB-ID-2343	10
Windows	**Coolplayer**	Stack Overflow	Files	210	✓	crashing	CVE-2008-3408	164
	Destiny	Stack Overflow	Files	2100	✓	crashing	OSVDB-ID-53249	963
	Dizzy	SEH Overwrite	Arg.	519	✓	crashing	EDB-ID-15566	13,260
	GAlan	Stack Overflow	Files	1500	✓	prefix	OSVDB-ID-60897	831
	GSPlayer	Stack Overflow	Files	400	✓	crashing	OSVDB-ID-69006	120
	Muse	Stack Overflow	Files	250	✓	crashing	OSVDB-ID-67277	481
	Soritong	SEH Overwrite	Files	1000	✓	crashing	CVE-2009-1643	845

Table 6.1: List of programs that MAYHEM demonstrated as exploitable.

examples from all the symbolic input sources that MAYHEM supports, including command-line arguments (Arg.), environment variables (Env. Vars), network packets (Network) and symbolic files (Files). Column 5 is the size of each symbolic input. Column 6 describes the

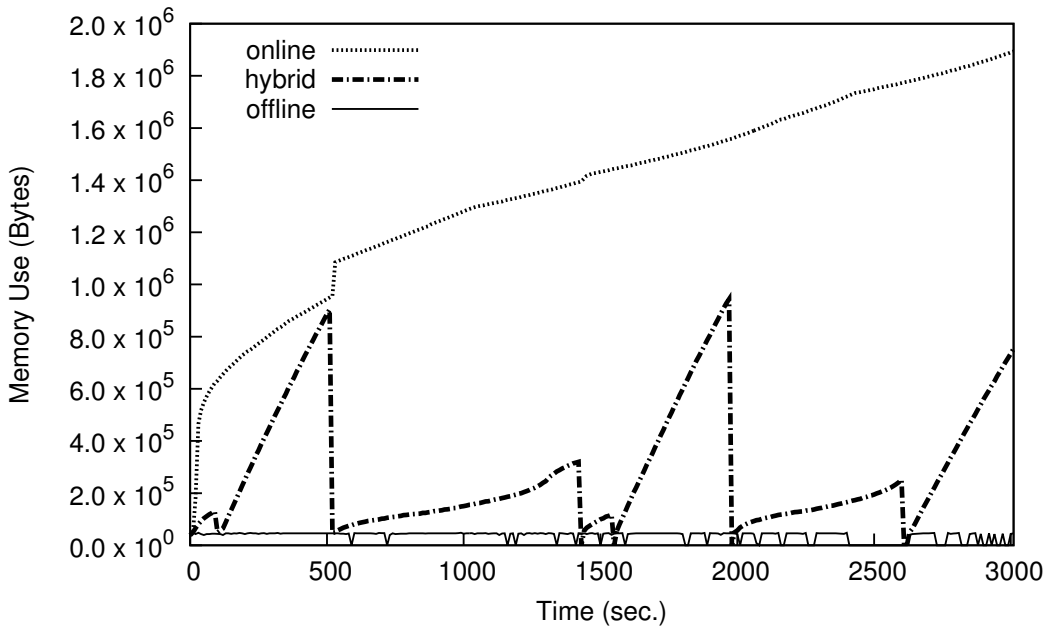


Figure 6.6: Memory use in online, offline, and hybrid mode.

precondition types that we provided to MAYHEM, for each of the 29 programs. They are split into three categories: length, prefix and crashing input as described in §6.3.4. Column 7 shows the advisory reports for all the demonstrated exploits. In fact, MAYHEM found 2 zero-day exploits for two Linux applications, both of which we reported to the developers.

The last column contains the exploit generation time for the programs that MAYHEM analyzed. We measured the exploit generation time as the time taken from the start of analysis until the creation of the first working exploit. The time required varies greatly with the complexity of the application and the size of symbolic inputs. The fastest program to exploit was the Linux wireless configuration utility `iwconfig` in 1.90 seconds and the longest was the Windows program `Dizzy`, which took about 4 hours.

6.7.3 Scalability of Hybrid Symbolic Execution

We measured the effectiveness of hybrid symbolic execution across two scaling dimensions: memory use and speed.

Less Memory-Hungry than Online Execution. Figure 6.6 shows the average memory use of MAYHEM over time while analyzing a utility in coreutils (`echo`) with online, offline and hybrid execution. After a few minutes, online execution reaches the maximum number of live interpreters and starts terminating execution paths. At this point, the memory keeps increasing linearly as the paths we explore become deeper. Note that at the beginning, hybrid execution consumes as much memory as online execution without exceeding the memory threshold, and utilizes memory resources more aggressively than offline execution throughout the execution. Offline execution requires much less memory (less than 500KB on average), but at a performance cost, as demonstrated below.

Additionally, we ran a Windows GUI program (`MiniShare`) to compare the throughput between offline and hybrid execution. We chose this program because it does not require user interaction (e.g., mouse click), to start symbolic execution. We ran the program for 1 hour for each execution mode. Hybrid execution was 10× faster than offline execution.

6.7.4 Handling Symbolic Memory in Real-World Applications

Recall from §6.4, index-based memory modeling enables MAYHEM to reason about symbolic indices. Our experiments from Table 6.1 show that more than 40% of the programs required symbolic memory modeling (column 6) to exploit. In other words, MAYHEM—after several hours of analysis—was unable to generate exploits for these programs without index-based memory modeling. To understand why, we evaluated our index-based memory modeling optimizations on the `atphttpd` server.

Bounds Resolution Table 6.2 shows the time taken by MAYHEM to find a vulnerability in atphttpd using different levels of optimizations for the bounds resolution algorithm. The times include exploit detection but not exploit generation time (since it is not affected by the bounds resolution algorithm). Row 3 shows that VSA reduces the average number of queries

	L Hits	R Hits	Misses	# Queries	Time (sec)
No opt.	N/A	N/A	N/A	217,179	1,841
+ VSA	N/A	N/A	N/A	49,424	437
+ R cache	N/A	3996	7	10,331	187
+ L cache	3940	56	7	242	77

Table 6.2: Effectiveness of bounds resolution optimizations. The L and R caches are respectively the Lemma and Refinement caches as defined in §6.4.

Formula Representation	**Time (sec.)**
Unbalanced binary tree	1,754
Balanced binary tree	425
Balanced binary tree + Linearization	192

Table 6.3: Performance comparison for different IST representations.

to the SMT solver from ∼54 to ∼14 queries per symbolic memory access, and reduces the total time by 75%.

Row 4 shows shows the number of queries when the refinement cache (R cache) is enabled on top of VSA. The R cache reduces the number of necessary binary searches to from 4003 to 7, resulting in a 57% speedup. The last row shows the effect of the lemma cache (L cache) on top of the other optimizations. The L cache takes most of the burden off the R cache, thus resulting in an additional 59% speedup. We do not expect the L cache to always be that efficient, since it relies heavily on the independence of formulas in the path predicate. The cumulative speedup was 96%.

Index Search Tree Representation. Recall from §6.4.2 MAYHEM models symbolic memory loads as ISTs. To show the effectiveness of this optimization we ran `atphttpd` with three different formula representations (shown in Table 6.3). The balanced IST was more

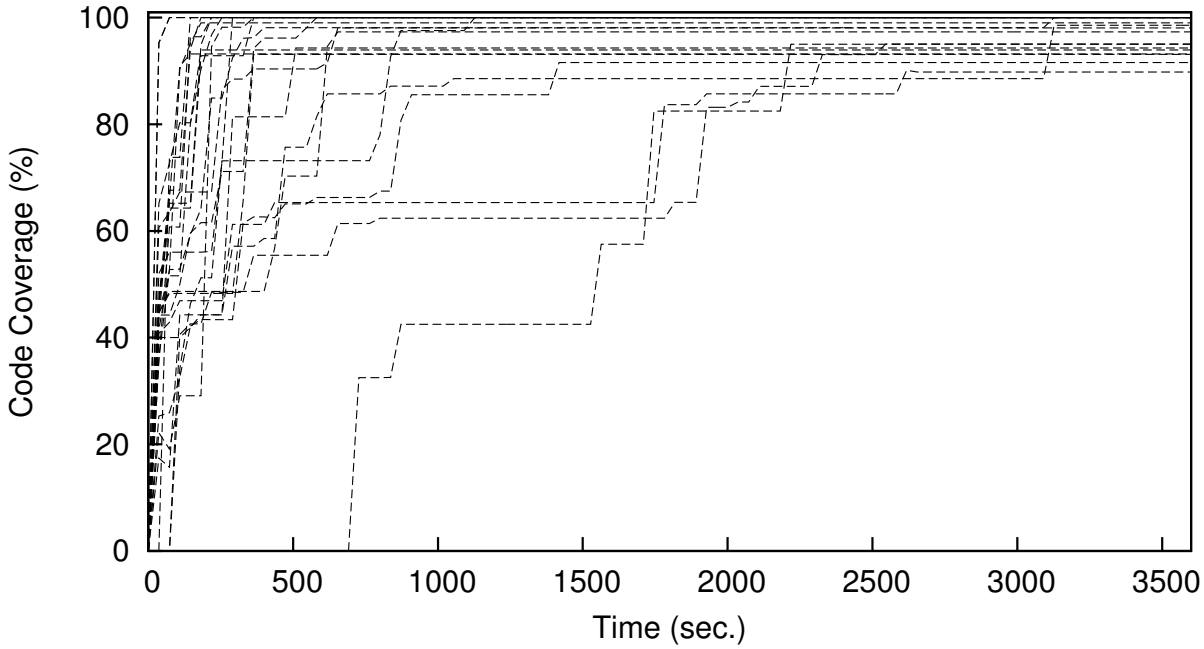


Figure 6.7: Code coverage achieved by MAYHEM as time progresses for 25 coreutils applications.

than 4× faster than the unbalanced binary tree representation, and with linearization of the formula we obtained a cumulative 9× speedup. Note, that with symbolic arrays (no ISTs) we were unable to detect an exploit within the time limit.

6.7.5 Mayhem Coverage Comparison

To evaluate MAYHEM's ability to cover new paths, we downloaded an open-source symbolic executor (KLEE) to compare the performance against MAYHEM. Note KLEE runs on source, while MAYHEM on binary.

We measured the code coverage of 25 coreutils applications as a function of time. MAYHEM ran for one hour, at most, on each of those applications. We used the generated test cases to measure the code coverage using the GNU gcov utility. The results are shown in Figure 6.7.

We used the 21 tools with the smallest code size, and 4 bigger tools that we selected. MAYHEM achieved a 97.56% average coverage per application and got 100% coverage on

	AEG		Mayhem			
Program	Time	LLVM	Time	ASM	Tainted ASM	Tained IL
iwconfig	0.506s	10,876	1.90s	394,876	2,200	12,893
aspell	8.698s	87,056	24.62s	696,275	26,647	133,620
aeon	2.188s	18,539	9.67s	623,684	7,087	43,804
htget	0.864s	12,776	6.76s	576,005	2,670	16,391
tipxd	2.343s	82,030	9.91s	647,498	2,043	19,198
ncompress	5.511s	60,860	11.30s	583,330	8,778	71,195

Table 6.4: AEG comparison: binary-only execution requires more instructions.

13 tools. For comparison, KLEE achieved 100% coverage on 12 coreutils without simulated system call failures (to have the same configuration as MAYHEM). Thus, MAYHEM seems to be competitive with KLEE for this data set. Note that MAYHEM is not designed specifically for maximizing code coverage. However, our experiments provide a rough comparison point against other symbolic executors.

6.7.6 Comparison against AEG

We picked 8 different programs from the AEG working examples [2] and ran both tools to compare exploit generation times on each of those programs using the same configuration (Table 6.4). MAYHEM was on average 3.4× slower than AEG. AEG uses source code, thus has the advantage of operating at a higher-level of abstraction. At the binary level, there are no types and high-level structures such as functions, variables, buffers and objects. The number of instructions executed (Table 6.4) is another factor that highlights the difference

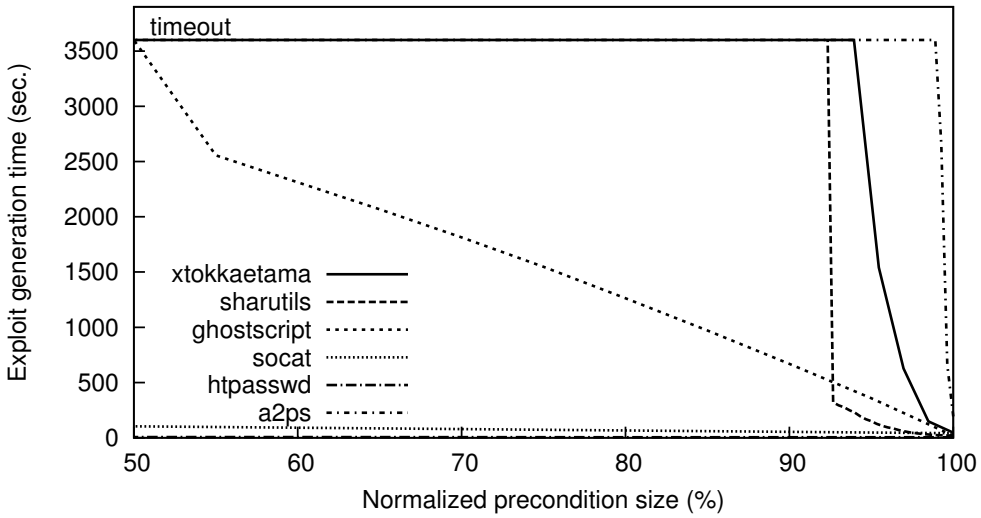


Figure 6.8: Exploit generation time versus precondition size.

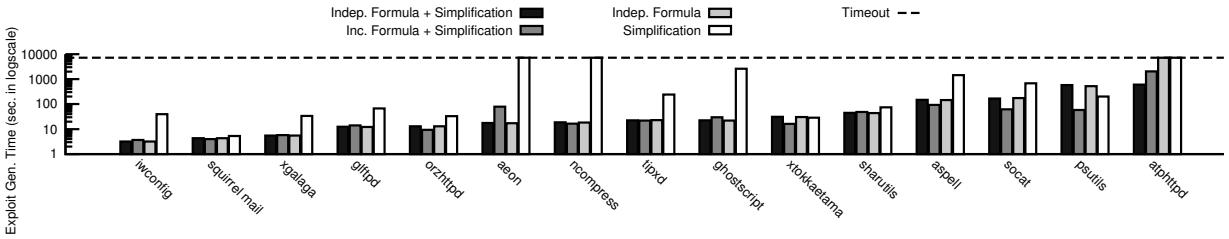


Figure 6.9: Exploit generation time of MAYHEM for different optimizations.

between source and binary-only analysis. Considering this, we believe this is a positive and competitive result for MAYHEM.

Precondition Size. As an additional experiment, we measured how the presence of a precondition affects exploit generation times. Specifically, we picked 6 programs that require a crashing input to find an exploitable bug and started to iteratively decrease the size of the precondition and measured exploit generation times. Figure 6.8 summarizes our results in terms of normalized precondition sizes—for example, a normalized precondition of 70% for a 100-byte crashing input means that we provide 70 bytes of the crashing input as a precondition to MAYHEM. While the behavior appeared to be program-dependent, in most of the programs we observed a sudden phase-transition, where the removal of a single character

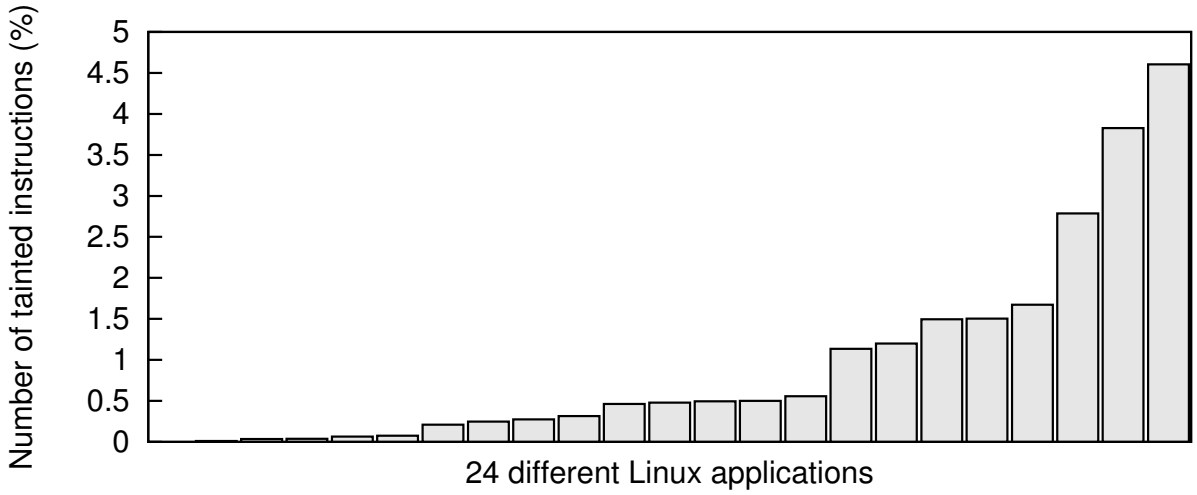


Figure 6.10: Tainted instructions (%) for 24 Linux applications.

could cause MAYHEM to not detect the exploitable bug within the time limit. We believe this to be an interesting topic for future work in the area.

6.7.7 Performance Tuning

Formula Optimizations. Recall from §6.3.5 MAYHEM uses various optimization techniques to make solver queries faster. To compare against our optimized version of MAYHEM, we turned off some or all of these optimizations.

We chose 15 Linux programs to evaluate the speedup obtained with different levels of optimizations turned on. Figure 6.9 shows the head-to-head comparison (in exploit finding and generation times) between 4 different formula optimization options. Algebraic simplifications usually speed up our analysis and offer an average speedup of 10% for the 15 test programs. Significant speedups occur when the independent formula optimization is turned on along with simplifications, offering speedups of 10-100×.

Z3 supports incremental solving, so as an additional experiment, we measured the exploit generation time with Z3 in incremental mode. In most cases solving times for incremental formulas are comparable to the times we obtain with the independent formulas optimization.

In fact, in half of our examples (7 out of 15) incremental formulas outperform independent formulas. In contrast to previous results, this implies that using the solver in incremental mode can alleviate the need for many formula simplifications and optimizations. A downside of using the solver in incremental mode was that it made our symbolic execution state mutable—and thus was less memory efficient during our long-running tests.

Tainted Instructions. Only tainted instruction blocks are evaluated symbolically by MAYHEM—all other blocks are executed natively. Figure 6.10 shows the percentage of tainted instructions for 24 programs (taken from Table 6.1). More than 95% of instructions were not tainted in our sample programs, and this optimization gave about 8× speedup on average.

6.8 Discussion

Most of the work presented in this chapter focuses on exploitable bug finding. However, we believe that the main techniques can be adapted to other application domains under the context of symbolic execution. We also believe that our hybrid symbolic execution and index-based memory modeling represent new points in the design space of symbolic execution.

We stress that the intention of MAYHEM is informing a user that an exploitable bug exists. The exploit produced is intended to demonstrate the severity of the problem, and to help debug and address the underlying issue. MAYHEM makes no effort to bypass OS defenses such as ASLR and DEP, which will likely protect systems against exploits we generate. However, our previous work on Q [49] shows that a broken exploit (that no longer works because of ASLR and DEP), can be automatically transformed—with high probability—into an exploit that bypasses both defenses on modern OSes. While we could feed the exploits generated by MAYHEM directly into Q, we do not explore this possibility in this thesis.

