# Reasoning about Software Security via Synthesized Behavioral Substitutes

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

vorgelegt von

**Tim Blazytko**
aus Essen

2020

| Gutachter: | Prof. Dr. Thorsten Holz<br>Prof. Dr. Konrad Rieck |
| --- | --- |
| Tag der mündlichen Prüfung: | 16.07.2020 |

# Abstract

Since software is an integral part of our daily life, it is of great importance to ensure its safety and security. Still, we frequently observe attackers exploiting security vulnerabilities to steal secret customer information, manipulate essential data and take control over critical infrastructure. On the other hand, security researchers make efforts to find and eliminate security flaws, which is often a non-trivial task. Even more, it is proven to be *undecidable*.

As a consequence, security analysis is often goal-oriented, effectively limiting the analysis scope by focusing only on certain types of bugs or proving the presence of specific program characteristics. Therefore, analysis techniques are based on assumptions that may only hold in artificial scenarios. In practice, such methods are either too broad and suffer from false positives or too narrow and miss many cases. Still, they often are very effective for their designed use case and alleviate tedious and time-consuming work of a human analyst.

Some techniques are based on *abstraction* in which we transform parts of a program into an abstract domain that is explicitly constructed to facilitate reasoning about specific characteristics. In this domain, a so-called *behavioral substitute* represents only the desired characteristics of a given program. Often, the transformation process relies on labor-intensive manually implemented rules, resulting in behavioral substitutes that are too generic or incomplete in some cases.

In this thesis, we propose problem-specific analysis techniques based on synthesized behavioral substitutes to advance research on topics related to *code deobfuscation*, *fuzzing* and *root cause analysis*. In each case, we design a domain-specific representation that allows *generic* reasoning in its associated area. We apply stochastic *program synthesis* techniques to automatically *learn* behavioral substitutes. For this, we use the target program as a black-box, basically using the program's behavior as feedback. This combination of crafting *target-specific* representations in a *problem-specific* domain allows us to reason about more generic instances of the problem while staying close to the target.

As a consequence, our methods are generic regarding the problem and geared to the target, allowing us to operate on a wide range of problem instances without implementing a target-specific analysis. In our empirical evaluation, we show for various real-world targets that we either outperform state-of-the-art approaches or that our techniques are orthogonal to existing approaches and perform in scenarios where others do not.

# Zusammenfassung

Software spielt eine wichtige Rolle in unserem Leben. Daher ist es essenziell, sicherzustellen, dass sie fehlerfrei und sicher gegen Angriffe ist. Wir beobachten jedoch regelmäßig, dass Angreifer Sicherheitslücken ausnutzen, um geheime Informationen zu stehlen, Daten zu manipulieren und um die Kontrolle über kritische Infrastruktur zu erlangen. Andererseits arbeiten Sicherheitsforscher permanent daran, Schwachstellen in Software zu finden und zu beheben. Dies ist jedoch keine triviale Aufgabe, sondern bewiesenermaßen *unentscheidbar*.

Daher sind Sicherheitsanalysen oft auf wenige spezifische Fragestellungen begrenzt; beispielsweise werden nur bestimmte Fehlerklassen oder das Vorhandensein bestimmter Programmeigenschaften betrachtet. Als Konsequenz davon treffen solche Analysetechniken Annahmen, die oft nur in künstlichen Szenarien zutreffen. In der Praxis sind diese Methoden entweder zu oberflächlich und produzieren False Positives oder sie sind zu eingeschränkt, sodass sie viele Szenarien nicht finden. Nichtsdestotrotz sind solche Analysen sehr effektiv in ihren jeweiligen Anwendungsfällen und erleichtern die mühselige und zeitaufwändige Arbeit menschlicher Analysten.

Manche dieser Techniken basieren auf *Abstraktion*: Ein Programm wird in einer abstrakten Domäne abgebildet, die Aussagen über spezielle Programmeigenschaften erleichtert. Die zu untersuchenden Eigenschaften des ursprünglichen Programms werden hierbei durch einen *verhaltensbasierten Repräsentanten* dargestellt. Die Regeln für diesen Transformationsprozess müssen meist mühselig von Hand implementiert werden, weshalb die so erstellten verhaltensbasierten Repräsentanten entweder zu generisch oder unvollständig für viele Analysen sind.