Limitations: MAYHEM does not have models for all system/library calls. The current implementation models about 30 system calls in Linux, and 12 library calls in Windows. To

analyze larger and more complicated programs, more system calls need to be modeled. This is an artifact of performing per-process symbolic execution. Whole-system symbolic executors such as S2E [29] or BitBlaze [32] can execute both user and kernel code, and thus do not have this limitation. The down-side is that whole-system analysis can be much more expensive, because of the higher state restoration cost and the time spent analyzing kernel code. Another limitation is that MAYHEM can currently analyze only a single execution thread on every run. MAYHEM cannot handle multi-threaded programs when threads interact with each other (through message-passing or shared memory). Last, MAYHEM executes only tainted instructions, thus it is subject to all the pitfalls of taint analysis, including undertainting, overtainting and implicit flows [110].

Future Work: Our experiments show that MAYHEM can generate exploits for standard vulnerabilities such as stack-based buffer overflows and format strings. An interesting future direction is to extend MAYHEM to handle more advanced exploitation techniques such as exploiting heap-based buffer overflows, use-after-free vulnerabilities, and information disclosure attacks. At a high level, it should be possible to detect such attacks using safety properties similar to the ones MAYHEM currently employs. However, it is still an open question how the same techniques can scale and detect such exploits in bigger programs.

6.9 Related Work

Brumley *et al.* [90] introduced the automatic *patch-based* exploit generation (APEG) challenge. APEG used the patch to point out the location of the bug and then used slicing to construct a formula for code paths from input source to vulnerable line. MAYHEM finds vulnerabilities and vulnerable code paths itself. In addition, APEG's notion of an exploit is more abstract: any input that violates checks introduced by the path are considered exploits. Here we

consider specifically control flow hijack exploits, which were not automatically generated by APEG.

Heelan [91] was the first to describe a technique that takes in a crashing input for a program, along with a jump register, and automatically generates an exploit. Our research explores the state space to find such crashing inputs.

AEG [2] was the first system to tackle the problem of both identifying exploitable bugs and automatically generating exploits. AEG worked solely on source code and introduced preconditioned symbolic execution as a way to focus symbolic execution towards a particular part of the search space. MAYHEM is a logical extension of AEG to binary code. In practice, working on binary code opens up automatic exploit generation to a wider class of programs and scenarios. In Section 4.5 we provide a more complete bibliography of work in AEG.

There are several binary-only symbolic execution frameworks such as Bouncer [22], BitFuzz [24], BitTurner [111] FuzzBall [112], McVeto [31], SAGE [12], and S2E [29], which have been used in a variety of application domains. The main question we tackle in MAYHEM is scaling to find and demonstrate exploitable bugs. The hybrid symbolic execution technique we present in this chapter is completely different from hybrid concolic testing [55], which interleaves random testing with concolic execution to achieve better code coverage.

6.10 Conclusion

We presented MAYHEM, a tool for automatically finding exploitable bugs in binary (i.e., executable) programs in an efficient and scalable way. To this end, MAYHEM introduces a novel hybrid symbolic execution scheme that combines the benefits of existing symbolic execution techniques (both online and offline) into a single system. We also present index-based memory modeling, a technique that allows MAYHEM to discover more exploitable bugs

at the binary-level. We used MAYHEM to analyze 29 applications and automatically identified and demonstrated 29 exploitable vulnerabilities.

6.11 Acknowledgements

We thank our shepherd, Cristian Cadar and the anonymous reviewers for their helpful comments and feedback. This research was supported by a DARPA grant to CyLab at Carnegie Mellon University (N11AP20005/D11AP00262), a NSF Career grant (CNS0953751), and partial CyLab ARO support from grant DAAD19-02-1-0389 and W911NF-09-1-0273. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

Chapter 7

Veritesting

Prepare properly, and no one will ask for more.

— My mother, Sofia, *Before exams.*

In this chapter, we present MergePoint, a new binary-only symbolic execution system for large-scale testing of commodity off-the-shelf (COTS) software. MergePoint introduces *veritesting*, a new technique that employs static symbolic execution to amplify the effect of dynamic symbolic execution. Veritesting allows MergePoint to find twice as many bugs, explore orders of magnitude more paths, and achieve higher code coverage than previous dynamic symbolic execution systems. MergePoint is currently running daily on a 100 node cluster analyzing 33,248 Linux binaries; has generated more than 15 billion SMT queries, 200 million test cases, 2,347,420 crashes, and found 11,687 bugs in 4,379 distinct applications.

7.1 Introduction

Symbolic execution is a popular automatic approach for testing software and finding bugs. Over the past decade, numerous symbolic execution tools have appeared—both in academia and industry—demonstrating the effectiveness of the technique in finding crashing inputs [19,

20], generating test cases with high coverage [21], exposing software vulnerabilities [62], and generating exploits [3].

Symbolic execution is attractive because it systematically explores the program and produces real inputs. Symbolic execution works by automatically translating a program fragment to a logical formula. The logical formula is satisfied by inputs that have a desired property, e.g., they execute a specific path or violate safety.

At a high level, there are two main approaches for generating formulas. First, dynamic symbolic execution (DSE) explores programs and generates formulas on a per-path basis. Second, static symbolic execution (SSE) translates program statements into formulas, where the formulas represent the desired property over any path within the selected statements.

In this chapter we describe MergePoint, a system for automatically checking all programs in a Linux distribution using a new technique called *veritesting*. The path-based nature of DSE introduces significant overhead when generating formulas, but the formulas themselves are easy to solve. The statement-based nature of SSE has less overhead and produces more succinct formulas that cover more paths, but the formulas are harder to solve. Veritesting *alternates* between SSE and DSE. The alternation mitigates the difficulty of solving formulas, while alleviating the high overhead associated with a path-based DSE approach. In addition, DSE systems replicate the path-based nature of concrete execution, allowing them to handle cases such as system calls and indirect jumps where static approaches would need summaries or additional analysis. Alternating allows MergePoint with veritesting to switch to DSE-based methods when such cases are encountered.

MergePoint operates on 32-bit Linux binaries and does not require any source information (e.g., debugging symbols). We have systematically used MergePoint to test and evaluate veritesting on 33,248 binaries from Debian Linux. The binaries were collected by downloading and mining for executable programs all available packages from the Debian main repository. We did not pick particular binaries or a dataset that would highlight specific

aspects of our system; instead we focus on our system as experienced in the general case. The large dataset allows us to explore questions with high fidelity and with a smaller chance of per-program sample bias. The binaries are exactly what runs on millions of systems throughout the world.

We demonstrate that MergePoint with veritesting beats previous techniques in the three main metrics: bugs found, node coverage, and path coverage. In particular, MergePoint has found **11,687 distinct bugs** (by stack hash) in 4,379 different programs. Overall, MergePoint has generated over 15 billion SMT queries and created over 200 million test cases. Out of the 11,687 bugs, 224 result in user input overwriting the instruction pointer, and we have confirmed shell-spawning exploits for 152.

Our main contributions are as follows. First, we propose a new technique for symbolic execution called veritesting. Second, we provide and study in depth the first system for testing every binary in an OS distribution using symbolic execution. Our experiments reduce the chance of per-program or per-dataset bias. We evaluate MergePoint with and without veritesting and show that veritesting outperforms previous work on all three major metrics. Finally, we improve open source software by finding over 10,000 bugs and generating millions of test cases. Debian maintainers have already incorporated 202 patches due to our bug reports. We have made our data available on our website [113].

7.2 Overview

At a high level, symbolic execution can be partitioned into two main approaches: dynamic symbolic execution for testing, and static symbolic execution for verification. Dynamic approaches work by generating per-path formulas to test specific paths, while static-based approaches generate formulas over entire programs with the goal of verifying overall safety. Our main insight is to carefully *alternate* between the two schemes to harness the benefits

of both while mitigating path explosion in dynamic approaches and solver blowup in static approaches. In particular, we start with dynamic symbolic execution, but switch to a static verification-based approach opportunistically. When we switch to static mode, we only check *program fragments* with the goal of testing, not verification. While we are not the first to suggest using static and dynamic techniques in combination, the careful application of alternation as proposed in veritesting reduces overall overhead, and results in improved performance along key metrics. Previous approaches typically lost on at least one metric, and sometimes several.

In this section we provide a high-level overview of standard metrics, the key parts of dynamic and static algorithms, as well as the tradeoffs between approaches.

7.2.1 Testing Metrics

Testing systems, including dynamic symbolic execution systems, are typically evaluated using three metrics: 1) number of real bugs found, 2) node coverage, and 3) path coverage.

Node (or code or line or statement) coverage measures the percentage of code covered by generated test cases with respect to the entire application. Node coverage is an effective way of measuring the performance of a test case generation system [114] and has been used repeatedly to measure symbolic execution systems' performance [21, 26].

Path coverage measures the percentage of program paths analyzed. Unlike node coverage, which has a finite domain (the total number of program statements), many programs have a potentially infinite number of paths (e.g., a server) and measuring the path coverage is not possible. In our evaluation, we use three distinct metrics for approximating path coverage §7.6.3.

The number of unique bugs is measured by counting the number of unique stack hashes [115] among crashes. We report bugs only when a generated test case can produce a core file during concrete execution.

All three metrics are important, and none dominates in all scenarios. For example, node coverage is useful, but even 100% node coverage may fail to find real bugs. Also, it may be possible to achieve 100% node coverage but never execute a loop more than once. Bugs that require several iterations to trigger, e.g., buffer overflows, will be missed. Testing more paths is better, but an analysis could game the metric by simply iterating over fast-to-execute loops more times and avoiding slow execution paths. Again, bugs may be missed and nodes may not be covered. One could find all bugs, but never know it because not all paths are exhausted.

7.2.2 Dynamic Symbolic Execution (DSE)

Algorithm 3 presents the core steps in dynamic symbolic execution. The algorithm operates on a representative imperative language with assignments, assertions and conditional jumps (adapted from previous work [26]). A symbolic executor maintains a state (ℓ, Π, Γ) where ℓ is the address of the current instruction, Π is the path predicate, and Γ is a symbolic store that maps each variable to either a concrete value or an expression over input variables. A satisfying assignment, typically checked by a SAT or SMT solver, is an assignment of values to symbolic input variables that will execute the same execution path. An unsatisfiable path predicate means the selected path is infeasible.

On line 1, the algorithm initializes the worklist with a state pointing to the start of the program. The `pickNext` function selects the next state to continue executing, and removes it from the worklist $\mathcal{S}$. There are a variety of search heuristics for selecting the next instruction to execute, including starting with a concrete trace [20, 30], generational search [13], DFS, and BFS. Symbolic execution switches over the instruction types in line 4. Safety checks are performed with assertions. For example, every memory dereference is preceded by an assertion that checks whether the pointer is in bounds. The semantics of assignment, assert,

Algorithm 3: Dynamic Symbolic Execution Algorithm with and without Veritesting

Input: Initial location ℓ_0, instruction decoder `instrAt`
Data: Worklist S, path predicate Π, symbolic store Γ

```
1  S ← {(ℓ0, true, ∅)}                                          // initial worklist
2  while S ≠ ∅ do
3  |  ((ℓ, Π, Γ), S) ← pickNext(S)
   |  // Symbolically execute the next instruction
4  |  switch instrAt(ℓ) do
5  |  |  case v := e                                            // assignment
6  |  |  |  S ← {(succ(ℓ), Π, Γ[veval → (Γ, e)])}
7  |  |  case if (e) goto ℓ'                                   // conditional jump
8  |  |  |  e ← eval(Γ, e)
9  |  |  |  if (isSat(Π ∧ e) ∧ isSat(Π ∧ ¬e)) then
10 |  |  |  |  // DSE forks 2 states
11 |  |  |  |  S ← {(ℓ', Π ∧ e, Γ), (succ(ℓ), Π ∧ ¬e, Γ)}
10 |  |  |  |  // Veritesting integration
11 |  |  |  |  S ← ∅
12 |  |  |  |  CFG ← CFGRecovery(ℓ, Π)
13 |  |  |  |  CFG_e, TransitionPoints ← CFGReduce(CFG)
14 |  |  |  |  OUT ← StaticSymbolic(CFG_e, Π, Γ)
15 |  |  |  |  for Point ∈ TransitionPoints do
16 |  |  |  |  |  if OUT[Point] ≠ ∅ then
17 |  |  |  |  |  |  S ← OUT[Point] ∪ S
18 |  |  |  |  S ← Finalize(S)
19 |  |  |  else if isSat(Π ∧ e) then
20 |  |  |  |  S ← {(ℓ', Π ∧ e, Γ)}
21 |  |  |  else  S ← {(succ(ℓ), Π ∧ ¬e, Γ)}
22 |  |  case assert(e)                                         // assertion
23 |  |  |  e ← eval(Γ, e)
24 |  |  |  if isSat(Π ∧ ¬e) then  reportBug(Π ∧ ¬e)
25 |  |  |
26 |  |  |  S ← {(succ(ℓ), Π ∧ e, Γ)}
27 |  |  case halt: continue                                   // end of path
28 |  S ← S ∪ S
```

and halt are all straightforward. The central design point we focus on in this chapter is handling a branch instruction, shown in line 7.

The two instances of line 11 contrast our approach with others'. In DSE, whenever both branches are feasible, two new states are added to the worklist (one for the true branch and one for the false), a process we refer to as "forking". Each one of the forked executors is later chosen from the worklist and explored independently.

Advantages/Disadvantages. Forking executors and analyzing a single path at a time has benefits: the analysis code is simple, solving the generated path predicates is typically fast (e.g., in SAGE [62] 99% of all queries takes less than 1 second) since we only reason about a single path, and the concrete path-specific state resolves several practical problems. For example, executors can execute hard-to-model functionality concretely (e.g., system calls), side-effects such as allocating memory in each DSE path are reasoned about independently without extra work, and loops are unrolled as the code executes. The disadvantage is the *path (or state) explosion*[1] problem: the number of executors can grow exponentially in the number of branches. The path explosion problem is the motivation for our veritesting algorithm §7.3.

7.2.3 Static Symbolic Execution (SSE)

Static Symbolic Execution (SSE) is a verification technique for representing a program as a logical formula. Potential vulnerabilities are encoded as logical assertions that will falsify the formula if safety is violated. Calysto [33] and Saturn [116, 117] are example SSE tools. Because SSE checks programs, not paths, it is typically employed to verify the absence of bugs. As we will see, veritesting repurposes SSE techniques for testing program fragments instead of verifying complete programs.

The main change is on line 11 of Algorithm 3. Modern SSE algorithms can summarize the effects of both branches at path confluence points. In contrast, DSE traditionally forks off two executors at the same line, which remain subsequently forever independent. Due

[1]Depending on the context, the two terms may be used interchangeably [25, 26]—an "execution state" corresponds to a program path to be explored.

to space, we do not repeat complete SSE algorithms here, and refer the reader to previous work [118, 116, 33]. (§7.3 shows our SSE algorithm using a dataflow framework.)

Advantages/Disadvantages. Unlike DSE, SSE does not suffer from path explosion. All paths are encoded in a single formula that is then passed to the solver (note the solver may still have to reason internally about an exponential number of paths). For acyclic programs, existing techniques allow generating compact formulas of size $O(n^2)$ [34, 76], where n is the number of program statements. Despite these advantages over DSE, state-of-the-art tools still have trouble scaling to very large programs [119, 27, 26]. Problems include the presence of loops (how many times should they be unrolled?), formula complexity (are the formulas solvable if we encode loops and recursion? [117]), the absence of concrete state (what is the concrete environment the program is running in?), as well as unmodeled behavior (a kernel model is required to emulate system calls). Another hurdle is completeness: for the verifier to prove absence of bugs, *all* program paths must be checked.

7.3 Veritesting

DSE has proven to be effective in analyzing real world programs [21, 12]. However, the path explosion problem can severely reduce the effectiveness of the technique. For example, consider the following 7-line program that counts the occurrences of the character 'B' in an input string:

```
1  int counter = 0, values = 0;
2  for ( i = 0 ; i < 100 ; i ++ ) {
3      if (input[i] == 'B') {
4          counter ++;
5          values += 2;
6      } }
```

```
7  if (counter == 75)      bug ();
```

The program above has 2^{100} possible execution paths. Each path must be analyzed separately by DSE, thus making full path coverage unattainable for practical purposes. In contrast, two testcases suffice for obtaining full code coverage: a string of 75 'B's and a string with no 'B's. However, finding such test cases in the 2^{100} state space is challenging[2]. We ran the above program with several state-of-the-art symbolic executors, including KLEE [21], S2E [29], Mayhem [3] and Cloud9 with state merging [26]. None of the above systems was able to find the bug within a 1-hour time limit (they ran out of memory or kept running). Veritesting allows us to find the bug and obtain full path coverage in 47 seconds on the same hardware.

Veritesting starts with DSE, but switches to an SSE-style approach when we encounter code that—similar to the example above—does not contain system calls, indirect jumps, or other statements that are difficult to precisely reason about statically. Once in SSE mode, veritesting performs analysis on a dynamically recovered CFG and identifies a core of statements that are easy for SSE, and a frontier of hard-to-analyze statements. The SSE algorithm summarizes the effects of all paths through the easy nodes up to the hard frontier. Veritesting then switches back to DSE to handle the cases that are hard to treat statically.

Conceptually, the closest recent work to ours is dynamic state merging (DSM) by Kuznetsov *et al.* [26]. DSM maintains a history queue of DSE executors. Two DSEs may merge (depending on a separate and independent heuristic for SMT query difficulty) if they coincide in the history queue. Fundamentally, however, DSM still performs per-path execution, and only opportunistically merges. Veritesting always merges, using SSE (not DSE) on all statements within a fixed lookahead. The result is Veritesting formulas cover more paths than DSE

[2]For example, $\binom{100}{75} \approx 2^{78}$ paths reach the buggy line of code. The probability of a random path selection strategy finding one of those paths is approximately $2^{78}/2^{100} = 2^{-22}$.

(at the expense of longer SMT queries), but avoid the overhead of managing a queue and merging path-based executors.

In the rest of this section, we present the main algorithm and the details of the technique.

7.3.1 The Algorithm

In default mode, MergePoint behaves as a typical dynamic concolic executor [30]. It starts exploration with a concrete seed and explores paths in the neighborhood of the original seed following a generational search strategy [12]. MergePoint does not always fork when it encounters a symbolic branch. Instead, MergePoint intercepts the forking process—as shown in line 11 of algorithm 3—of DSE and performs veritesting.

Algorithm 3 presents the high-level process of veritesting. The algorithm augments DSE with 4 new steps:

1. `CFGRecovery`: recovers the CFGreachable from the address of the symbolic branch (§7.3.2).

2. `CFGReduce`: takes in a CFG, and outputs candidate transition points and a CFG_e, an acyclic CFGwith edges annotated with the control flow conditions (§7.3.3). Transition points indicate program locations where DSE may continue.

3. `StaticSymbolic`: takes the acyclic CFG_e and current execution state, and uses SSE to build formulas that encompass all feasible paths in the CFG_e. The output is a mapping from CFG_e nodes to SSE states (§7.3.4).

4. `Finalize`: given a list of transition points and SSE states, returns the DSE executors to be forked (§7.3.5).