In dieser Arbeit stellen wir problemspezifische Analysetechniken für Code Deobfuscation, Fuzzing und Root-Cause-Analyse vor, die verhaltensbasierte Repräsentanten automatisch synthetisieren, anstatt sie nach einem manuell erstellten Regelwerk zu übersetzen. Dazu konstruieren wir stets zuerst eine domänenspezifische Repräsentation, in welcher Schlussfolgerungen über generische Probleminstanzen getroffen werden können. Anschließend verwenden wir Methoden der stochastischen Programmsynthese, um einen verhaltensbasierten Repräsentanten eines Zielprogramms zu lernen. Dabei dient das Zielprogramm als Orakel, dessen Verhalten als Rückinformation für die Synthese dient. Diese Modellierung einer problemspezifischen Domäne zusammen mit dem Lernen programmspezifischer Repräsentationen ermöglicht es, über generische Probleminstanzen zu urteilen und dabei trotzdem nah am Zielprogramm zu bleiben.

Diese Symbiose erlaubt uns, über zahlreiche Probleme Schlussfolgerungen anzustellen, ohne programmspezifische Analysen implementieren zu müssen. In unserer empirischen Evaluation zeigen wir dies anhand einiger Programme aus diversen Anwendungsbereichen. Wir demonstrieren, dass unsere Methode andere moderne Ansätze entweder übertrifft oder orthogonal zu diesen ist – somit eignet sie sich auch für Fälle, in denen die bisherigen Ansätze nicht funktioniert haben.

# Acknowledgment

First, I would like to express my gratitude to Prof. Dr. Thorsten Holz for being an excellent advisor. You always gave me the freedom to follow my research ideas. Providing me with valuable suggestions and honest feedback at all times, it was a great experience working with you.

I would also like to thank Prof. Dr. Konrad Rieck. Since our initial meeting in 2015, you were always open-minded and very supportive. It is an honor to have you on the thesis committee.

In particular, I would like to thank my collaborators, Moritz Schlögel, Cornelius Aschermann, Moritz Contag, Sergej Schumilo, Simon Wörner, Ali Abbasi, Joel Frank, Philipp Koppe and Benjamin Kollenda. It was very inspiring and great fun working with you. I have learned a lot during our mutual time. Most of all, I enjoyed our discussions and teamwork, spending more nights than mornings together, writing papers and tyrannizing students.

Talking about students—a big thank you to our skilled research assistances in no particular order: Julius Basler, Marcel Bathke, Berthold Dors and Robert Stark. You helped us enormously with our experiments and speeded up our research.

Moreover, I would also like to thank my current and former colleagues, especially Emre Güler, Andre Pawlowski, Thorsten Eisenhofer, Nils Bars, Dennis Tatang, Robert Gawlik, Behrad Garmany, Teemu Rytilahti, Philipp Görz, Tobias Scharnowski, Florian Quinkert and Lea Schönherr. Your support, feedback, as well as our discussions, were always fruitful and helped me to proceed in my research.

Finally, I want to express my gratitude to my parents Martina and Achim, my life partner Katharina and my friends Sven, Niko, Annika and Andi. You always supported me and brought me back to earth when I was in outer space. Guyz, you rock!

# Contents

**Abstract** **iii**

**Zusammenfassung** **v**

**Acknowledgment** **vii**

**Contents** **ix**

**1 Introduction** **1**
1.1 Topic of this Work . . . 2
1.1.1 Behavioral Substitutes . . . 2
1.1.2 Code Deobfuscation . . . 3
1.1.3 Fuzzing . . . 4
1.1.4 Root Cause Analysis . . . 5
1.2 Contributions . . . 5
1.3 List of Publications . . . 6

**2 Synthesis of Behavioral Substitutes** **9**
2.1 Program Synthesis . . . 9
2.2 SMT-based Synthesis . . . 10
2.2.1 Preliminaries . . . 10
2.2.2 Search Space and Methods . . . 11
2.3 Stochastic Synthesis . . . 13
2.3.1 Stochastic Optimization . . . 13
2.3.2 Search Methods . . . 15
2.4 Program Synthesis in the Context of This Work . . . 18

**3 Expression Synthesis to Simplify Obfuscated Code** **21**
3.1 Introduction . . . 21
3.2 Challenges in Code Deobfuscation . . . 23
3.2.1 Obfuscation . . . 23
3.2.2 Trace Simplification . . . 27
3.3 Design . . . 27
3.3.1 Trace Dissection . . . 28

3.3.2 Random Sampling . . . 28
3.3.3 Synthesis . . . 29
3.4 Program Synthesis . . . 29
3.4.1 Node Selection . . . 30
3.4.2 Grammar . . . 30
3.4.3 Random Playout . . . 32
3.4.4 Measuring Similarity of Outputs . . . 33
3.4.5 Backpropagation . . . 33
3.4.6 Expression Simplification . . . 34
3.4.7 Algorithm Configuration . . . 34
3.5 Implementation . . . 34
3.6 Experimental Evaluation . . . 35
3.6.1 Parameter Choice . . . 35
3.6.2 Mixed Boolean-Arithmetic . . . 36
3.6.3 VM Instruction Handler . . . 38
3.6.4 ROP Gadget Analysis . . . 41
3.7 Discussion . . . 42
3.8 Related Work . . . 43
3.9 Conclusion . . . 45

**4 Input Structure Synthesis to Guide Feedback-driven Fuzzing 47**
4.1 Introduction . . . 47
4.2 Challenges in Fuzzing Structured Languages . . . 49
4.2.1 Black-box Fuzzing . . . 49
4.2.2 Gray-box Fuzzing . . . 50
4.2.3 Structured Gray-box Fuzzing . . . 52
4.2.4 Grammar Inference . . . 52
4.2.5 Shortcomings of Existing Fuzzers . . . 52
4.3 Design . . . 53
4.3.1 Input Generalization . . . 54
4.3.2 Input Mutation . . . 55
4.4 Implementation . . . 59
4.5 Experimental Evaluation . . . 59
4.5.1 Measurement Setup . . . 60
4.5.2 State-of-the-Art Bug Finding Tools . . . 61
4.5.3 Structured Gray-box Fuzzers . . . 64
4.5.4 Grammar Inference Techniques . . . 66
4.5.5 Mutations Statistic . . . 67
4.5.6 Real-World Bugs . . . 68
4.6 Discussion . . . 68
4.7 Related Work . . . 71
4.8 Conclusion . . . 72

**5 Predicate Synthesis to Automate Root Cause Explanation 73**
5.1 Introduction . . . 73
5.2 Challenges in Root Cause Analysis . . . 76

5.2.1 Running Example . . . . . 76
5.2.2 Crash Triaging . . . . . 77
5.3 Design . . . . . 78
5.3.1 Input Diversification . . . . . 79
5.3.2 Monitoring Input Behavior . . . . . 79
5.3.3 Explanation Synthesis . . . . . 80
5.4 Predicate-based Root Cause Analysis . . . . . 81
5.4.1 Predicate Types . . . . . 81
5.4.2 Predicate Evaluation . . . . . 82
5.4.3 Synthesis of Constant Values . . . . . 84
5.4.4 Ranking . . . . . 85
5.5 Implementation . . . . . 86
5.6 Experimental Evaluation . . . . . 86
5.6.1 Setup . . . . . 87
5.6.2 Experiment Design . . . . . 89
5.6.3 Results . . . . . 90
5.6.4 Case Studies . . . . . 92
5.7 Discussion . . . . . 97
5.8 Related Work . . . . . 98
5.9 Conclusion . . . . . 99

**6 Conclusion** **101**
6.1 Limitations . . . . . 102
6.2 Future Work . . . . . 103

**List of Figures** **105**

**List of Tables** **107**

**Bibliography** **109**

# Chapter 1

# Introduction

Software is an integral part of our life. It is used to guide (autonomous) vehicles, power medical devices, process financial transactions, protect private information and control critical infrastructure. In short, it has a multitude of application scenarios that are crucial to our everyday life. Hence, guaranteeing the safety and security of software is of utmost importance. Nevertheless, not a day passes without security-critical bugs being uncovered in all kinds of software. Some of the more recent examples are *Spectre* [136] and *Meltdown* [147] that are based on speculative execution and enable attackers to leak sensitive data from memory. Other prominent examples are *Heartbleed* [70], a bug in the cryptographic library `OpenSSL` that allowed attackers to remotely leak confidential information such as secret keys, and *Shellshock* [71] that enabled attackers to execute arbitrary commands remotely on the user's machine.