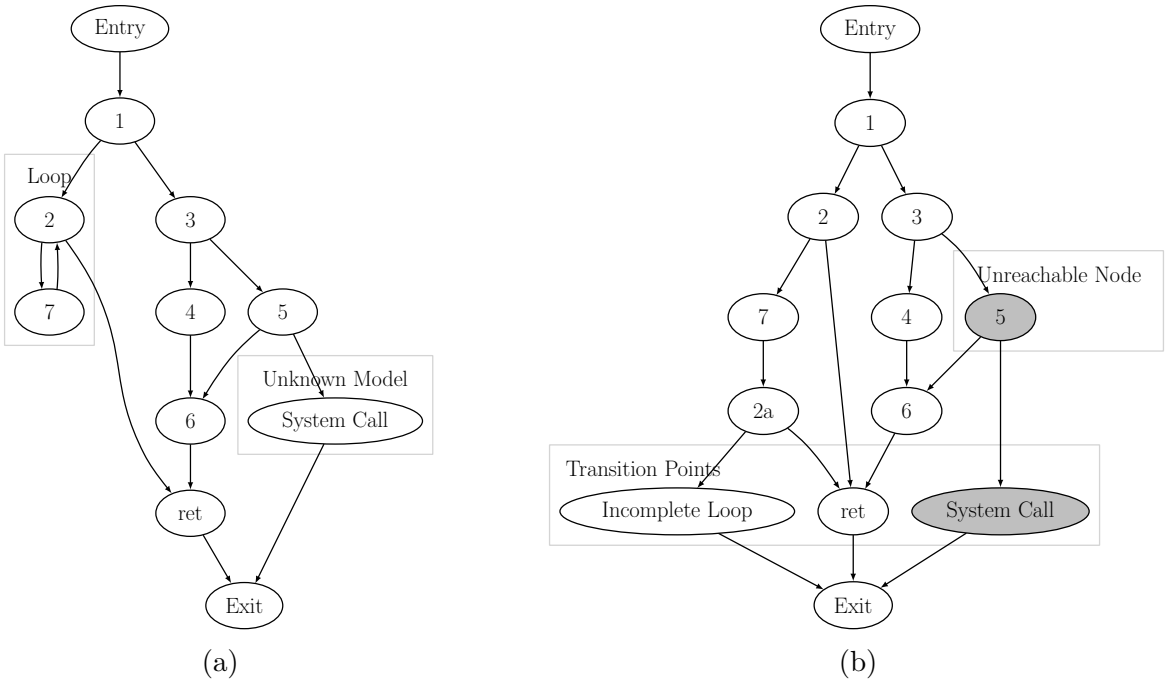


Figure 7.1: Veritesting on a program fragment with loops and system calls. (a) Recovered CFG. (b) CFG after transition point identification & loop unrolling. Unreachable nodes are shaded.

7.3.2 CFG Recovery

The goal of the CFG recovery phase is to obtain a partial control flow graph of the program, where the entry point is the current symbolic branch. We now define the notion of underapproximate and overapproximate CFG recovery.

A recovered CFG is an underapproximation if all edges of the CFG represent feasible paths. A recovered CFG is an overapproximation if all feasible paths in the program are represented by edges in the CFG. Statically recovering a perfect (non-approximate) CFG on binary code is known to be a hard problem and the subject of active research [120, 121]. A recovered CFG might be an underapproximation or an overapproximation, or even both in practice.

Veritesting was designed to handle both underapproximated and overapproximated CFGs without losing paths or precision (see §7.3.4). MergePoint uses the CFG recovery mecha-

nism from our Binary Analysis Platform (BAP) [41]. The algorithm is customized to stop recovery at function boundaries, system calls and unknown instructions.

The output of this step is a partial (possibly approximate) intra-procedural control flow graph. Unresolved jump targets (e.g., `ret`, `call`, etc.) are forwarded to a generic `Exit` node in the CFG. Figure 7.1a shows the form of an example CFG after the recovery phase.

7.3.3 Transition Point Identification & Unrolling

Once the CFG is obtained, MergePoint proceeds to identifying transition points. Transition points define the boundary of the SSE algorithm (where DSE will continue exploration). To calculate transition points, we require the notion of postdominators and immediate postdominators:

Definition 20 (Postdominator)**.** *A node d postdominates a node n, denoted as* pdom *(d,n), iff every path from n to the exit of the graph goes through d.*

Definition 21 (Immediate Postdominator)**.** *A node d immediately postdominates node n, denoted as* ipdom *(d,n), iff:* $\mathsf{pdom}(d,n) \wedge \neg\exists z \neq d : \mathsf{pdom}(d,z) \wedge \mathsf{pdom}(z,n)$.

Transition Points. For an entry node e ending in a symbolic branch, a transition point is defined as a node n such that $\mathsf{ipdom}(e,n)$. For a fully recovered CFG, a single transition point may be sufficient, e.g., the bottom node in Figure 7.1a. However, for CFGs with unresolved jumps or system calls, any predecessor of the `Exit` node will be a possible transition point (e.g., the `ret` node in Figure 7.1b). Transition points represent the frontier of the visible CFG, which stops at unresolved jumps, function boundaries and system calls. The number of transition points gives an upper-bound on the number of states that may be forked.

Unrolling Loops. Loop unrolling represents a challenge for static verification tools. However, MergePoint is dynamic and can concretely execute the CFGto identify how many

times each loop will execute. The number of concrete loop iterations determines the number of loop unrolls. MergePoint also allows the user to extend loops beyond the concrete iteration limit, by providing a minimum number of unrolls.

To make the CFGacyclic, back edges are removed and forwarded to a newly created node for each loop, e.g., the "Incomplete Loop" node in Figure 7.1b, which is a new transition point that will be explored if executing the loop more times is feasible. In a final pass, the edges of the CFGare annotated with the conditions required to follow the edge.

The end result of this step is a CFG_e and a set of transition points. Figure 7.1b shows an example CFG— without edge conditions—after transition point identification and loop unrolling.

7.3.4 Static Symbolic Execution

Given the CFG_e, MergePoint applies SSE to summarize the execution of multiple paths. Previous work [51] first converted the program to Gated Single Assignment (GSA) [122] and then performed symbolic execution. In MergePoint, we encode SSE as a single-pass dataflow analysis where GSA is computed on the fly. Table 7.1 presents the SSE algorithm, following standard notation [123, Section 9].

To illustrate the algorithm, we run SSE on the following example program:

```
if (x > 1) y = 1; else if (x < 42) y = 17;
```

Figure 7.2 shows the progress of the variable context as SSE iterates through the blocks. SSE starts from the entry of the CFG_e and executes basic blocks in topological order. Basic blocks contain straightline code and execution follows Algorithm 4, taking as input (from $IN[B]$) a gating path expression γ [51], and a variable context Γ and outputting the updated versions (for $OUT[B]$). γ enables multi-path SSE by encoding the conditionals required to follow an execution path using *ite* (if-then-else) expressions. For example, following the true branch

Algorithm 4: Veritesting Transfer Function

Input: Basic block B, Gating path expression γ, Variable context Γ

```
1 foreach inst ∈ B do
2     switch inst do
3         case v := e
4             Γ ← Γ[v → eval(Γ, e)]
5         case assert(e)
6             γ ← γ[Λ → ite(eval(Γ, e), Λ, ⊥)]
7 return γ, Γ
```

after the condition $(x > 1)$ in Figure 7.2 gives: $\gamma = ite(x > 1, \Lambda, \bot)$, where Λ denotes the taken path and $\bot$ the non-taken. The path predicate during SSE is obtained by substitution in the gating path expression: $\Pi = \gamma[\Lambda \rightarrow true, \bot \rightarrow false]$.

To compute the input set ($IN[B]$) for a basic block we apply a meet operation across all incoming states from predecessor blocks following Algorithm 5. The gating path expression is obtained for each incoming edge and then applied to the variable context. For example, for

Domain	Symbolic execution state (γ, Γ)
Direction	Forwards
Transfer Function	Algorithm 4
Boundary	Initial execution state (Λ, Γ_{init})
Initialize	$OUT[B] = (\bot, \emptyset)$
Dataflow Equations	$IN[B] = \bigwedge_{P,pred(B)} OUT[P]$ $OUT[B] = f_B(IN[B])$
Meet Function	Algorithm 5

Table 7.1: SSE as a dataflow algorithm. $IN[B]$ and $OUT[B]$ denote the input and output sets of basic block B.

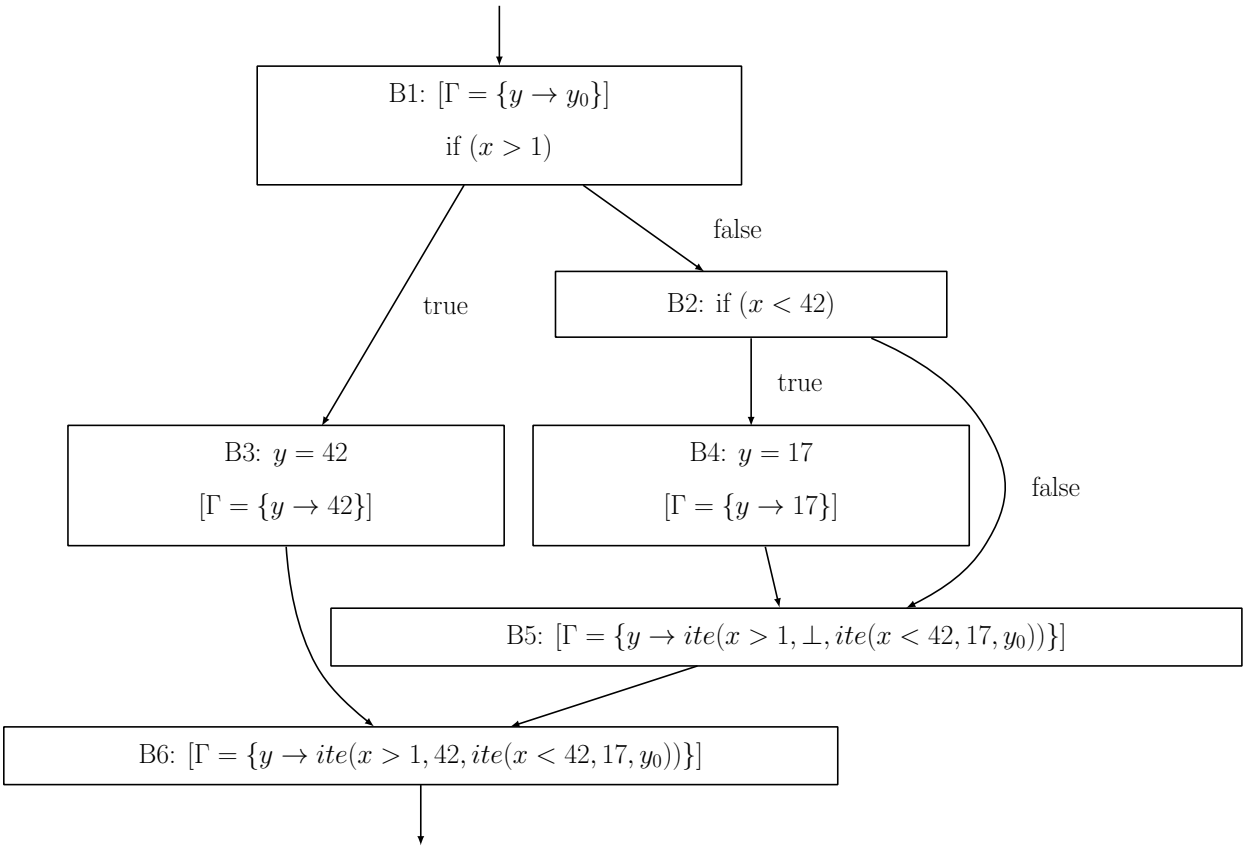


Figure 7.2: Variable context transformations during SSE.

the edge from B3 to B6 in Figure 7.2, Γ is updated to $\{y \to \gamma_3[\Lambda \to \Gamma[y]] = ite(x > 1, 42, \bot)\}$. To merge Γ's (or γ's) from paths that merge to the same confluence point, we apply the following recursive merge operation Ψ to each symbolic value:

$$\Psi(v_1, \bot) = v_1; \quad \Psi(\bot, v_2) = v_2;$$

$$\Psi(ite(e, v_1, v_2), ite(e, v_1', v_2')) = ite(e, \Psi(v_1, v_1'), \Psi(v_2, v_2'))$$

This way, at the last node of Figure 7.2, the value of y will be $\Psi(ite(x > 1, 42, \bot), ite(x > 1, \bot, ite(x < 42, 17, y_0)))$ which is merged to $ite(x > 1, 42, ite(x < 42, 17, y_0))$, capturing all possible paths. During SSE, MergePoint keeps a mapping from each traversed node to the corresponding state (OUT). Values from unmerged paths ($\bot$ values) can be immediately

Algorithm 5: Veritesting Meet Function

Input: Basic block B, Pred. blocks B_1, B_2, Path Predicate Π_{DSE}

```
1  function Context (B, Parent) begin
2      γ, Γ ← OUT(Parent); taken, e ← edge(Parent, B);
3      e ← eval(Γ, e); Π ← Π_DSE ∧ γ[Λ → true, ⊥ → false];
4      if taken ∧ isSat(Π ∧ e) then
5          return γ[Λ → ite(e, Λ, ⊥)], Γ
6      else if ¬taken ∧ isSat(Π ∧ ¬e) then
7          return γ[Λ → ite(e, ⊥, Λ)], Γ
8      else return ⊥, ∅;                                  // infeasible edge
9  γ1, Γ1 ← Context(B, B1); γ2, Γ2 ← Context(B, B2);
10 γ ← Ψ(γ1, γ2); Γ ← Γ1;
11 foreach v ∈ Γ2 do
12     Γ[v] = Ψ(γ1[Λ → Γ1[v]], γ2[Λ → Γ2[v]])
13 return γ, Γ
```

simplified, e.g., $ite(e, x, \bot) = x$. Similarly, nested *ite* expressions with the same condition can be reduced, e.g., $ite(e, ite(e, x, y), z)$ reduces to $ite(e, x, z)$.

Variable merging in Line 11 (Algorithm 5) is performing substitution multiple times on the same gating path expression γ. Memoizing the merged gating path expression $GE = \Psi(\gamma_1[\Lambda \rightarrow true], \gamma_2[\Lambda \rightarrow false])$, and reusing it to select variables with *ite* expressions improves expression sharing:

$$\Gamma[v] = \Psi(\mathsf{ite}(GE, \Gamma_1[v], \Gamma_2[v])$$

Given a gating path expression GE and evaluated variable contexts, the expression above creates only one new expression node—to build the *ite* expression; no other expression needs to be created.

ITE Merging. Path merging with SSE adds *ite* expressions to express the values of variables at confluence points. These variables may be used later in the execution to build more complex expressions. For example, consider two variables x, y, merged after conditioning on a symbolic expression e: $x = \mathsf{ite}(e, x_1, x_0)$, $y = \mathsf{ite}(e, y_1, y_0)$. Adding these variables together will result

in an expression $sum = \mathsf{ite}(e, x_1, x_0) + \mathsf{ite}(e, y_1, y_0)$ of size $|sum|_s^e = 8$ (note the duplication of the condition). The two *ite* expressions can be merged to $sum' = \mathsf{ite}(e, x_1 + y_1, x_0 + y_0)$, which has the same size $|sum'|_s^e = 8$, but fewer logical disjunctions (*ite* expressions). In contrast, if one of the variables is a constant, e.g., $y = 5$ in both branches, merging the two expressions may lead to larger expressions $|\mathsf{ite}(e, x_1 + 5, x_0 + 5)|_s^e > |\mathsf{ite}(e, x_1, x_0) + 5|_s^e$. Empirically, we have observed that merging *ite* expressions is beneficial, but measuring the exact effect of such simplifications is a good idea for future work (Chapter 8).

The Importance of GSA. Using gating path expressions, Algorithm 5 utilizes the structure of the control flow graph to express the values of variables across multiple paths. We found that preserving the structure of gating path expressions helps in two ways. First, the Ψ operator eliminates redundancies during merging—both in the path predicate, and in expressions found in the variable context. Second, the generated expressions are easily amenable to simplifications.

The first two implementation attempts of MergePoint did not rely on GSA for performing SSE. The first one, followed the construction of Hansen *et al.* [27], where merging two paths path predicates Π_1, Π_2 resulted in the disjuction of the two: $\Pi_1 \vee \Pi_2$. Because merged predicates may share structure, Hansen *et al.* post-process predicates through Espresso [124]'s heuristics to remove redundancies. The implementation worked on diamond-like examples, like the one showed in the beginning of Section 7.3. However, we were unable to make it scale for non-trivial examples. Integrating the simplification heuristics and making them effective on arbitrary conditionals and unstructured programs proved burdensome. Further, the SSE executor spent a substantial amount of time in simplifications to remove the redundancies that were introduced by disjoining the two predicates.

The second implementation attempt, followed the construction used by Kuznetsov *et al.* [26]. Instead of disjoining entire predicates, Kuznetsov *et al.* first remove all shared conjuncts between the two path predicates, effectively cancelling out the constraints that

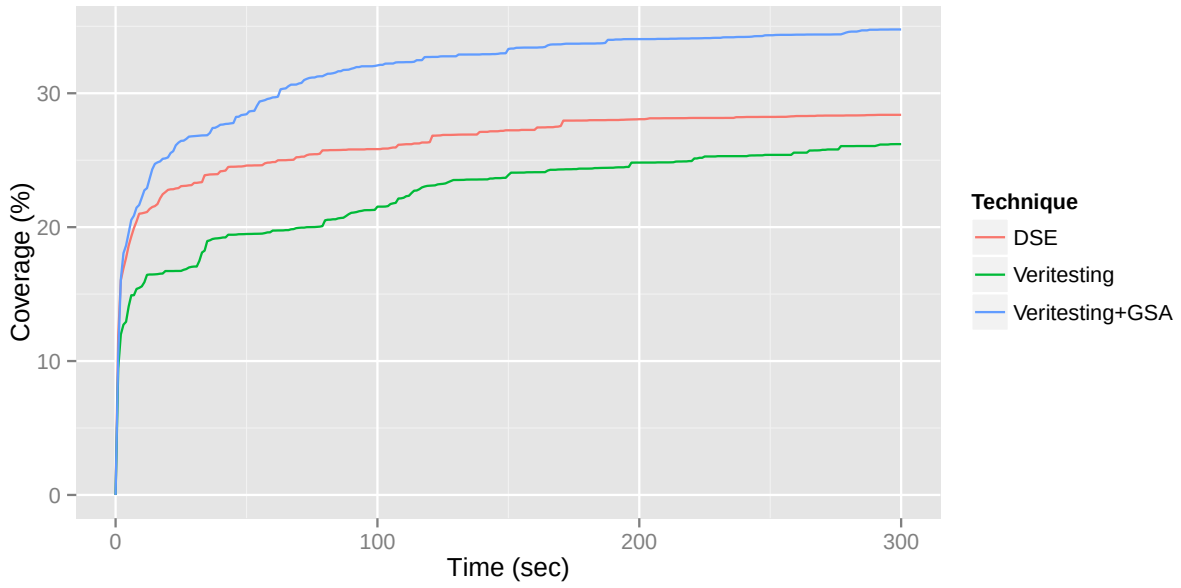


Figure 7.3: Code coverage with time on the first 100 programs from BIN (with and without GSA).

correspond to the shared path prefix (the exact algorithm is not described in their paper, but the Cloud9 [46] source code is available online). This approach worked better on more examples, but—mirroring Kuznetsov *et al.*'s results—did not improve code coverage. Figure 7.3 shows the performance difference between DSE, and veritesting with Cloud9's merging function (for reference, we are also including veritesting with GSA). The experiment was run for 5 minutes on the first (alphabetically-ordered) 100 programs from our BIN suite (Section 7.6). The inability to improve code coverage, led us to explore different merging constructions [51], and eventually GSA [122].

To show the importance of gating path expressions, we use a concrete example.