Bugs like these are regularly uncovered, although numerous security researchers give their best to identify and eliminate software vulnerabilities before malicious actors can exploit them. This brings up the question *why* the problem is not solved yet, considering the vast amount of resources spent on fixing it. Amongst others, this is due to the fact that findings bugs is not only a non-trivial task but proven to be *undecidable* in general [182]. While finding bugs is one important component of software security, the area is vastly more complex: software security covers topics ranging from code understanding to bug triaging, patching and exploitation, each of them dealing with problems that are in its core undecidable. Some elementary but undecidable questions are:

- Can a given program location be reached?
- Is there any way to trigger this specific bug?
- Is there a data flow between two instructions?
- Can a specific bug be misused by an attacker?
- Why does this particular input crash the program?

In general, it is impossible to answer these questions. However, in practice, various methods allow answering such questions at least *partially*. While it is mostly impossible

to prove that a characteristic cannot hold, it often can be proved that a certain aspect holds by providing a *proof of concept (PoC)*. This PoC can be an input that triggers a certain bug, reaches a given program state or crashes a program. It also can be a working exploit that allows an attacker to execute arbitrary code by misusing a software vulnerability. Crafting such a PoC can be a tedious task that requires a lot of manual work.

To reduce the effort for a human analyst, many techniques based on static analysis and dynamic analysis were introduced to partially automate these tasks. While techniques based on *static analysis* such as abstract interpretation [72, 132, 164], machine learning [226–228] and symbolic execution [48, 195] reason about the program without executing it, dynamic techniques such as taint tracking [191, 195, 223] and fuzzing [34, 179, 194, 230, 231] operate on concrete program runs. Other (static and dynamic) techniques are based on *abstraction*; they transform parts of the program into an *intermediate representation* or *behavioral substitute* that simplifies reasoning about certain program characteristics [50, 77, 129, 131].

In most cases, all these techniques are neither sound nor complete [200]. They are not *complete* since they only explore a limited space of program behavior. They are not *sound*, since they may generate false positives due to simplified assumptions that deviate from the real program context. Therefore, they still require a human in the loop to analyze the results. Nonetheless, this reduces the work of a human analyst in many real-world scenarios.

## 1.1 Topic of this Work

In this thesis, we present methods we developed to improve and facilitate reasoning about different aspects of software security. In detail, we improve techniques related to code deobfuscation, fuzzing and root cause analysis. To this end, we design *problem-specific* analysis techniques that are generic in their associated field and synthesize *target-specific* program representations that allow us to reason about generic instances while being geared to the target. In the following, we discuss the topic of this work in more detail. Furthermore, we highlight the limitations of other state-of-the-art approaches that tackle similar goals.

### 1.1.1 Behavioral Substitutes

When reasoning about software behavior, we are often interested in a specific scenario or goal. For instance, we might want to know whether two snippets of code are semantically equivalent, whether we can reach a defined program state or which input swaps a certain branch. While generic reasoning is undecidable, we often find answers to such questions by the means of abstraction: we use techniques such as abstract interpretation or work on intermediate representations.

In *abstraction interpretation* [164], we define and evaluate the instruction semantics of concrete operations in an abstract domain. While this kind of analysis decreases precision, it allows us to analyze specific characteristics efficiently. Examples for this

are detecting and removing dead branches [72] or resolving targets of indirect jumps in a control-flow graph [132].

When reasoning about assembly code in an automated manner, we require precise descriptions of the instruction semantics. However, assembly code mostly modifies flag registers and stack registers implicitly. Therefore, many approaches lift assembly code into an *intermediate representation* that models these side-effects in an explicit manner [50, 77, 129, 131]. On top of this intermediate representation, we can build more powerful analysis techniques. For instance, we can symbolically execute a path in the program and pass its constraints to an SMT solver—a program that efficiently solves bit-vector arithmetic problems. Thereby, we craft an input that triggers the symbolically executed path in a concrete program run. In another example, we can perform a taint analysis to track the program parts that depend on user-provided input [195].