Example 6 (Formulas with GSA)**.** *Consider the following program:*

```
1 z = 0;
2 if ( x > 1 ) goto 6;
3 if ( y > 1 ) goto 7;
```

```
4 z = 1;
5 goto 8;
6 z = 2;
7 goto 8; // first mergepoint
8 ...     // second mergepoint
```

Assuming a starting Π_{DSE}*, we will compute the path predicate in two mergepoints:*

- *In Line 7, two paths are merging:* $\pi_1 = [1, 2, 6, 7]$ *and* $\pi_2 = [1, 2, 3, 7]$*. The path predicates are:* $\Pi_{\pi_1} = \Pi_{DSE} \wedge x > 1$ *and* $\Pi_{\pi_2} = \Pi_{DSE} \wedge x \leq 1 \wedge y > 1$*. The path predicate* Π_7 *at Line 7 can be computed: 1) by disjoining* $\Pi_7 = (\Pi_{DSE} \wedge x > 1) \vee (\Pi_{DSE} \wedge x \leq 1 \wedge y > 1)$*, 2) by removing shared terms first* $\Pi_7 = \Pi_{DSE} \wedge ((x > 1) \vee (x \leq 1 \wedge y > 1))$*, and 3) with gating path expressions* $\Pi_7 = \Pi_{DSE} \wedge \mathsf{ite}(x > 1, true, \mathsf{ite}(y > 1, true, false))$*.*

- *In Line 8, the two paths from Line 7 merge with path* $\pi_3 = [1, 2, 3, 4, 5, 8]$*, with a path predicate* $\Pi_{\pi_3} = \Pi_{DSE} \wedge x \leq 1 \wedge y \leq 1$*. Perform the same operations between* Π_7*, and* Π_{π_3} *we get: a) by disjoining:* $\Pi_8 = (\Pi_{DSE} \wedge x > 1) \vee (\Pi_{DSE} \wedge x \leq 1 \wedge y > 1) \vee (\Pi_{DSE} \wedge x \leq 1 \wedge y \leq 1)$*, 2) by removing shared terms:* $\Pi_7 = \Pi_{DSE} \wedge ((x > 1) \vee (x \leq 1 \wedge y > 1) \vee (x \leq 1 \wedge y \leq 1))$*, 3) with gating path expressions:* $\Pi_8 = \Pi_{DSE} \wedge \mathsf{ite}(x > 1, true, \mathsf{ite}(y > 1, true, true))$*.*

Assuming that $|\Pi_{DSE}|_s^e = c$*, and full term reuse (Section 7.4.3) is applied we have that* $\left|\Pi_8^{disjoint}\right|_s^e = c + 15$*,* $\left|\Pi_8^{shared}\right|_s^e = c + 12$*, and* $\left|\Pi_8^{GSA}\right|_s^e = c + 9$ *(due to lack of negations). Further, we see that the structure of the program with if-then-else statements naturally corresponds to the structure of the generated formula. The structure is also preserved for merged variables; with GSA the value of* z *would be* $\mathsf{ite}(x > 1, 2, \mathsf{ite}(y > 1, 0, 1))$*.*

Note all formulas in the example above are logically equivalent. Running simplifications on top that remove redundancy or extraneous expressions is possible (although fragile, especially

if they are syntax driven). GSA offers a straightforward way to retain the structure of the CFG in the formula and remove redundancy.

Handling Overapproximated CFGs. At any point during SSE, the path predicate is computed as the conjunction of the DSE predicate Π_{DSE} and the SSE predicate computed by substitution: $\Pi_{SSE} = \gamma[\Lambda \to true, \bot \to false]$[3]. MergePoint uses the resulting predicate to perform path pruning (lines 4 and 6 in Algorithm 5) offering two advantages: any infeasible edges introduced by CFG recovery are eliminated, and our formulas only consider feasible paths.

7.3.5 Transition Point Finalization

After the SSE pass is complete, we check which states need to be forked. We first gather transition points and check whether they were reached by SSE (line 16 in Algorithm 3). For the set of distinct—based on their jump target address—transition points, MergePoint will fork a new symbolic state in a `Finalize` step, where a DSE executor is created (ℓ, Π, Γ) using the state (γ, Γ) of each transition point.

Generating Test Cases. Though MergePoint can generate an input for each covered path, that would result in an exponential number of test cases in the size of the CFG_e. By default, we only output one test per CFG node explored by static symbolic execution. (Note that for branch coverage the algorithm can be modified to generate a test case for every edge of the CFG.) The number of test cases can alternatively be minimized by generating test cases only for nodes that have not been covered by previous test cases.

Underapproximated CFGs. Last, before proceeding with DSE, veritesting checks whether we missed any paths due to the underapproximated CFG. To do so, veritesting queries the

[3]Note that for solvers with incremental capabilities, it is not necessary to conjoin the two predicates for every query. Π_{DSE} can be pushed to the solver context once, before the SSE phase begins.

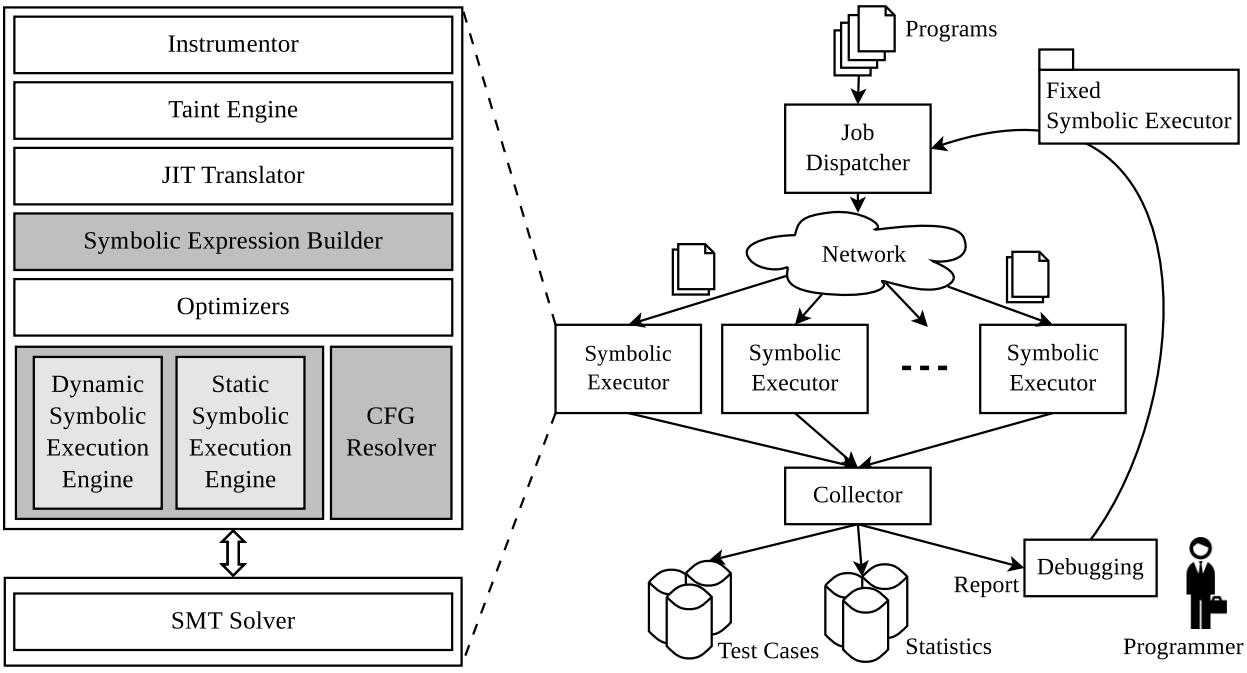


Figure 7.4: MergePoint Architecture.

negation of the path predicate at the `Exit` node (the disjunction of the path predicates of forked states). If the query is satisfiable, an extra state is forked to explore missed paths.

Incremental Deployment. Veritesting is an online algorithm, it runs as the program executes. If any step of the veritesting algorithm fails, the system falls back to DSE until the next symbolic branch. An advantage of this approach is that the implementation can be gradually deployed; supporting all possible programming constructs is not necessary, since veritesting runs on a best-effort basis.

7.4 MergePoint Architecture

The ultimate goal of MergePoint is to perform effective testing on thousands of applications. In this section, we provide a high-level description of the system and key design choices.

7.4.1 Overview

MergePoint follows the design of a concolic executor. The symbolic execution engine runs on top of an instrumentation tool and x86 instructions are JITed to an intermediate representation before being symbolically executed. A taint analysis layer ensures that the symbolic executor is used only when necessary, i.e., only for instructions operating on input-derived data. The layers of the MergePoint executor are shown on the left of Figure 7.4.

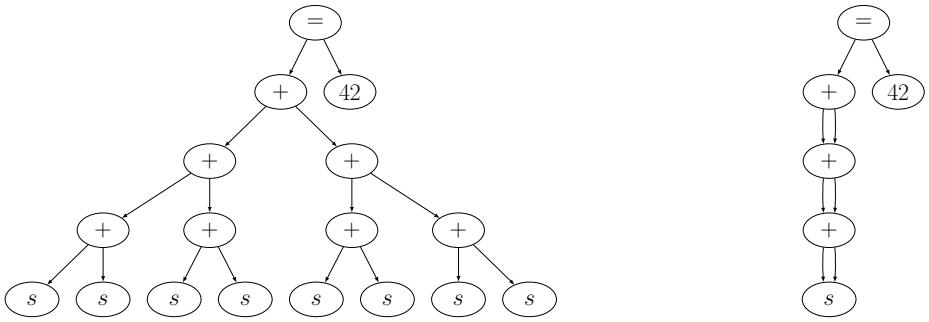
To enable veritesting, the MergePoint executor is enhanced with two main modules (shaded): a static symbolic executor and a CFG recovery module. In the rest of this section, we discuss how the executor fits within the MergePoint distributed infrastructure (§7.4.2), and a key design decision in the handling of symbolic expressions (§7.4.3).

7.4.2 Distributed Infrastructure

As a stand-alone tool, the MergePoint executor takes in a program and a user configuration (including a time limit, inputs, etc.) and outputs test cases, bugs, and statistics. One goal of MergePoint is to test software en masse. However, a single 30-minute experiment on 1,000 programs requires almost 3 weeks of CPU time. To test our techniques, we developed a distributed infrastructure that utilizes multiple nodes to run programs in parallel. Figure 7.4 presents the end-to-end architecture of MergePoint.

MergePoint employs a first-come, first-served central queuing policy. The policy is simple, yet yields high-utilization: a program waiting at the top of the dispatcher queue is sent to the next available symbolic executor instance.

Data generated at every node are aggregated and stored in centralized storage. We use stored data as an immediate feedback mechanism about the performance behavior of the symbolic executor on a large number of programs. The feedback mechanism served as a guide on several design choices, e.g., using a hash-consed language (§7.4.3).




```
1 x = s   (s is symbolic)
2 x = x + x
3 x = x + x
4 x = x + x
5 assert (x == 42)
```

Figure 7.5: Hash consing example. Top-left: naïvely generated formula. Top-right: hash-consed formula.

7.4.3 A Hash-Consed Expression Language

Whenever a program variable is used in an expression, `eval` in Algorithm 3 replaces it with its value in the symbolic store. A naïve substitution algorithm may introduce an exponential blowup, even for a straightline program. For example, the path predicate for Figure 7.5 is $s + s + s + s + s + s + s + s = 42$ (where there are $2^3 = 8$ uses of the s variable).

Hash consing [65] is a technique for avoiding duplication during substitution and reusing previously constructed expressions. Previous work in symbolic execution has made extensive use of hash-consing variants to avoid duplicate expressions. Examples include creating maximally-shared graphs [51], using expression caching [62], or ensuring that structurally equivalent expressions that are passed to the SMT solver are reused [21].

MergePoint goes one step further and builds hash-consing *into* the language. The constructor for every expression type is hash-consed by default, meaning that the implementor of the symbolic executor is incapable of creating duplicate expressions. Every previously computed expression is stored and will be reused. MergePoint also provides iterators over

hash-consed expressions for standard operations (fold, map, map-reduce, etc.), to ensure all traversals are linear in the size of the expression.

Following the approach of [125], MergePoint stores hash-consed expressions in an array of weak references—meaning that the expressions will be deallocated as long as no other object has a live reference to them—that can be efficiently garbage collected.

Two symbolic expressions are structurally equivalent when they have the same type (e.g., *ite*) and their children expressions are identical. Unfortunately, this means that $x + 2$ is not equivalent to $2 + x$. To handle such cases, MergePoint normalizes expressions based on a global ordering of expressions wherever applicable (e.g., for associative operators). After normalization, MergePoint performs the standard algebraic simplification step, to eliminate redundant expressions, e.g., $x \oplus x = 0$. The simplification/normalization steps can be expensive—in the worst case linear in the size of the expression. To amortize the cost of normalization/simplification, MergePoint caches previously performed transformations. Note that caching expressions conflicts with our garbage collection scheme (we would never deallocate unreachable expressions), and thus the cache is periodically flushed.

The normalization and simplification steps are performed *before* the expression is actually hash-consed. Delaying hash-consing ensures that all intermediate expressions generated during transformations are not stored in the expression table.

7.5 Implementation

MergePoint runs in a virtual machine cloud. Our architecture uses a central dispatcher to send individual programs to analysis nodes. The main MergePoint Veritesting implementation is built on top of Mayhem [3], and consists of an additional 17,000 lines of OCaml and 9,000 lines of C/C++. The communication between multiple nodes and the dispatcher is implemented in 3,500 lines of Erlang. MergePoint uses the BAP [41] platform

for translating x86 code to an intermediate representation, CFG recovery and loop unrolling. We use PIN [63] for instrumentation and Z3 [66] for solving SMT queries.

7.6 Evaluation

In this section we evaluate our techniques using multiple benchmarks with respect to three main questions:

1. Does Veritesting find more bugs than previous approaches? We show that MergePoint with veritesting finds twice as many bugs than without.

2. Does Veritesting improve node coverage? We show MergePoint with veritesting improves node coverage over DSE.

3. Does Veritesting improve path coverage? Previous work showed dynamic state merging outperforms vanilla DSE [26]. We show MergePoint with veritesting improves path coverage and outperforms both approaches.

We detail our large-scale experiment on 33,248 programs from Debian Linux. MergePoint generated billions of SMT queries, hundreds of millions of test cases, millions of crashes, and found 11,687 distinct bugs.

Overall, our results show MergePoint with veritesting improves performance on all three metrics. We also show that MergePoint is effective at checking a large number of programs. Before proceeding to the evaluation, we present our setup and benchmarks sets. All experimental data from MergePoint are publicly available online [113].

Experiment Setup. We ran all distributed MergePoint experiments on a private cluster consisting of 100 virtual nodes running Debian Squeeze on a single Intel 2.68 GHz Xeon core with 1GB of RAM. All comparison tests against previous systems were run on a single node

Intel Core i7 CPU and 16 GB of RAM since these systems could not run on our distributed infrastructure.

We created three benchmarks: coreutils, BIN, and Debian. Coreutils and BIN were compiled so that coverage information could be collected via `gcov`. The Debian benchmark consists of binaries used by millions of users worldwide.

Benchmark 1: GNU coreutils (86 programs)[4]. We use the coreutils benchmark to compare to previous work since: 1) the coreutils suite was originally used by KLEE [21] and other researchers [21, 126, 26, 3, 47] to evaluate their systems, and 2) configuration parameters for these programs used by other tools are publicly available [127]. Numbers reported with respect to coreutils do not include library code to remain consistent with compared work. Unless otherwise specified, we ran each program in this suite for 1 hour.

Benchmark 2: The BIN suite (1,023 programs). We obtained all the binaries located under the `/bin`, `/usr/bin`, and `/sbin` directories from a default Debian Squeeze installation[5]. We kept binaries reading from `/dev/stdin`, or from a file specified on the command line. In a final processing step, we filtered out programs that require user interaction (e.g., GUIs). BIN consists of 1,023 binary programs, and comprises 2,181,735 executable lines of source code (as reported by `gcov`). The BIN benchmark *includes* library code packaged with the application in the dataset, making coverage measurements more conservative than coreutils. For example, an application may include an entire library, but only one function is reachable from the application. We nonetheless include all uncovered lines from the library source file in our coverage computation. Unless otherwise specified, we ran each program in this suite for 30 minutes.

[4]All generated test cases were executed natively to compute code coverage results. To avoid destructive side-effects we removed 3 coreutils (`rm`, `rmdir` and `kill`) from the original KLEE suite.

[5]What better source of benchmark programs than the ones you use everyday?

	Veritesting	DSE
coreutils	**2 bugs/2 progs**	0/0
BIN	**148 bugs/69 progs**	76 bugs/49 progs

Table 7.2: Veritesting finds 2× more bugs.

Benchmark 3: The BIN_W suite (558 programs). BIN_W consists of all BIN programs, that are also in the default Debian Wheezy (7.4) installation (since our original experiments [5], Debian updated to a newer version). We used the BIN_W of programs to generate our formula dataset (Section 7.7.2), and to perform exploration completion experiments (Section 7.6.3). Experiments on BIN_W programs were run for 1 hour.

Benchmark 4: Debian (33,248 programs). This benchmark consists of all binaries from Debian Wheezy and Sid. We extracted binaries and shared libraries from every package available from the `main` Debian repository. We downloaded 23,944 binaries from Debian Wheezy, and 27,564 binaries from Debian Sid. After discarding duplicate binaries in the two distributions, we are left with a benchmark comprising 33,248 binaries. This represents an order of magnitude more applications than have been tested by prior symbolic execution research. We analyzed each application for less than 15 minutes per experiment.

7.6.1 Bug Finding

Table 7.2 shows the number of bugs found by MergePoint with and without veritesting. Overall, veritesting finds 2× more bugs than without for BIN. Veritesting finds 63 (83%) of the bugs found without veritesting, as well as 85 additional distinct bugs that traditional DSE could not detect.

Veritesting also found two previously unknown crashes in coreutils, even though these applications have been thoroughly tested with symbolic execution [21, 47, 126, 26, 3]. Further

	Veritesting	DSE	Difference
coreutils	75.27%	63.62%	**+11.65%**
BIN	40.02%	34.71%	**+5.31%**

Table 7.3: Veritesting improves node coverage.

investigation showed that the coreutils crashes originate from a library bug that had been undetected for 9 years. The bug is in the time zone parser of the GNU portability library `Gnulib`, which dynamically deallocates a statically allocated memory buffer. It can be triggered by running `touch -d 'TZ="""'`, or `date -d 'TZ="""'`. Furthermore, `Gnulib` is used by several popular projects, and we have confirmed that the bug affects other programs, e.g. `find`, `patch`, `tar`.

As a point of comparison, we ran Kuznetsov's DSM implementation [26], which missed the bugs. We also compared MergePoint with veritesting to S2E [29], a state-of-the-art binary-only symbolic execution system. S2E also missed the bugs. KLEE [21] argued that coreutils is one of the most well-tested suite of open-source applications. Since then, coreutils has become the de facto standard for evaluating bug-finding systems based on symbolic execution. Given the extensive subsequent testing of coreutils, finding two new crashes is evidence that veritesting extends the reach of symbolic execution.

7.6.2 Node Coverage

We evaluated MergePoint both with and without Veritesting on node coverage. Table 7.3 shows our overall results. Veritesting improves node coverage on average in all cases. MergePoint also achieved 27% more code coverage on average than S2E. Note that any positive increase in coverage is important. In particular, Kuznetsov *et al.* showed both dynamic

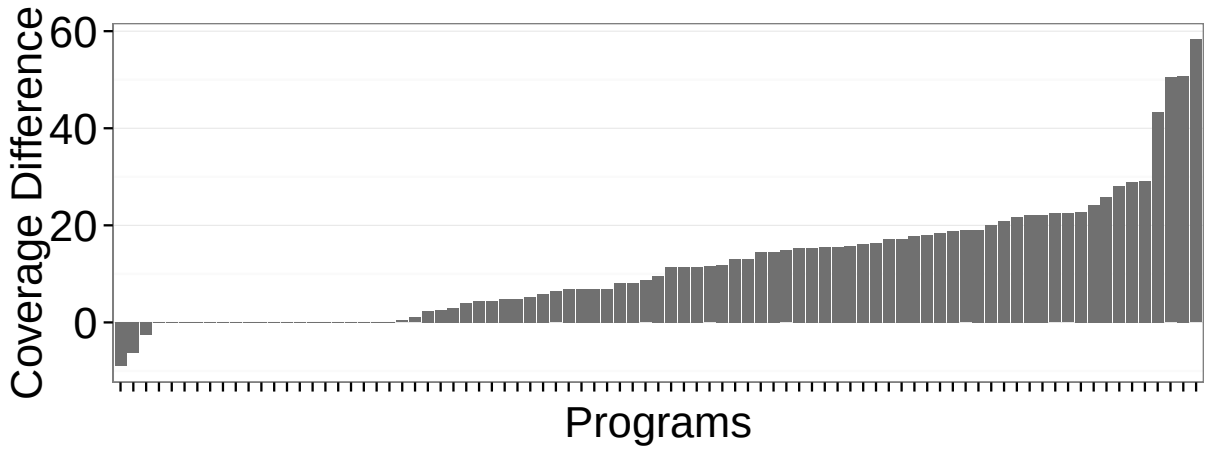


Figure 7.6: Code coverage difference on coreutils before and after veritesting.