In the context of this work, we call such intermediate representations *behavioral substitutes*. A behavioral substitute is a program in a domain-specific language that allows us to reason about specific program characteristics efficiently. To put it differently, the domain-specific languages are explicitly designed to simplify reasoning about desired program characteristics or behavior.

In contrast to other works, we do not explicitly specify the transformations that translate concrete program behavior into an abstract domain. Instead, we first design domain-specific languages and then apply techniques based on program synthesis to automatically *learn* behavioral substitutes while using the target program's behavior as a black-box. Afterward, we use these behavioral substitutes to solve problems related to code deobfuscation, fuzzing and root cause analysis. In the following, we give a brief introduction to these topics.

### 1.1.2 Code Deobfuscation

*Code obfuscation* aims to impede the understanding of what a piece of code does; it makes it harder to understand its *semantics*. Usually, we apply code transformations that preserve the core semantics but hide the actual calculations or control flow of the underlying code. Code obfuscation is mostly used to protect intellectual property in proprietary software or to evade malware analysis. In the following, we briefly discuss state-of-the-art code obfuscation and deobfuscation techniques.

Two common obfuscation techniques that aim to thwart static code analysis techniques are opaque predicates and control-flow flattening[63]. While *opaque predicates* insert fake branches in the control-flow graph based on random values, domain knowledge or simple arithmetic identities, *control-flow flattening* removes the structure of control-flow graphs by introducing a state-machine whose states always return to a central dispatcher. Although these techniques can be defeated statically [72, 159], they offer little protection against a dynamic analysis setting.

*Virtual-machine (VM)*-based obfuscation defines a custom CPU in software which interprets the unobfuscated code as bytecode. To reconstruct the underlying high-level

code, a human analyst has to identify the different VM components, reverse engineer the semantics of all VM handlers and, finally, reconstruct the control-flow graph of the protected code. This process often is very time-consuming and tedious [183].

Arithmetic encodings or *Mixed Boolean-Arithmetic (MBA)* [236] replace simple arithmetic expressions with syntactically complex ones. As a consequence, they hide semantically simple calculations in syntactically complex encodings. Although there exist some approaches to simplify specific arithmetic encodings [80, 81, 101], it generally remains an NP-hard problem [144].

State-of-the-art code deobfuscation techniques combine taint analysis, symbolic execution and compiler optimizations [66, 225]. While compiler optimizations often remove junk code and computations of constant data via constant propagation, taint analysis is used to remove non-input dependent instructions. Furthermore, symbolic execution simplifies and summarizes common code operations. Although taint analysis and symbolic execution are quiete powerful, their performance scales with the syntactical complexity of the underlying code. As a consequence, they fail on large amounts of code and cannot deobfuscate syntactically complex expressions such as arithmetic encodings, even if the underlying semantics are simple.

In this work, we introduce a code deobfuscation technique that is based on program synthesis. Instead of analyzing the obfuscated code itself, we learn semantically equivalent expressions based on its input-output (I/O) behavior. We use this as a building block to simplify obfuscation techniques based on arithmetic encodings and automatically infer the semantics of VM instruction handlers.

### 1.1.3 Fuzzing

*Fuzzing* is a software testing technique that pseudo-randomly provides inputs to a program with the intention of causing software faults. These inputs can be generated randomly based on a defined set of rules or derived from some input. While a *black-box fuzzer* produces a constant stream of inputs without any internal knowledge of the target application, a *gray-box fuzzer* monitors the target application and uses the coverage produced by different inputs as guidance for generating new inputs. One of the best-known gray-box fuzzers is AFL [231], which started a new fuzzing wave of academic research. While it performs well on many targets, specific situations remain challenging to overcome. As a consequence, various fuzzers were built that aim to solve such challenges. For instance, REDQUEEN [34] introduces the concept of *input-to-state correspondence* to solve input constraints such as magic bytes and nested checksums, QSYM [230] and T-FUZZ [171] overcome hard constraints via symbolic execution.

Another aspect that affects the performance of fuzzers is the expected input structure of a target program. To fuzz targets such as language interpreters and parsers, *structured fuzzers* rely on provided input specifications that describe the expected input structure [33, 173]. Since these input specifications often have to be crafted manually, setting up a particular target is not only time-consuming but also drastically reduces the number of prospective targets overall.

In the context of this thesis, we introduce an approach that automatically infers structural patterns of input languages, basically learning an input description during fuzzing. In combination with structure-aware input mutations, we can produce target-specific structured inputs without relying on any provided input specification.

### 1.1.4 Root Cause Analysis

Once a fuzzer found a crashing input, our goal is, in the long term, to fix the underlying bug. Before we can find out *how* to fix the bug, we have to understand *why* the crash occurs. Often, a program crashes due to illegal memory reads or writes, stack and heap-based buffer overflows or use-after-free bugs. While we can use a debugger to inspect the program at its crashing location and see why the crash occurs, we have to *triage* the bug to find the *root cause*—the *real* cause of a crash that might be far earlier in the program's execution flow. This is often a non-trivial task since the root cause might be a logic error in a complex state machine, an uninitialized variable or an integer overflow (e. g., in the context of memory allocation), often located many thousands or millions of lines before the crashing location in the execution trace.

A human analyst can simplify the process by using tools such as `valgrind` [163] or sanitizers (e. g., ASAN [197] and MSAN [204]) that facilitate the detection of illegal memory access closer to the root cause. Furthermore, automated techniques [68, 69, 221] based on data flow analysis such as reverse execution and backward taint analysis start at the crashing location and isolate instructions that contribute to the bug. However, for sophisticated bugs like type confusions where there is no data flow edge between the root cause and the crashing location, root cause analysis remains a challenging task.

As part of this thesis, we introduce a generic approach for root cause analysis in which we statistically synthesize predicates that *locate* behavioral differences in crashing and non-crashing inputs. Based on these predicates, we can precisely pinpoint the root cause for different complex bug classes.

## 1.2 Contributions

In this thesis, we explore how to synthesize behavioral substitutes that allow us to solve problems related to different aspects of software security efficiently. We begin by providing a systematic overview of synthesizing behavioral substitutes. Afterward, we present three scenarios in which synthesized behavioral substitutes allow us to improve code deobfuscation, fuzzing and root cause analysis techniques. In summary, we make the following contributions.

**Chapter 2: Synthesis of Behavioral Substitutes.** In the foundational chapter of this thesis, we provide a systematic overview of different aspects of program synthesis with the goal to synthesize behavioral substitutes. We start by giving an introduction to program synthesis, followed by a brief introduction to SMT solvers and synthesis based on logical reasoning. Afterward, we illustrate program synthesis in a stochastic

setting. We conclude by discussing how the techniques described in following chapters of this thesis are modeled as synthesis problems.

**Chapter 3: Expression Synthesis to Simplify Obfuscated Code.** The third chapter presents the state of the art in VM-based obfuscation and deobfuscation techniques. Afterward, we introduce a novel method for code deobfuscation based on program synthesis. Instead of using techniques that operate on the assembly code itself, we only use the code as an oracle to generate input-output pairs. Then, we synthesize arithmetic expressions with the same input-output behavior using Monte Carlo Tree Search. Using this semantics-based approach, we simplify syntactically complex expressions as provided by obfuscation transformations based on arithmetic encodings. Furthermore, we use it to break modern VM-based obfuscators by automatically learning the instruction semantics of their arithmetic handlers.

**Chapter 4: Input Structure Synthesis to Guide Feedback-driven Fuzzing.** After analyzing the shortcomings of current fuzzers for structured languages, we design and implement an approach that automatically infers structural patterns of the target language. To learn these patterns, we use code coverage as feedback and delete such parts from inputs that are irrelevant to the observed coverage. Afterward, we make use of these learned structural patterns to implement structure-aware mutations that produce new structured inputs of the target language without prior knowledge of the underlying specification. Implementing this approach on top of a generic feedback-driven fuzzer allows us to outperform many fuzzers that make significantly stronger assumptions such as access to input specifications, seeds and source code. Although structured fuzzers with manually provided input specifications outperform our approach, we can still increase the test coverage by using their inputs as seed.