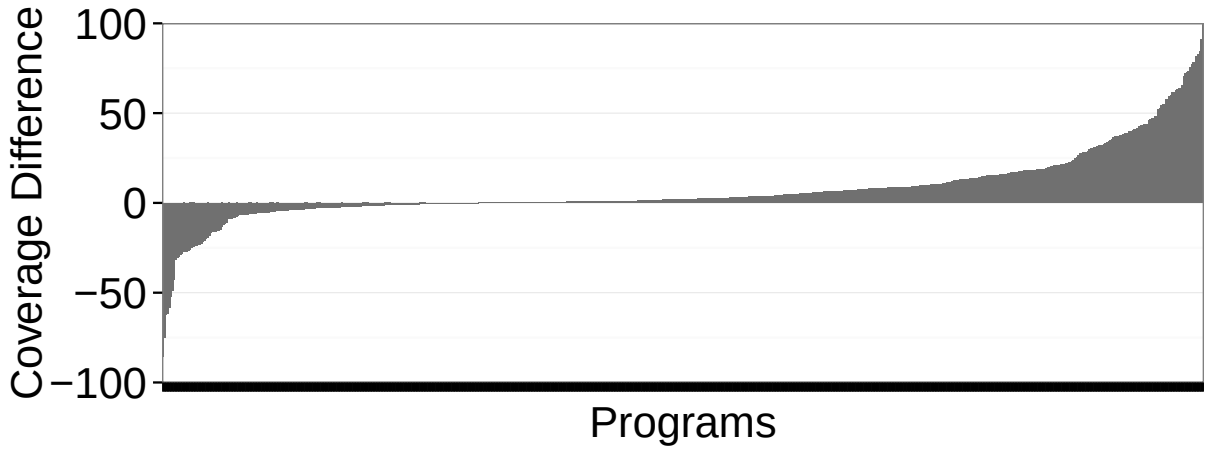


Figure 7.7: Code coverage difference on BIN before and after veritesting, where it made a difference.

state merging and static symbolic execution *reduced* node coverage when compared to vanilla DSE [26, Figure 8].

Figures 7.6 and Figure 7.7 break down the improvement per program. For coreutils, enabling veritesting decreased coverage in only 3 programs (`md5sum`, `printf`, and `pr`). Manual investigation of these programs showed that veritesting generated much harder formulas, and spent more than 90% of its time in the SMT solver, resulting in timeouts. In Figure 7.7 for

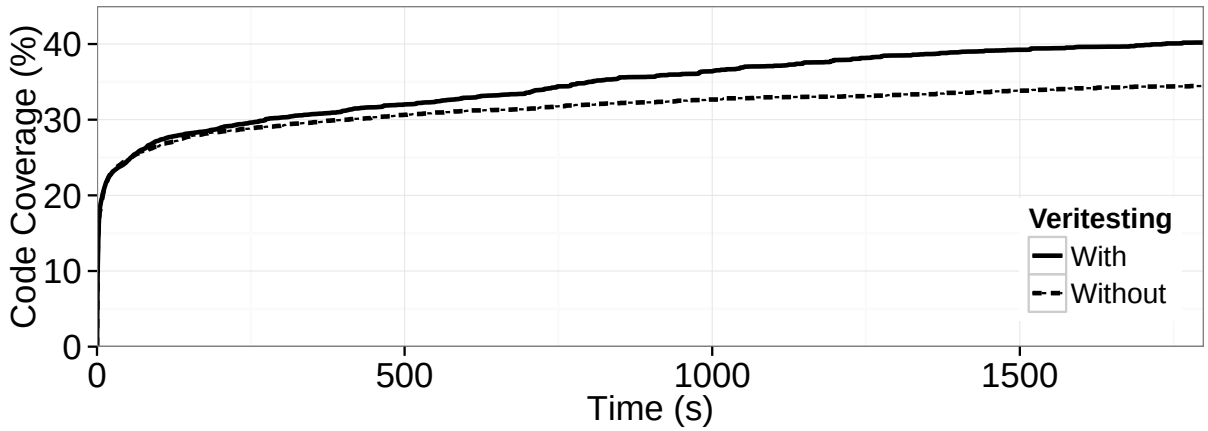


Figure 7.8: Coverage over time (BIN suite).

BIN, we omit programs where node coverage was the same for readability. Overall, the BIN performance improved for 446 programs and decreased for 206.

Figure 7.8 shows the average coverage over time achieved by MergePoint with and without veritesting for the BIN suite. After 30 minutes, MergePoint without veritesting reached 34.45% code coverage. Veritesting achieved the same coverage in less than half the original time (12min 48s). Veritesting's coverage improvement becomes more substantial as analysis time goes on. Veritesting achieved higher coverage velocity, i.e., the rate at which new coverage is obtained, than standard symbolic execution. Over a longer period of time, the difference in velocity means that the coverage difference between the two techniques is likely to increase further, showing that the longer MergePoint runs, the more essential veritesting becomes for high code coverage.

The above tests demonstrates the improvements of veritesting for MergePoint. We also ran both S2E and MergePoint (with veritesting) on coreutils using the same configuration for one hour on each utility in coreutils, excluding 11 programs where S2E emits assertion errors. Figure 7.9 compares the increase in coverage obtained by MergePoint with veritesting over S2E. MergePoint achieved 27% more code coverage on average than S2E. We investigated programs where S2E outperforms MergePoint. For instance, on `pinky`—

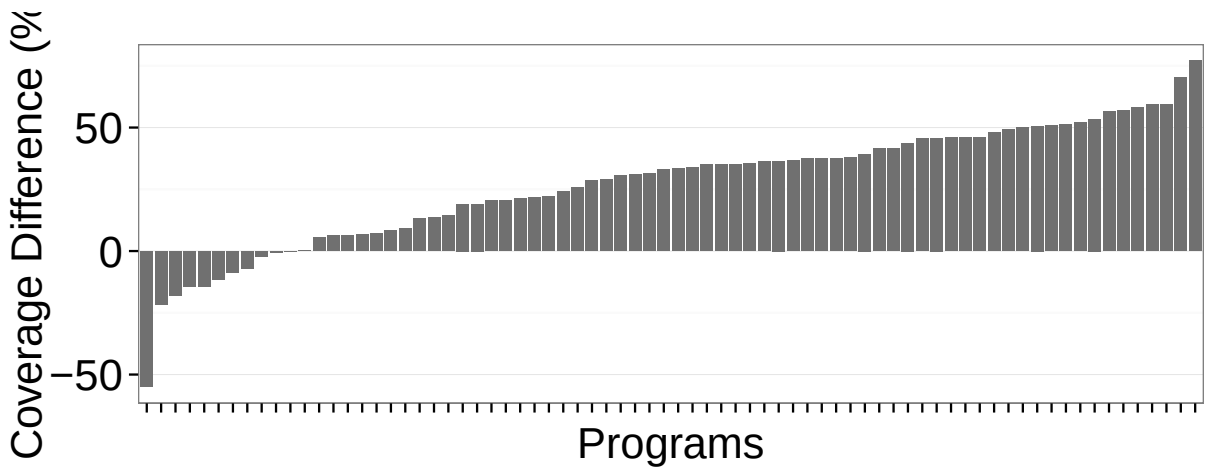


Figure 7.9: Code coverage difference on coreutils obtained by MergePoint vs. S2E

the main outlier in the distribution—S2E achieves 50% more coverage. The main reason for this difference is that `pinky` uses a system call not handled by the current MergePoint implementation (netlink socket).

7.6.3 Path Coverage

We evaluated the path coverage of MergePoint both with and without veritesting using three different metrics: time to complete exploration, multiplicity, and fork rate.

Time to complete exploration. The metric reports the amount of time required to completely explore a program, in those cases where exploration finished.

The number of paths checked by an exhaustive DSE run is also the total number of paths possible. In such cases we can measure a) whether veritesting also completed, and b) if so, how long it took relative to DSE. MergePoint without veritesting was able to exhaust all paths for 46 programs. MergePoint with veritesting completes all paths 73% faster than without veritesting. Unlike other metrics, the time to complete exploration is not biased (for example, if we were measuring the number of explored paths, one could always select the shortest paths with the easier-to-solve formulas, thus gaming the metric), since it ensures that

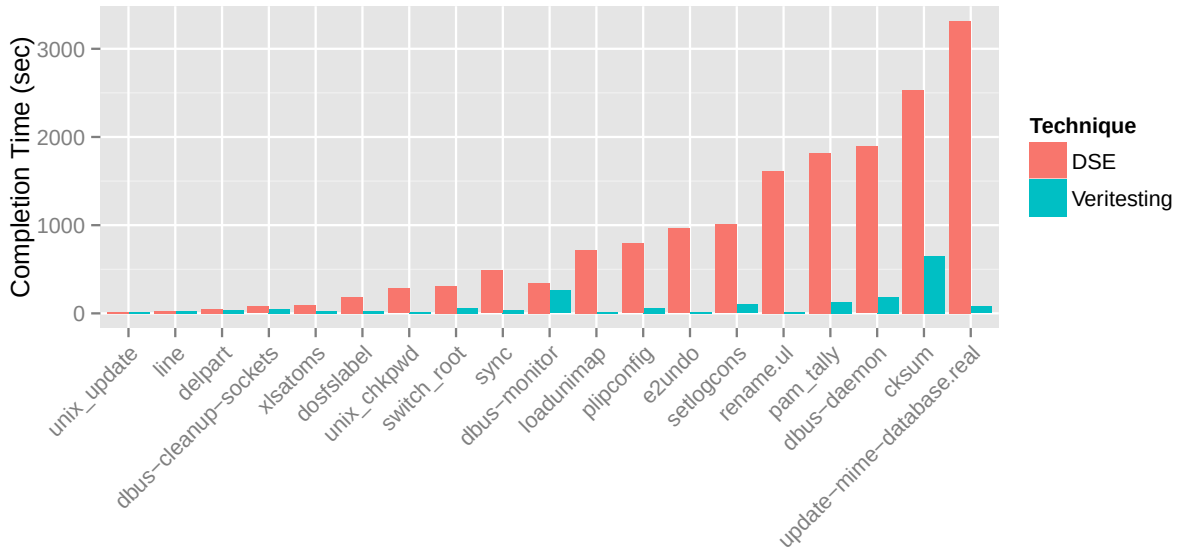


Figure 7.10: Time to complete exploration with DSE and Veritesting.

both techniques accomplished the same task, i.e., explored all program paths. The speedup shows that veritesting is faster when reaching the same goal.

Since our original experiments on BIN, we performed additional one hour experiments on path completion with BIN_W. Again, we gathered all programs that terminate with DSE and checked their times with veritesting. Figure 7.10 summarizes the results—the diagram includes only the programs that required more than 10 seconds to terminate with DSE. We note that veritesting is always superior, and gives an average speedup of 17.8×.

Multiplicity. Multiplicity was proposed by Kuznetsov *et al.* [26] as a metric correlated with path coverage. The initial multiplicity of a state is 1. When a state forks, both children inherit the state multiplicity. When combining two states, the multiplicity of the resulting state is the sum of their multiplicities. A higher multiplicity indicates higher path coverage.

We also evaluated the multiplicity for veritesting. Figure 7.11 shows the state multiplicity probability distribution function for BIN. The average multiplicity over all programs was 1.4×10^{290} and the median was 1.8×10^{12} (recall, higher is better). The distribution resembles

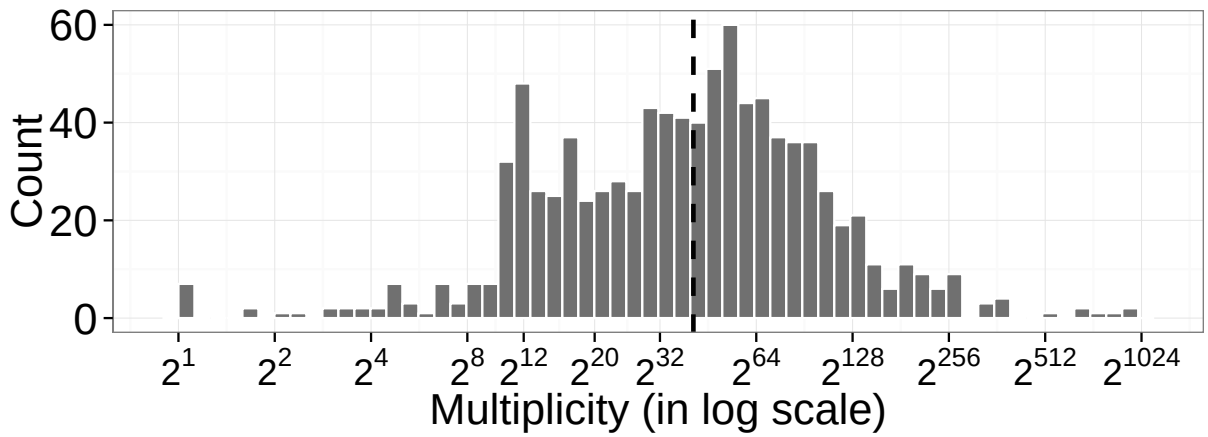


Figure 7.11: Multiplicity distribution (BIN suite).

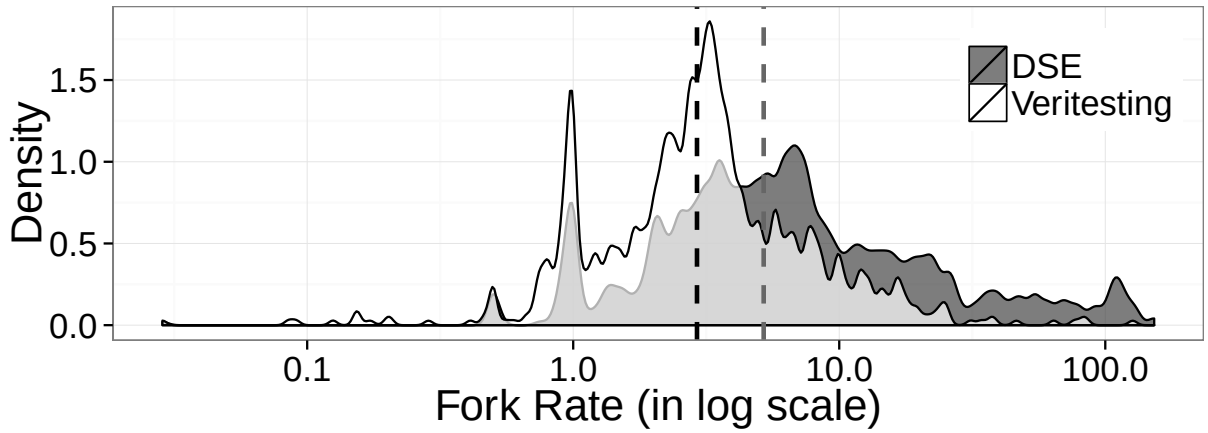


Figure 7.12: Fork rate distribution before and after veritesting with their respective medians (the vertical lines) for BIN.

a lognormal with an abnormal spike for programs with multiplicity of 4,096 (2^{12}). Further investigation showed that 72% of those programs came from the `netpbm` family of programs. Veritesting was unable to achieve very high multiplicity, due to the presence of unresolved calls in the recovered CFG. Improving the CFG recovery phase should further improve performance. Note that even with a multiplicity of 4,096, veritesting still improves coverage by 13.46% on the `netpbm` utilities. The multiplicity average and median for coreutils were

1.4×10^{199} and 4.4×10^{11}, respectively. Multiplicity had high variance; thus the median is likely a better performance estimator.

Fork rate. Another metric is the *fork rate* of an executor, which gives an indication of the size of the outstanding state space. If we represent the state space as a tree, where each node is a path, and its children are the paths that it forks, then the fork rate is the fanout factor of each node. Lower fork rate is better because it indicates a potentially exponentially-smaller state space. For instance, a tree of height n with a fanout factor of 5 has approximately 5^n nodes, while a tree with a fanout factor of 10 will have roughly 10^n nodes. Thus, a tree with a fanout factor of 5 is 2^n times smaller than a tree with a fanout factor of 10.

Figure 7.12 shows the fork rate distribution with and without veritesting of BIN. The graph shows that veritesting decreases the average fork rate by 65% (the median by 44%) from 13 to 4.6 (lower is better). In other words, without veritesting we used to have 13 new paths (forks) to explore for every analyzed path; with veritesting we have only 4.6. Thus, for the BIN programs, veritesting reduces the state space by a factor of $\left(\frac{13}{4.6}\right)^n \approx 3^n$, where n is the depth of the state space. This is an exponential reduction of the space, allowing symbolic execution to consider exponentially more paths during each analysis.

7.6.4 Checking Debian

In this section, we evaluate veritesting's bug finding ability on every program available in Debian Wheezy and Sid. We show that veritesting enables large-scale bug finding.

Since we test 33,248 binaries, any type of per-program manual labor is impractical. We used a single input specification for our experiments: `-sym-arg 1 10 -sym-arg 2 2 -sym-arg 3 2 -sym-anon-file 24 -sym-stdin 24` (3 symbolic arguments up to 10, 2, and 2 bytes respectively, and symbolic files/stdin up to 24 bytes). MergePoint encountered at least one symbolic branch in 23,731 binaries. We analyzed Wheezy binaries once, and Sid

Total programs	33,248
Total SMT queries	15,914,407,892
Queries hitting cache	12,307,311,404
Symbolic instrs	71,025,540,812
Run time	235,623,757s
Symb exec time	125,412,247s
SAT time	40,411,781s
Model gen time	30,665,881s
# test cases	199,685,594
# crashes	2,365,154
# unique bugs	11,687
# fixed bugs	202
Confirmed control flow hijack	152

Table 7.4: Overall numbers for checking Debian.

binaries twice (one experiment with a 24-byte symbolic file, the other with 2100 bytes to find buffer overflows). Including data processing, the experiments took 18 CPU-months.

Our overall results are shown in Table 7.4. Veritesting found 11,687 distinct bugs (by stack hash) that crash programs. The bugs appear in 4,379 of the 33,248 programs. Veritesting also finds bugs that are potential security threats. 224 crashes have a corrupt stack, i.e. a saved instruction pointer has been overwritten by user input. Those crashes are most likely exploitable, and we have already confirmed exploitability of 152 programs. As an interesting data point, it would have cost $0.28 per unique crash had we run our experiments on the

Amazon Elastic Compute Cloud, assuming that our cluster nodes are equivalent to large instances.

The volume of bugs makes it difficult to report all bugs in a usable manner. Note that each bug report includes a crashing test case, thus reproducing the bug is easy. Instead, practical problems such as identifying the correct developer and ensuring responsible disclosure of potential vulnerabilities dominate our time. As of this writing, we have reported 1,043 crashes in total [128]. Not a single report was marked as unreproducible on the Debian bug tracking system. 202 bugs have already been fixed in the Debian repositories, demonstrating the real-world impact of our work. Additionally, the patches gave an opportunity to the package maintainers to harden at least 29 programs, enabling modern defenses like stack canaries and DEP.

7.7 Limits & Trade-offs

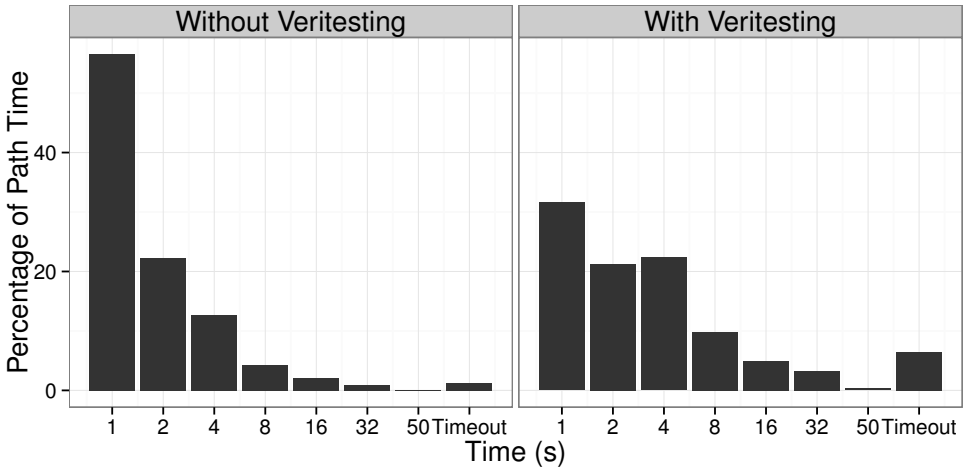
Our experiments so far show that veritesting can effectively achieve path completion faster, reduce the fork rate, achieve higher code coverage, and find more bugs. In this section, we discuss when the technique works well and when it does not, discuss limitations, as well as possibilities for future work.

7.7.1 Execution Profile

Each run takes longer with veritesting because multi-path SMT formulas tend to be harder. The coverage improvement demonstrates that the additional SMT cost is amortized over the increased number of paths represented in each run. At its core, veritesting is pushing the SMT engine harder instead of brute-forcing paths by forking new DSE executors. This result confirms that the benefits of veritesting outweigh its cost. The distribution of path times (Figure 7.13b) shows that the vast majority (56%) of paths explored take less than 1

Component	DSE	Veritesting
Instrumentation	40.01%	16.95%
SMT Solver	19.23%	63.16%
Symbolic Execution	39.76%	19.89%

(a)



(b)

Figure 7.13: MergePoint performance before and after veritesting for BIN. The above figures show: (a) Performance breakdown for each component; (b) Analysis time distribution.

second for standard symbolic execution. With veritesting, the fast paths are fewer (21%), and we get more timeouts (6.4% vs. 1.2%). The same differences are also reflected in the component breakdown. With veritesting, most of the time (63%) is spent in the solver, while with standard DSE most of the time (60%) is spent re-executing similar paths that could be merged and explored in a single execution.

Best vs Worst. On our path completion experiments, we saw that veritesting consistently outperformed DSE (Figure 7.10). However, on code coverage experiments the performance distribution was not as one-sided (Figures 7.6 and 7.7), and on some programs veritesting performed worse (we emphasize that on average over our BIN dataset, results indicate the

trade-off is beneficial). In order to better understand cases where veritesting is worse than per-path DSE, we performed a targeted experiment to identify the characteristics that make a program amenable to veritesting. To do so, we gathered the programs where veritesting impacts the coverage by more than 5%, positively or negatively, and generated a performance breakdown similar to Figure 7.13a:

Improvement	SMT	Timeout	Coverage
Positive	30%	1%	82.14%
Negative	73%	6%	34.31%

We observe again that veritesting performs best when the solver is not dominating the cost of symbolic execution or causing timeouts (recall from Section 7.6.2). We performed a similar breakdown for the bugs we found in our BIN suite (Table 7.2) and observed the same behavior. Out of 148 bugs, veritesting found 79 bugs that were not found by DSE in programs where the average amount of time spent in the SMT solver was 44%. In contrast, DSE found 69 bugs, out of which 26 went undetected by veritesting, with the average amount of time spent in the solver being 67%.

Hash Poisoning. To identify the root cause of this behavior, we performed manual analysis on the symbolic execution runs where DSE performed significantly better than veritesting—including `md5sum`, `pr`, `printf`, and many more programs from our dataset (Section 7.6.2). In all cases where DSE was better, we observed the same pattern: DSE was exploring one path at a time, infrequently timing out on a single path that involved a hash or other hard to reason operation (e.g., a round of `MD5`), while veritesting was exploring multiple paths simultaneously—including the one with the hash operation—resulting mostly in timeouts.

We refer to this problem as *hash poisoning*, since a single path can "poison" neighboring merged paths.

Hash poisoning highlights the trade-off of our approach: veritesting can analyze multiple paths simultaneously, but including a single path that is hard to reason with existing solvers is enough to cause neighboring paths not to be analyzed (due to timeouts). With DSE this constraint does not apply, since every path is explored individually. Interestingly, when we reduced the size of the input to make DSE fast enough to terminate, veritesting was *faster* than DSE in terms of path completion, despite the harder formulas. Thus, in terms of path completion, veritesting is preferable even when formulas become hard—it is better to solve a hard formula included in multiple paths once, instead of re-solving it every time. Utilizing techniques similar to QCE [26], to decide when to merge two paths, based on analysis parameters (e.g., timeouts) and program traits (e.g., hash operations) is an interesting possibility for future work.

Hash-consing Effect. We also measured the effect of our hash-consed based language on veritesting. We performed 4 experiments on our BIN suite (5 min/prog) and measured performance across two axes: veritesting vs. DSE and hash-consing vs. no hash-consing. The table below summarizes our results in terms of average coverage and generated test cases:

Technique	No hash-consing	Hash-consing	Difference
Veritesting	24.24%	29.64%	+**5.40%**
DSE	26.82%	28.98%	+**2.16%**

We see that hash-consing affects performance dramatically: disabling it would make veritesting worse than DSE (within the 5 minute interval). In fact, our initial veritesting

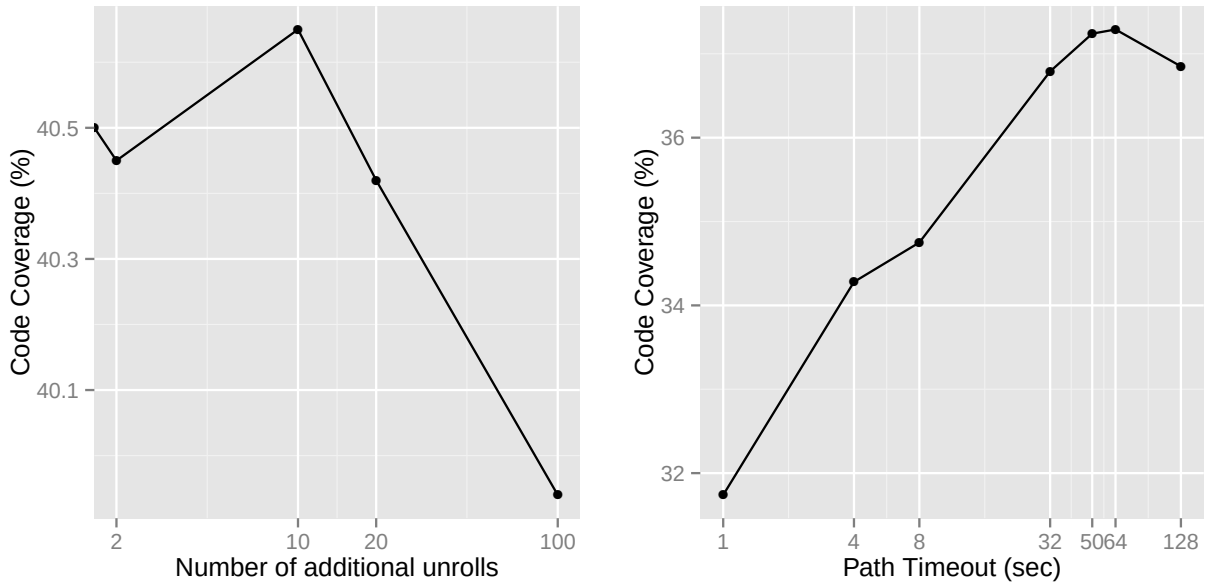
implementation did not include hash-consing, and did not improve coverage. The cost of duplicating or naïvely recursing over expressions is prohibitive for expressions encompassing multiple paths (note that DSE is not as strongly affected).

7.7.2 Discussion

Unrolling Loops, Functions, and Symbolic Execution Parameters. Veritesting as presented in Section 7.3 operates on intra-procedural control flow graphs, and loop unrolling was based on the concrete number of loop iterations. Since our original prototype implementation, we extended our system to both handle functions (by resolving calls and generating inter-procedural control flow graphs), and unroll loops further than the concrete number of iterations. Recursion, and non-terminating loops require both features to have an upper bound, i.e., a threshold over which function resolution or loop unrolling should stop and fork a different executor. Note that such bounds are also necessary for performance; an unrolled loop in the CFG may never be executed and the time spent unrolling could be used elsewhere.

Figure 7.14a shows the trade-off for loop unrolling in terms of code coverage for our BIN suite. We observe that the loop unrolling parameter directly affects performance. How many times should we unroll loops to get the best performance? Could we determine this parameter choice before we run the experiment? Will these values be best for all programs? These and other parameter-related questions (e.g., Figure 7.14b shows how the solver timeout parameter affects performance for veritesting) are part of the bigger problem of *tuning* symbolic execution parameters (Chapter 8), and are outside the scope of this thesis. Nevertheless, we believe exploring such questions further, is an interesting direction for the future.

The Q4 Dataset & Opportunities. Veritesting explores the middle-ground between DSE and SSE by trading-off larger (and potentially slower to solve) formulas for more paths.



(a) (b)

Figure 7.14: MergePoint performance before and after veritesting for BIN. The above figures show: (a) Performance breakdown for each component; (b) Analysis time distribution.

The time spent in formula solving is paramount to determine how to best take advantage of this trade-off, and ultimately how to best make use of the SMT solver. We present some preliminary observations from the Q4 dataset, a set of more than 100 million QF_BV formulas collected from 4 one-hour experiments on BIN_W. In the 4 experiments, we varied two parameters: input size (32 bytes vs 64 bytes for each symbolic file), and technique (DSE and veritesting). For every allocated solver, the dataset includes the full query log in SMTLIB format (using `push`, `pop`, `check-sat` etc. directives). The queries represent exactly what was sent to the SMT solver during each experiment; cached queries, simplifications, and other symbolic execution components are not part of the query log. We note however, that the number of queries solved is actually throttled due to the formula printer; in a one-hour experiment without logging, up to 4× more queries can be resolved.

As expected, veritesting increases the average solving time from 1.4ms per query (with DSE) to 8.6ms (with veritesting). Veritesting also increases the average formula size (in terms

of QF_BV nodes) from 662K to 837K. Nevertheless, our experiments (Section 7.6) show the solver cost is amortized among a larger number of paths (and can lead to 17.8× faster path completion).

The formula graphs in Figures 3.6 and 3.7 presented in Chapter 3 did not show a clear connection between formula size (or conflicts) and solving time. We now re-evaluate this statement by controlling for symbolic execution technique (DSE vs veritesting) and program. Figures 7.15 and 7.16 show how solving time is influenced by formula size and conflicts for a sample set of programs. We observe that both size and (especially) conflicts appear to be correlated with solving time. Nevertheless, there are still a few exceptions, e.g., `whoami` or `md5sum`, where the data points do not closely follow the LOESS curve. A closer inspection showed that while the this is true for queries from *all* symbolic execution runs of a single program, results may be different when we condition by symbolic execution run. Figure 7.17 shows the connection between solving time and conflicts for a single symbolic execution run (with veritesting). Even for programs where the correlation was not clear before, e.g., `whatis`, we now observe a "continuity" between points, possibly implying that hardness is localized in the program. Note, that such a result would be consistent with our prior observations, e.g., hash poisoning, where we witnessed that a specific part of the program (e.g., one performing a hash operation) is responsible for generating hard formulas.

Can we predict formula hardness? Can we use estimators such as size or conflicts for prediction? Is hardness localized (e.g., per path)? Do these patterns persist across solver implementations? Given an oracle that predicts solving time, how could we make use of it in veritesting? These are all excellent questions to address in future work.

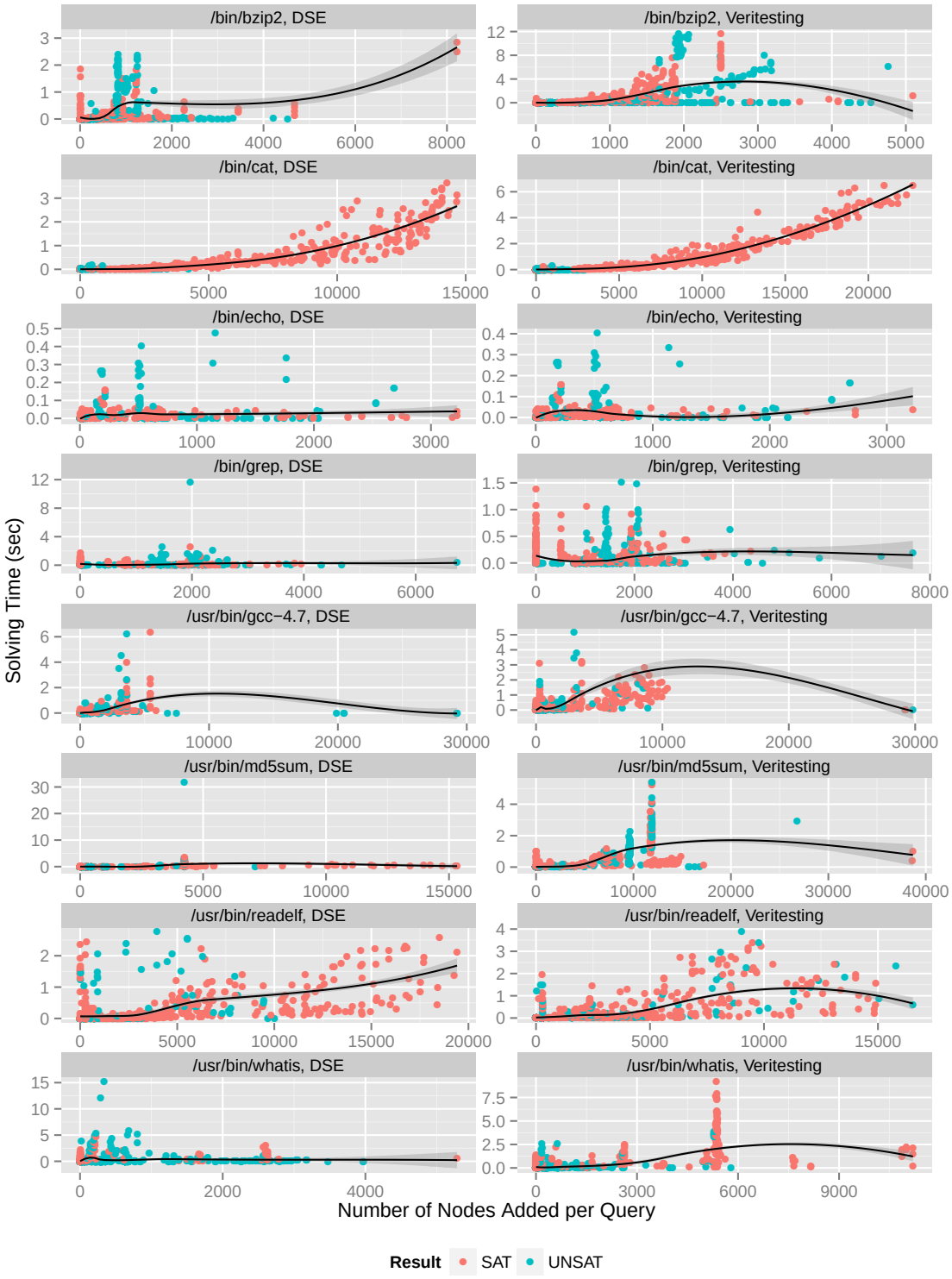


Figure 7.15: Solving time with the number of nodes added per program.

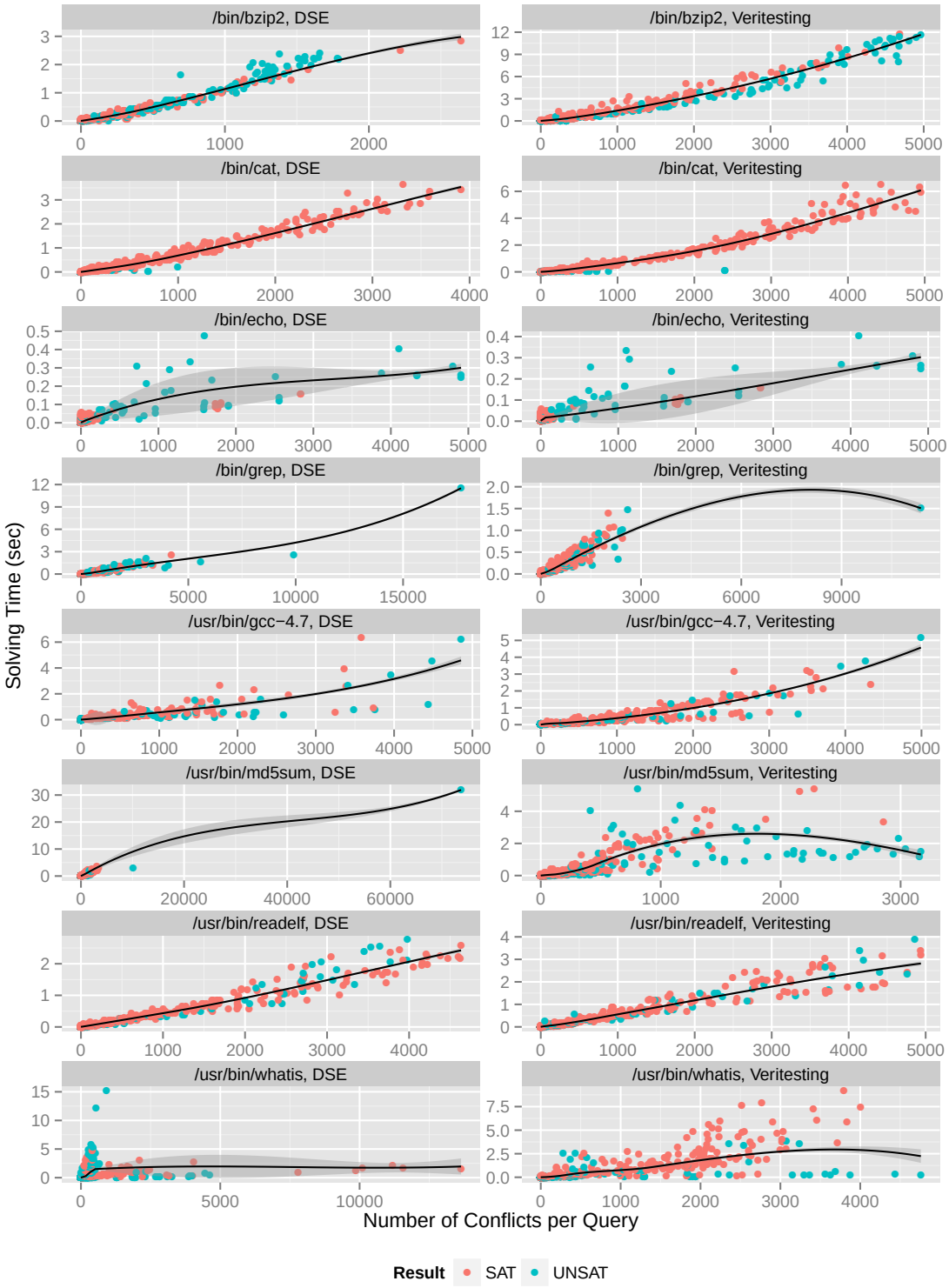


Figure 7.16: Solving time with the number of conflicts per program.