**Chapter 5: Predicate Synthesis to Automate Root Cause Explanation.** This chapter starts with an analysis of the shortcomings of current approaches for root cause analysis. Afterward, we present a bug-agnostic approach that statistically pinpoints the root cause for various types of bugs. Starting with a crashing input, we derive neighboring crashing and non-crashing inputs and collect trace information for all derived inputs. The core of this chapter is our statistical analysis that isolates crashing behavior based on Boolean predicates. Using the collected trace information that represent crashing and non-crashing behavior, we synthesize predicates that distinguish crashes from non-crashes, basically predicting whether a provided input is likely to crash the program. Following the idea that the root cause is related to the earliest predicates that predict a program crash, we rank these predicates according to their accuracy and average execution order. Our evaluation shows that this allows a human analyst to precisely pinpoint the root cause for many complex bugs, including type confusions, use-after-frees, heap buffer overflows and uninitialized variables.

## 1.3 List of Publications

This work is mainly based on three papers that were all published at the Usenix Security Symposium. In the following, we provide a high-level overview of these papers and list

the role of collaborators. Afterward, we mention the author's papers that are not part of this thesis.

**Syntia.** Chapter 3 is based on SYNTIA [43], a program synthesizer for code deobfuscation. It learns arithmetic expressions using Monte Carlo Tree Search based on input-output samples obtained from the obfuscated code. The paper was published at Usenix Security Symposium 2017 in cooperation with Moritz Contag, Cornelius Aschermann and Thorsten Holz. While I produced the idea and implementation, evaluation and writing was conducted mainly in cooperation with Moritz Contag. Cornelius Aschermann helped in the algorithmic design and the writing process.

**Grimoire.** GRIMOIRE [44] is a feedback-driven fuzzer based on REDQUEEN that automatically learns language constructs of highly structured input formats during fuzzing based on the coverage produced by different inputs. It uses these constructs to create structure-aware mutations that improve state space exploration for targets such as language interpreters and parsers. Chapter 4 is based on this paper; it was published at USENIX Security Symposium 2020 by Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner and Thorsten Holz. The main ideas of GRIMOIRE, as well as large parts of the implementation, were developed by myself. Cornelius Aschermann introduced the original idea and helped to stabilize REDQUEEN in collaboration with Sergej Schumilo. The evaluation was done by Moritz Schlögel and myself.

**Aurora.** Chapter 5 is based on a paper called AURORA [45] in which we use statistical crash analysis to isolate and explain the root cause for many different bug classes. Based on collected trace information for crashing and non-crashing inputs, it synthesizes Boolean predicates that pinpoint the root cause by describing behavioral differences in crashes and non-crashes. The paper was published at USENIX Security Symposium 2020 by Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner and Thorsten Holz. Design, writing, implementation and evaluation were realized in collaboration with Moritz Schlögel. I developed and implemented major parts of the statistical analysis and predicate synthesis algorithms. Cornelius Aschermann helped in formalizing the statistical model.

While this thesis is built upon the three aforementioned papers, I contributed to three further publications. While I was mainly involved in the evaluation and writing in *REDQUEEN: Fuzzing with Input-to-State Correspondence* [34] and *Towards Automated Discovery of Crash-Resistant Primitives in Binaries* [138], I also contributed to writing in *Towards Automated Generation of Exploitation Primitives for Web Browsers* [88].

# Chapter 2

# Synthesis of Behavioral Substitutes

In this chapter, we discuss how to learn target-specific behavioral substitutes in the context of a defined goal. For this, we introduce the concept of program synthesis, the task of automatically learning a program in a domain-specific language that satisfies a given high-level specification. We first describe different components of program synthesis. Afterward, we describe the concept of program synthesis in the context of logical reasoning: SMT-based program synthesis. Finally, we discuss the setting of program synthesis in a probabilistic world and describe its applications and inner workings for stochastic program synthesis.

## 2.1 Program Synthesis

Program synthesis is the task of automatically generating programs for a given specification. We call a program that takes a specification as input and outputs a program a *synthesizer*. The synthesizer's input and output formats are flexible and typically target a problem-specific application domain. Gulwani [102] divides program synthesis into three different dimensions: user intent, search space and search technique.