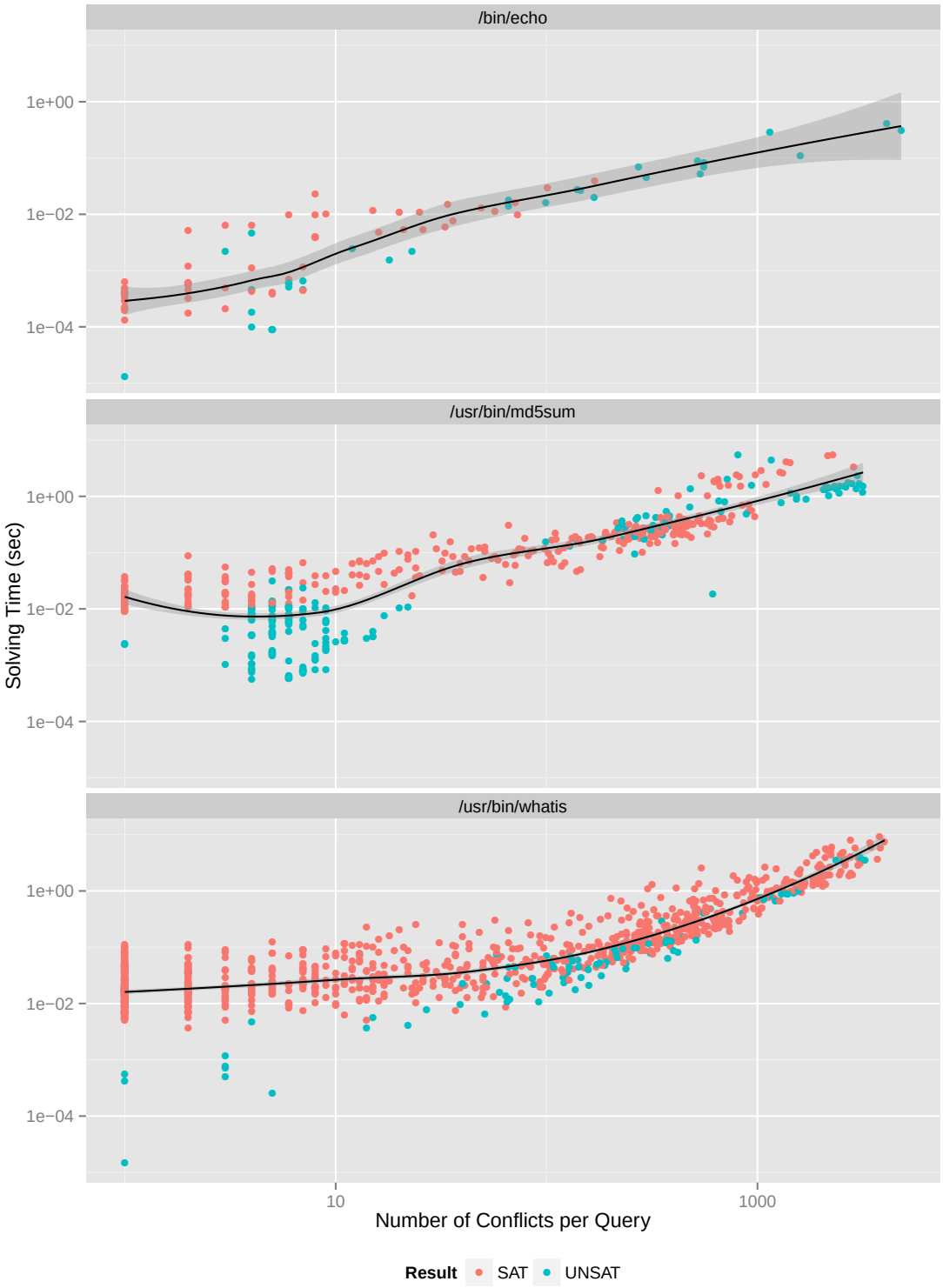


Figure 7.17: Solving time with the number of conflicts per symbolic execution run.

7.8 Related Work

Symbolic Execution. Symbolic execution has been a success in software testing for almost a decade. Microsoft's SAGE [23] is responsible for finding one third of all bugs discovered by file fuzzing during the development of Windows 7 [23]. KLEE [21] was the first tool to show that symbolic execution can generate test cases that achieve high coverage on real programs by demonstrating it on the UNIX utilities. Since then, a wealth of research projects has investigated extensions and improvements to symbolic execution [29], as well as targeted other application domains [129, 22].

Today, there is a multitude of source-based (e.g., EXE [19], KLEE [21], Cloud9 [47], CUTE [30], PEX [56], etc), and binary-only symbolic execution frameworks (e.g., Bouncer [22], BitFuzz [24], FuzzBall [112], McVeto [31], SAGE [12], S2E [29], MAYHEM [3], etc), many of which are routinely used—both in academia and industry—to solve problems in a variety of application domains. Because it is impossible to do justice to all this work we refer the reader to symbolic execution survey papers [60, 61, 25]. In the rest of the section, we mainly focus on work related to state space management and veritesting.

Merging Paths in Verification. Merging execution paths is not new for verification. Koelbl and Pixley [118] pioneered state space merging in static symbolic execution. Concurrently and independently, Xie *et al.* [116] developed Saturn, a verification tool capable of encoding of multiple paths before converting the problem to SAT. These two groups developed the first static symbolic execution execution algorithm with state merging for verification. Babic [51] extended their work further and improved their static algorithm to produce smaller and faster to solve formulas by leveraging Gated Single Assignment (GSA) [122] and maximally-shared graphs (similar to hash-consing [65]). Hansen *et al.* [27] follow an approach similar to Koelbl and Pixley, providing a verification algorithm for x86 executables that merges states at CFG confluence points.

The static portion of our veritesting algorithm is built on top of their ideas. In our approach, we leverage the static algorithms to amplify the effect of dynamic symbolic execution and take advantage of the strengths of both techniques.

Converting Control Dependencies to Data Dependencies. All the above static algorithms are based on *if-conversion* [130], a technique for converting code with branches into predicated straight-line statements. Depending on the context, the technique is also known as ϕ-folding [101], a compiler optimization technique that collapses simple diamond-shaped structure in the control flow graph (CFG). -Overify [131] also performs transformations similar to if-conversion to allow for faster verification times. Collingbourne *et al.* [132] used ϕ-folding to verify semantic equivalence of SIMD instructions. Conceptually, veritesting is an application of if-conversion/extended ϕ-folding, allowing the simultaneous execution of paths across arbitrary code constructs—if statements, loops, even on unstructured CFGs.

Eliminating Redundant States. Another class of state space management techniques detects and drops redundant states. Boonstoppel *et al.* proposed RWSet [59], a state pruning technique identifying redundant states based on similarity of their live variables. If live variables of a state are equivalent to a previously explored path, RWSet will stop exploring the states, as it will not offer additional coverage. Veritesting does not prune paths, but directly avoids redundancy by merging related states.

Dynamic State Merging. Kuznetsov *et al.* [26] proposed a dynamic state merging algorithm compatible with search heuristics. Dynamic state merging mitigates state explosion by merging execution states concurrently with the program exploration. To alleviate the increased cost of queries, they selectively merge when a query cost estimator indicates that merging will be beneficial. Our approach borrows techniques from verification to perform state merging, and mitigates additional query costs by focusing on veritesting.

Compositional Testing & Loops. Godefroid *et al.* [133] introduced function summaries to test code compositionally. The main idea is to record the output of an analyzed function, and to reuse it if it is called again with similar arguments—an idea popularly used in testing [134]. The work was later expended to generate such summaries on demand [135]. Most symbolic execution work unfolds one execution tree at a time. Recent work has looked at generalizing over multiple paths—and specifically to loops—either by identifying specific patterns from a known grammar [136], or by automatically simplifying loop formulas [79].

The connection between compositional testing and veritesting is arguable and largely based on semantics. One interpretation, is that compositional analysis and generalizations over loops are entirely orthogonal and complementary to veritesting. For example, we can generate summaries for a number of functions (using any technique) and still use veritesting to analyze the full program. A different interpretation however, could be that veritesting is a form of compositional testing. Our SSE pass during DSE is generating context-sensitive on-demand summaries of arbitrary code fragments. Investigating the connection between the two techniques and finding points of convergence and divergence is an interesting idea for future work.

Verification. For brevity, we mainly focused on verification techniques based on static symbolic execution. We left out a wealth of other work in verification condition (VC) generation algorithms, e.g., weakest preconditions [42], with several optimizations in terms of formula size and solving times [34] and unstructured programs [75]. The idea of veritesting is compatible with these techniques, however, a forward verification algorithm, seems to be better suited for integration with dynamic symbolic execution. Splitting verification conditions has also been proposed [137] for improving performance.

Abstract Interpretation. Abstract interpretation [138, 139], formalizes the idea that the verification of a program can be done at some level of abstraction where irrelevant details about the semantics and the specification are ignored. The abstract interpretation

framework has been shown to be powerful enough to be applied in diverse areas such as static analyses [138, 139] (e.g., dataflow [140]), model checking [141], predicate abstraction [142], or even more fundamentally program transformations [143]. Symbolic execution—both static and dynamic—is a very expensive form of (potentially non-terminating) abstract interpretation, where the abstract domain consists of formulas over program states. Our veritesting approach is another application of abstract interpretation, that combines the SSE dataflow framework (with meet and transfer functions) for merging states across multiple paths, with the ability to perform abstract interpretation on a single path at a time (DSE).

Model Checking. Model checking [144] is an alternative approach to software testing and verification, also very successful with multiple well-known tools such as BLAST [145], SLAM [146]. Recent bounded model checkers, such as LLBMC [147], highlight the many similarities of bounded model checking and symbolic execution (conversion to an IR, followed by adding assertions, and finally checking validity). Despite the seemingly many differences between model checking and abstract interpretation in general, the two techniques have been shown to be a subset of the other [148, 149, 150]. These parallels were also pointed out more recently by Beyer *et al.*, where they emphasize that the theoretical relationships between the two techniques have currently little impact on the practice of verification; program analyzers target the efficient computation of simple facts about large programs, while model checkers focus on the removal of false alarms through refined analyses of small programs [119].

Nevertheless, research in model checking has served as a precursor to findings in testing and verification. For instance, Clarke *et al.* [28] were the first to use a form of if-conversion before translating the problem to SAT (before Koelbl *et al.* [118], or Xie *et al.* [116] introduced it in symbolic execution). Similarly, the idea of using BDDs for compact state representation [151], or using iterative abstraction refinement [36] appeared first in model checking, and was then followed by similar techniques in symbolic execution [51].

State explosion—albeit in different form—in model checking is challenging, with a multitude of publications in the area [152]. Recent work explored the trade-off between formula size and the usage of the solver [35], showing that adjusting the formula size can lead to significant performance improvements. This work—similar to CEGAR [36]—shows again that treating the underlying solver as a black box (e.g., by sending a single large formula capturing the entire program) is naïve and potentially unscalable. Veritesting is inspired by the same intuition; the goal is to allow the user to adjust the amount of work sent to the solver, and thus explore the trade-off between state explosion and formula size.

Shortly after releasing our bug reports, Romano [153] reported bugs on at least 30,000 distinct binaries. He reports 5 minutes of symbolic execution per binary. Romano's result is impressive. Unfortunately, Romano has not provided a report detailing his methods, benchmarks, or methods, leaving comparison to his work impossible.

7.9 Conclusion

In this chapter we proposed MergePoint and veritesting, a new technique to enhance symbolic execution with verification-based algorithms. We evaluated MergePoint on 1,023 programs and showed that veritesting increases the number of bugs found, node coverage, and path coverage. We showed that veritesting enables large-scale bug finding by testing 33,248 Debian binaries, and finding 11,687 bugs. Our results have had real world impact with 202 bug fixes already present in the latest version of Debian.

Veritesting represents a new design point in the space of symbolic executors, allowing testing techniques to leverage static verification techniques and advancements in SMT technology. While we do not expect, that veritesting will resolve the verification versus testing conundrum any time soon [154], we believe that it may represent a middle ground for the two approaches to eventually converge. We end this chapter, with a quotation from one of

the papers that introduced symbolic execution, back in 1976, as a middle ground between concrete testing and verification (King [16]):

> Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding etc. This paper describes a practical approach between these two extremes.

Acknowledgments

We would like to thank Samantha Gottlieb, Tiffany Bao, and our anonymous reviewers for their comments and suggestions. We also thank Mitch Franzos and PDL for the support they provided during our experiments. This research is supported in part by grants from DARPA and the NSF, as well as the Prabhu and Poonam Goel Fellowship.

Part IV

Conclusion

Chapter 8

Conclusion & Future Work

Always tell the truth.

— My love, Elisavet, *Our first date.*

This dissertation explored the thesis that using symbolic execution for modeling, finding, and demonstrating security-critical bugs such as control flow hijacks is possible, and that exploiting state space trade-offs via pruning, reduction, or segmentation can improve symbolic execution as a testing and bug-finding tool.

We presented two symbolic execution systems capable of demonstrating control flow hijacks on real-world programs both at the source (AEG), and binary (MAYHEM) level. By exploiting specific trade-offs in symbolic execution, such as state pruning (Chapter 5) and reduction (Chapter 6) we were able to increase the efficacy of those tools in identifying exploitable bugs. While we demonstrated our approach on real programs, scaling to larger programs remains an open challenge. The AEG challenge is far from a solved problem, and the last section in Chapter 4 presents a series of ideas for future directions in AEG.

The last part of the thesis introduced veritesting, a symbolic execution technique for exploiting the trade-off between formula expressivity and number of forked states. Our experiments on a large number of programs, show that veritesting finds more bugs, obtains

higher node and path coverage, and can cover a fixed number of paths faster when compared to vanilla symbolic execution. We showed that veritesting enables large-scale bug finding by testing 33,248 Debian binaries, and finding 11,687 bugs. Our results have had real world impact with 202 bug fixes already present in the latest version of Debian.

Symbolic Execution & Future Work There are several ways to view symbolic execution:

- As a **transformation**. Binary programs are in a format that allows direct interpretation by a CPU. Through symbolic execution, programs are converted—potentially one fragment at a time—to logical formulas and interpreted by specialized solvers. In that respect, symbolic execution is just a converter from programs to formulas.

- As an **optimization** for static analyses. Being a fully path- and context-sensitive analysis, symbolic execution can precisely explore only realizable program paths. Code that is not relevant to user input is sliced away. Similar to JIT compilers, symbolic execution has access to the execution state and can perform optimizations that would otherwise be unsound.

- As a **testing** technique. Originally proposed as a program testing technique [16, 17, 18], symbolic execution offers an systematic approach for exploring program paths. During the exploration of each path, specific properties (e.g., no null pointer dereferences) can be tested.

- As a **verification** technique. By systematically exploring paths, symbolic execution can allow us to check whether a specific property holds for all program paths, i.e., to verify the property. Symbolic execution is closely related to static verification techniques, such as static symbolic execution (SSE) [118, 116, 51], and weakest preconditions [155].

Throughout this thesis, we saw the boundaries between some of the above views become fuzzy, or completely fade away. For example, veritesting (Chapter 7) was presented as

a testing technique, but—if taken to the extreme with complete path exploration—could be used for verification. Similarly, the linear function (Chapter 6) transformation can be considered as an optimization, while SSE with GSA (Chapter 7) could be in any of the above categories. At its very root, symbolic execution allows us to express program computations in logic, a fundamental primitive for most analyses. Lying in the cross-section of testing, runtime analyses, and verification, symbolic execution seems to be a promising area for future research. In the rest of this section, we briefly present some of the main lessons learned (Section 8.1), as well as a list of problem areas in symbolic execution that may be interesting to explore further (Section 8.2).

8.1 Lessons Learned.

The full list of lessons learned would be too long to enumerate—even in a thesis. We tried to condense the list in four brief points, which we present below.

Measurements and Systematization. Symbolic executors are software systems. Being able to systematically test them on extensive benchmarks, measure performance/profiling metrics, and perform reproducible experiments is necessary to extract meaningful conclusions.

Representations Matter. Details matter, and even a small change in representation can have very substantial impact on performance (Chapters 6 and 7). Choose representations carefully, and only after extensive testing and measurements.

Use Domain Knowledge. Whenever available use domain knowledge (Chapters 4 to 6). Having a generic algorithm is a good first step, but going from toy examples to real-world programs typically requires domain knowledge.

Focus on State Space Management. Symbolic execution faces a number of scalability challenges (Chapter 3), and state explosion (at the program or solver level) is by far the

largest. Managing state explosion, and reasoning about multiple states simultaneously is necessary to scale to larger programs.

8.2 Problem Areas and Open Questions.

We conclude this thesis with a short list of possible problems and questions that could be addressed in future work.

When, Where, and How to Merge. The veritesting algorithm we presented (Chapter 7) offers a base algorithm for merging execution states. Combining with existing merging heuristics [26], or developing new ones; adopting different strategies for identifying merge points; or applying optimizations to our formula generation algorithm would all be very welcome as future work. Identifying when, where and how to merge for achieving the best performance for an arbitrary program/executor are still open questions.

SMT Solvers, Symbolic Execution, and How to Coevolve. Symbolic execution was invented around 1975 [16, 17, 18]. The field remained dormant until about 2005, when the area exploded with hundreds of research papers [15] within the past decade. The timing is not a coincidence; recent advances in SMT technology (back in 2005) allowed symbolic execution to become more practical [25]. Since 2005 SMT solvers have evolved further and progress is expected to continue. Gradually moving from single-path to multi-path analysis is the straightforward way for capitalizing on new SMT advancements. Such a design shift, would make symbolic executors rely more heavily on SMT solvers for performance, and executor implementers would have to be more conscious of SMT limitations, and internal state. Similarly, formulas generated from symbolic execution could provide SMT developers with concrete, useful instances of hard to solve formulas. Ultimately, symbolic execution could become a solver submodule with full access to the solver state and specific primitives for handling forking and translation from binary (or other language) to formulas. The dynamics

between symbolic execution and SMT solving are hard to predict, but attempts to bring the two fields closer together could benefit the development of both. Exactly how this can be achieved remains an open question.

Automatic term rewriting. Symbolic execution engines rely heavily on expression simplification, and most engines typically have a long list of hard-coded rules for reducing one expression to another. Compiling such lists takes time, and new rules are added on a by-need basis; when the solver keeps timing out on similar formulas, the executor developers manually investigate and add a simplification rule to resolve the problem. Such rewriting rules can be highly effective in improving performance, and being able to generate them automatically would be invaluable. The recent work of Romano *et al.* [52] is a great step in that direction. What remains to be determined, is whether such rewriting rules should be added at the solver level (so that others can benefit from them) or they should remain at the symbolic executor level (if the rules are only applicable on formulas generated from symbolic execution).

Functions. Symbolic execution algorithms presented in this dissertation operated on expressions and formulas (QF_BV and QF_ABV) that were similar to circuits, there was no notion of a function. We believe that a possible next step would be to introduce functions to our language; recursion would be the follow-up. Our inability to use functions translated to a number of missed opportunities. For example, consider two α-equivalent expressions $x + x + x$ and $y + y + y$. Even with hash-consing (Chapter 7) enabled, these two expressions would not share any common structure. However, if functions were available, we could have a single shared function $f(x) = x + x + x$, and two applications of the same function (a form of functional hash-consing). Our initial attempts on small benchmarks with function-based simplifications showed substantial improvements in expression sizes (the functions can effectively refactor existing expressions), but these did not translate to better solving times—current solver implementations perform substitution for non-recursive functions. We expect that enabling functions in our formulas could lead to significant performance improvements.

Tuning symbolic execution. At the core of each symbolic execution system lies a highly parameterizable interpreter for manipulating symbolic expressions. The performance of the symbolic execution engine typically depends on two factors: 1) user-provided symbolic execution parameters (e.g., concretize memory or not?), and 2) the analyzed program. We have empirically found that in many cases, the problem of effectively analyzing a program reduces to the problem of *carefully* selecting the symbolic execution parameters. Being able to automatically generate "generally good" configuration parameters for symbolic execution could be invaluable for practical systems. Automatically customizing analysis parameters for a given program and employing program-aware simplifications/optimizations would be even more valuable, but would also be more costly.

Combining Blackbox and Whitebox Analyses. This thesis focused entirely on white-box techniques for bug-finding. However, there is a wealth of work in blackbox techniques such as mutational fuzzing [10, 11, 156, 157], which are heavily used—with great success—especially in industry. Simpler to setup, and with typically higher throughput in terms of executions per second, blackbox fuzzing is capable of finding vulnerabilities in real-world software. Gracefully combining blackbox with whitebox techniques may help mitigate the drawbacks of each technique. Software companies are already starting to follow such a holistic approach [13], and more research on how to combine such heterogeneous techniques could be very valuable.

Test Case Generation Competitions. Competitions are a great way of making progress within a field. SMTcomp [158] is the perfect example of how a well-designed competition that is open to everyone can lead to advancements in a field. Creating a similar competition for symbolic executors or more generally for test case generation tools could be very beneficial for software testing as a whole.

Automatic bug fixing. Symbolic execution and other bug-finding techniques are progressing rapidly. However, at the end of the chain, the human developer is still responsible for manually fixing every reported bug. Currently, it appears that developers cannot keep up with the rate of bug reports (this was also part of the motivation for our work on AEG). Out of the 1,182 bug reports we submitted to the Debian bug tracker with our work on veritesting (Chapter 7), only 202 (16%) was fixed within 11 months after the report. Research in the area of software repair and automatic bug fixing, could potentially alleviate the burden of the developers. Symbolic execution and test case generation tools are a good starting point, considering that false positives are virtually eliminated, e.g., a program crash is—in most cases—an indicator of bad behavior.

Bibliography

[1] Edward J Schwartz, David Brumley, and Jonathan M Mccune. A Contractual Anonymity System. In *Proceedings of the Network and Distribution System Security Symposium*, 2010. (Page v, 145)

[2] 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*, 2011. (Page v, 32, 56, 92, 93, 99, 100, 148, 149, 158, 161, 179, 184)

[3] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012. (Page v, 4, 8, 43, 44, 47, 49, 50, 52, 60, 62, 64, 65, 67, 68, 91, 94, 95, 99, 100, 188, 195, 210, 212, 213, 232)

[4] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Communications of the ACM*, 57(2):74–84, 2014. (Page v, 49)

[5] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1083–1094, New York, New York, USA, 2014. ACM Press. (Page v, 9, 50, 66, 67, 213)

[6] Ariane. The Ariane Catastrophe. http://www.around.com/ariane.html. (Page 3)

[7] Ralph Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy Magazine*, 9(3):49–51, 2011. (Page 3)

[8] CNN. Toyota recall costs: $2 billion. http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm, 2010. (Page 3)

[9] Institute Systems Sciences IBM. It is 100 Times More Expensive to Fix Security Bug at Production Than Design. https://www.owasp.org/images/f/f2/Education_Module_Embed_within_SDLC.ppt. (Page 3)

[10] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990. (Page 3, 144, 246)

[11] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited : A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report October 1995, 1995. (Page 3, 246)

[12] Patrice Godefroid, Michael Y Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium*, number July, 2008. (Page 3, 5, 43, 47, 49, 54, 57, 62, 64, 65, 67, 68, 74, 94, 122, 124, 149, 157, 161, 162, 165, 184, 194, 196, 232)

[13] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, 2012. (Page 3, 8, 43, 101, 191, 246)

[14] RS Pressman and D Ince. *Software engineering: a practitioner's approach*. 1992. (Page 4)

[15] ASER Group. Online Bibliography for Symbolic Execution. https://sites.google.com/site/asergrp/bibli/symex. (Page 4, 244)

[16] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. (Page 4, 36, 90, 114, 148, 237, 242, 244)

[17] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975. (Page 4, 242, 244)

[18] W.E. Howden. Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, C-24(5):554–560, 1975. (Page 4, 242, 244)

[19] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE : Automatically Generating Inputs of Death. In *Proceedings of the ACM Conference on Computer and Communications Security*, New York, NY, USA, 2006. ACM. (Page 4, 54, 65, 67, 144, 188, 232)

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART : Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005. ACM. (Page 4, 54, 188, 191)

[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. (Page 4, 5, 7, 32, 47, 49, 54, 55, 59, 62, 64, 67, 68, 69, 92, 114, 125, 127, 134, 137, 144, 149, 158, 160, 161, 162, 188, 190, 194, 195, 209, 212, 213, 214, 232)

[22] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, pages 117–130. ACM, 2007. (Page 4, 106, 144, 184, 232)

[23] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. Technical report, Microsoft MSR-TR-2012-55, 2012. (Page 4, 70, 90, 232)

[24] Juan Caballero, Pongsin Poosankam, Stephen Mccamant, Domagoj Babic, and Dawn Song. Input Generation via Decomposition and Re-Stitching : Finding Bugs in Malware Categories and Subject Descriptors. In *ACM Conference on Computer and Communications Security*, pages 413–425, 2010. (Page 4, 184, 232)

[25] C Cadar and K Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013. (Page 4, 56, 69, 193, 232, 244)

[26] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the ACM Confrence on Programming Language Design and Implementation*, pages 193–204, 2012. (Page 4, 32, 69, 190, 191, 193, 194, 195, 203, 211, 212, 213, 214, 215, 218, 225, 233, 244)

[27] Trevor Hansen, Peter Schachte, and Harald Sø ndergaard. State joining and splitting for the symbolic execution of binaries. *Runtime Verification*, pages 76–92, 2009. (Page 4, 194, 203, 232)

[28] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Design Automation Conference*, pages 368–371, 2003. (Page 4, 235)

[29] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, New York, NY, USA, 2011. ACM. (Page 4, 43, 44, 47, 49, 56, 59, 60, 62, 68, 94, 149, 158, 160, 161, 183, 184, 195, 214, 232)

[30] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the European Software Engineering Conference*, page 263, New York, New York, USA, 2005. ACM Press. (Page 5, 54, 64, 67, 122, 124, 149, 191, 196, 232)

[31] A Thakur, J Lim, A Lal, A Burton, E Driscoll, M Elder, T Andersen, and T Reps. Directed Proof Generation for Machine Code. In *International Conference on Computer Aided Verification*, pages 1–17, 2010. (Page 8, 94, 149, 165, 184, 232)

[32] Dawn Song, David Brumley, Heng Yin, and Juan Caballero. BitBlaze: A new approach to computer security via binary analysis. In *4th International Conference on Information Systems Security*, pages 1–25, 2008. (Page 8, 44, 57, 149, 165, 183)

[33] Domagoj Babic and Alan J Hu. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the International Conference on Software Engineering*, pages 211–220, New York, NY, USA, 2008. ACM. (Page 8, 193, 194)

[34] Cormac Flanagan and JB Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205, New York, NY, USA, 2001. ACM. (Page 8, 75, 76, 194, 234)

[35] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate Abstraction with Adjustable-Block Encoding. In *Formal Methods in Computer Aided Design*, pages 189–197, 2010. (Page 9, 236)

[36] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *International Conference on Computer Aided Verification*, number 97, 2000. (Page 9, 235, 236)

[37] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. (Page 15)

[38] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. (Page 15)

[39] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. (Page 15, 19)

[40] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. (Page 15)

[41] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011. (Page 16, 22, 165, 173, 198, 210)

[42] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, N.J., 1976. (Page 26, 234)

[43] Leonardo de Moura and Nikolaj Bjø rner. Satisfiability Modulo Theories: An Appetizer. In *Brazilian Symposium on Formal Methods*, pages 23–36, 2009. (Page 28)

[44] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection , Analysis , and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium*, 2005. (Page 31, 49, 144, 155, 162)

[45] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed Symbolic Execution. In *International Symposium on Static Analysis*, pages 95–111, 2011. (Page 32, 55)

[46] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9 : A Software Testing Service 1 Introduction 2 Software Testing as a Service 3 Parallel Symbolic Execution. In *SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, number October, 2009. (Page 34, 204)

[47] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems*, pages 183–198. ACM Press, 2011. (Page 34, 212, 213, 232)

[48] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight Detection of Infinite Loops at Runtime. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 161–169. Ieee, November 2009. (Page 37, 77)

[49] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the USENIX Security Symposium*, pages 379–394, 2011. (Page 44, 93, 97, 100, 182)

[50] Torbjörn Granlund and PL Montgomery. Division by invariant integers using multiplication. In *Programming Language Design and Implementation*, pages 61–72, 1994. (Page 50)

[51] Domagoj Babic. *Exploiting structure for scalable software verification.* PhD thesis, University of British Columbia, Vancouver, Canada, 2008. (Page 50, 199, 204, 209, 232, 235, 242)

[52] Anthony Romano and Dawson Engler. Expression Reduction from Programs in a Symbolic Binary Executor. In *Proceedings of the 20th International Symposium Model Checking Software*, pages 301–319. Springer, 2013. (Page 50, 245)

[53] Mark Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981. (Page 51)

[54] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *Programming Language Design and Implementation*, pages 246–256, 1990. (Page 51)

[55] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426, May 2007. (Page 54, 184)

[56] Nikolai Tillmann and Jonathan De Halleux. Pex – White Box Test Generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153, 2008. (Page 55, 232)

[57] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. Ieee, June 2009. (Page 55)

[58] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, 2010. (Page 55)

[59] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, Berlin, Heidelberg, 2008. Springer-Verlag. (Page 56, 233)

[60] 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, August 2009. (Page 56, 69, 232)

[61] 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*, pages 317–331. IEEE, 2010. (Page 56, 69, 232)

[62] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In IEEE Press, editor, *Proceedings of the 35th IEEE International Conference on Software Engineering*, pages 122–131, Piscataway, NJ, USA, 2013. (Page 57, 67, 188, 193, 209)

[63] 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 *Programming Language Design and Implementation*, pages 190–200. ACM, 2005. (Page 60, 173, 211)

[64] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. *Compiler Construction*, 2004. (Page 65, 167)

[65] Eiichi Goto. Monocopy and Associative Algorithms in Extended Lisp. 1974. (Page 66, 209, 232)

[66] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008. (Page 67, 68, 173, 211)

[67] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Network and Distributed System Security Symposium*. The Internet Society, 2011. (Page 69)

[68] Edward 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 22nd USENIX Security Symposium*, pages 353–368. USENIX, 2013. (Page 69)

[69] Martin Davis, G Logemann, and D Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 1962. (Page 71)

[70] Joao P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, 1996. (Page 71)

[71] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope : A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. 2009. (Page 74)

[72] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard - Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010. (Page 74)

[73] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width. In *SMT Workshop*, volume 23, pages 1–12, 2012. (Page 74)

[74] Randal E Bryant, Daniel Kroening, Sanjit A Seshia, Ofer Strichman, and Bryan Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2007. (Page 74)

[75] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 82–87. ACM, 2005. (Page 76, 234)

[76] K Rustan M Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005. (Page 76, 194)

[77] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth. An Efficient Method of Computing Static Single Assignment Form. In *ACM Symposium on Principles of Programming Languages*, 1989. (Page 76)

[78] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007. (Page 77)

[79] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, page 23, New York, New York, USA, 2011. ACM Press. (Page 77, 234)

[80] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses*, pages 86–106. Springer, 2012. (Page 81)

[81] Leyla Bilge and Tudor Dumitras. Before We Knew It—An Empirical Study of Zero-Day Attacks in the Real World. In *Proceedings of the 2012 ACM Conference on Computer and communications Security*, pages 833–844. ACM, 2012. (Page 81)

[82] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4):1–54, 2009. (Page 89)

[83] Leonardo de Moura and Nikolaj Bjø rner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69, September 2011. (Page 90)

[84] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007. (Page 93, 97)

[85] Pratyusa K Manadhata. *An Attack Surface Metric*. Phd, Carnegie Mellon University, 2008. (Page 96)

[86] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. *Proceedings of the 11th ACM conference on Computer and communications security CCS 04*, page 298, 2004. (Page 97, 108)

[87] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. Automatic discovery of API-level exploits. In *Proceedings of the 27th International Conference on Software Engineering*, pages 312–321. ACM, 2005. (Page 98)

[88] Jason Medeiros. Automated Exploit Development, The Future of Exploitation is Here. Technical report, Grayscale Research, 2007. (Page 98)

[89] Lurene Grenier and Lin0xx. Byakugan: Increase Your Sight. Technical report, 2007. (Page 98)

[90] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2008. (Page 98, 99, 144, 183)

[91] Sean Heelan. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. PhD thesis, University of Oxford, 2009. (Page 99, 144, 172, 184)

[92] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000. (Page 99)

[93] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING : Finding Name Resolution Vulnerabilities in Programs. In *USENIX Security Symposium*, 2012. (Page 100)

[94] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers for Software Security. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, pages 1–12, 2012. (Page 100)

[95] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Laszlo Szekeres, Stephen McCamant, and Dawn Song. Transformation-aware Exploit Generation using a HI- CFG. Technical report, 2013. (Page 100)

[96] Shih-kun Huang, Min-hsiang Huang, Po-yen Huang, Chung-wei Lai, Han-lin Lu, and Wai-meng Leong. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In *IEEE International Conference on Software Security and Reliability*, 2012. (Page 100)

[97] Shih-kun Huang, Han-lin Lu, Wai-meng Leong, and Huan Liu. CRAXweb : Automatic Web Application Testing and Attack Generation. In *International Conference on Software Security and Reliability*, 2013. (Page 100)

[98] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*, 2013. (Page 100)

[99] Gustavo Grieco, Laurent Mounier, and A General Approach. A stack model for symbolic buffer overflow exploitability analysis. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 216–217, 2013. (Page 100)

[100] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Theory and Techniques for Automatic Generation of Vulnerability-Based Signatures. *IEEE Transactions on Dependable and Secure Computing*, 5(4):224–241, October 2008. (Page 106, 144)

[101] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the Symposium on Code Generation and Optimization*, pages 75–86, 2004. (Page 110, 233)

[102] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, 2007. (Page 135)

[103] Sang Kil Cha, Brian Pak, David Brumley, and Richard Jay Lipton. Platform-Independent Programs. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 547–558, New York, New York, USA, 2010. ACM Press. (Page 143)

[104] Umesh Shankar, Talwar Kunal, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security Symposium*, 2001. (Page 144)

[105] Rob Johnson and David Wagner. Finding User / Kernel Pointer Bugs With Type Inference. In *13th USENIX Security Symposium*, 2004. (Page 144)

[106] V Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005. (Page 144)

[107] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH Computer Architecture News*, 32(5):85, December 2004. (Page 144)

[108] Dawson Engler and Daniel Dunbar. Under-constrained Execution : Making Automatic Code Destruction Easy and Scalable. In *International Symposium on Software Testing and Analysis*, volume 0, pages 0–3, 2007. (Page 145)

[109] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007. (Page 149)

[110] 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). *2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010. (Page 183)

[111] BitTurner. Binary Analysis. `http://www.bitturner.com`. (Page 184)

[112] Lorenzo Martignoni, Stephen Mccamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-Exploration Lifting : Hi-Fi Tests for Lo-Fi Emulators. In *Architectural Support for Programming Languages and Operating Systems*, 2012. (Page 184, 232)

[113] Mayhem. Open Source Statistics & Analysis, 2013. (Page 189, 211)

[114] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. (Page 190)

[115] D Molnar, XC Li, and DA Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium*, pages 67–82, 2009. (Page 190)

[116] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *ACM Symposium on Principles of Programming Languages*, volume 40, pages 351–363, January 2005. (Page 193, 194, 232, 235, 242)

[117] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, Complete and Scalable Path-Sensitive Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, New York, NY, USA, 2008. ACM. (Page 193, 194)

[118] 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. (Page 194, 232, 235, 242)

[119] Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proceedings of the International Conference on Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer-Verlag. (Page 194, 235)

[120] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008. (Page 197)

[121] Sebastien Bardin, Philippe Herrmann, Jerome Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The BINCOA Framework for Binary Code Analysis. In *Proceedings of the International Conference on Computer Aided Verification*, pages 165–170, Berlin, Heidelberg, 2011. Springer-Verlag. (Page 197)

[122] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, volume 30 of *PLDI '95*, pages 47–55. ACM, 1995. (Page 199, 204, 232)

[123] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Page 199)

[124] RL Rudell. *Multiple-valued Logic Minimization for PLA Synthesis*. Number June. 1986. (Page 203)

[125] JC Filliâtre and S Conchon. Type-safe modular hash-consing. In *Proceedings of the Workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM. (Page 210)

[126] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 716–726, Piscataway, NJ, USA, June 2012. IEEE Press. (Page 212, 213)

[127] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE Coreutils Experiment. `http://klee.github.io/klee/CoreutilsExperiments.html`, 2008. (Page 212)

[128] Mayhem. 1.2K Crashes in Debian, 2013. (Page 222)

[129] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–342, 2013. (Page 232)

[130] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *ACM Symposium on Principles of Programming Languages*, pages 177–189, New York, NY, USA, 1983. ACM Press. (Page 233)

[131] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -OVERIFY : Optimizing Programs for Fast Verification. In *14th Workshop on Hot Topics in Operating Systems*. USENIX, 2013. (Page 233)

[132] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems*, pages 315–328, New York, NY, USA, 2011. ACM Press. (Page 233)

[133] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, New York, NY, USA, 2007. ACM. (Page 234)

[134] Virgil D. Gligor. *A Guide to Understanding Security Testing and Test Documentation*. National Computer Security Center, 1994. (Page 234)

[135] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of the Theory and Practice of Software, International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag. (Page 234)

[136] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*, page 225, 2009. (Page 234)

[137] K Rustan M Leino, Michal Moskal, and Wolfram Schulte. Verification Condition Splitting. 2008. (Page 234)

[138] Patrick M. Cousot. An Introduction to A Mathematical Theory of Global Program Analysis. 1977. (Page 234, 235)

[139] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, 1979. (Page 234, 235)

[140] Matthew S. Hecht and Jeffrey D. Ullman. A Simple Algorithm for Global Data Flow Analysis Problems. *SIAM Journal on Computing*, 4(4):519, 1975. (Page 235)

[141] Edmund M Clarke, Odna Grumberg, and David E Long. Model Checking and Abstraction. *ACM transactions on Programming Languages and Systems*, (September):1512–1542, 1994. (Page 235)

[142] Susanne Graf and Hassen Saïdi. Verifying Invariants Using Theorem Proving. In *Computer Aided Verification*, 1997. (Page 235)

[143] P Cousot and Radhia Cousot. Modular static program analysis. *Compiler Construction*, pages 1–20, 2002. (Page 235)

[144] Edmund M Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic, 1981. (Page 235)

[145] Dirk Beyer, Thomas a. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, September 2007. (Page 235)

[146] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–3, 2002. (Page 235)

[147] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC : Bounded Model Checking of C and C ++ Programs Using a Compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, pages 146–161, Philadelphia, PA, USA, 2012. Springer-Verlag. (Page 235)

[148] Patrick Cousot and Radhia Cousot. Compositional and Inductive Semantic Definitions in Fixpoint, Equational, Constraint, Closure-condition, Rule-based and Game-Theoretic Form. In *Computer Aided Verification*, 1995. (Page 235)

[149] David A. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretations. In *ACM Symposium on Principles of Programming Languages*, 1998. (Page 235)

[150] Bernhard Steffen. Data Flow Analysis as Model Checking. *Theoretical Aspects of Computer Software*, pages 346–364, 1991. (Page 235)

[151] Kenneth L McMillan. *Symbolic Model Checking*. PhD thesis, 1992. (Page 235)

[152] Radek Pelánek. Fighting State Space Explosion : Review and Evaluation. *Formal Methods for Industrial Critical Systems*, (201), 2009. (Page 236)

[153] A. J. Romano. Linux Bug Release, July 2013. (Page 236)

[154] George Candea. The Tests-versus-Proofs Conundrum. *IEEE Security & Privacy Magazine*, 12(February):65–68, 2014. (Page 236)

[155] Edsger W. Dijkstra. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968. (Page 242)

[156] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM Conference on Computer & Communications Security*, pages 511–522, 2013. (Page 246)

[157] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, August 2014. (Page 246)

[158] SMTComp. SMT Competition. http://smtcomp.org. (Page 246)