**User Intent.** The specification or *user intent* describes the semantics of the desired program behavior resulting from a successful synthesis task and will be provided as input to the synthesizer. User intent can be provided in various ways, including (1) a logical specification that describes the logical relationship between inputs and outputs of a program [104, 152, 203], (2) a finite number of input-output examples [91, 123], (3) a program execution trace representing a list of instructions describing intermediate states for a specific input [141], (4) access to a program that is queried as an oracle by the synthesizer [91, 123]. The choice of the specification is domain-specific.

**Search Space.** The space of all possible programs that can be constructed by a specific synthesizer is called *search space*. The representation of a program is domain-specific. Possible representations include bit-vector programs [104, 123], regular expressions [166], finite automata [31, 41] and context-free grammars [28]. Depending on the domain, the search space can be finite or infinite.

**Search Technique.** The *search technique* describes how a synthesizer explores the search space to construct a program. Possible search methods include (1) enumerative search that explores the search space in some order [60, 99, 154], (2) logical reasoning based on constraint solving [91, 104, 123] and (3) statistical and stochastic techniques based on probabilistic inference [103], genetic algorithms [127, 216] or Markov Chain Monte Carlo [193].

To sum up, a synthesizer takes as input a specification describing the semantics of the desired program behavior to be constructed. The search technique describes the synthesizer's algorithm to explore the search space—the set of all possible programs—and find a suitable match for the provided specification. In the following, we outline these concepts by illustrating concrete instances of search spaces and search methods. First, we discuss program synthesis in the context of logical reasoning. Afterward, we lay the groundwork for stochastic program synthesis, building the base of this thesis.

## 2.2 SMT-based Synthesis

In this section, we introduce synthesis techniques that are based on logical reasoning. We first describe the preliminaries related to logic and satisfiability so that we obtain a better understanding of SMT solvers. Afterward, we discuss several synthesis approaches that use SMT solvers as a core component for different synthesis tasks.

### 2.2.1 Preliminaries

In the next paragraphs, we introduce the required preliminaries to understand SMT solvers. While the contents are mainly based on *Decision Procedures: An Algorithmic Point of View* [140], the structure is partly inspired by Blazytko [42].

*Propositional logic* consists of a set of variables, logical symbols such as connectives ($\wedge$, $\vee$ and $\neg$) and a syntax—rules that define how to construct well-formed sentences [140]. A well-formed sentence is called *formula*. It is either a variable or a combination of parentheses, variables and the symbols $\wedge$, $\vee$ and $\neg$.

**Definition 2.1 (syntax of propositional logic, formula).** *The grammar* $(\{S\}, \Sigma = V \cup O, P, S)$ *with the of variables* $V = \{v_0, v_1, \ldots, v_n\}$*, the set of logical symbols* $O = \{\wedge, \vee, \neg\}$ *and the production rules*

$$P = \{S \rightarrow v_0 \mid v_1 \mid \ \ldots \ \mid v_n \mid S\ S\ \wedge \ \mid S\ S\ \vee \ \mid S\ \neg\}$$

*defines the syntax of propositional logic. A formula* $\varphi$ *is a well-formed sentence with respect to the syntax.*

While a formula is a syntactical construct, we can add meaning to it by assigning truth values to its variables. This way, we define the formula's *semantics*. We say a formula is *satisfiable* if it evaluates to `true`. The *interpretation* or *model* is a concrete assignment that satisfies the formula. If the formula cannot be satisfied, it is *unsatisfiable*.

**Definition 2.2 (satisfiability, interpretation, model).** *Let $\varphi$ be a formula of propositional logic. If there exists an assignment such that $\varphi$ becomes true, $\varphi$ is satisfiable, otherwise unsatisfiable. If $\varphi$ is satisfiable, there exists a concrete variable assignment that is called interpretation or model.*

It is possible to formulate satisfiability as a decision problem. For a given formula $\varphi$, the *SAT problem* asks if there exists a model for $\varphi$. A *SAT solver* can be used to decide the SAT problem.

**Definition 2.3 (SAT problem, SAT solver).** *The SAT problem is a decision problem that asks if a formula of propositional logic $\varphi$ is satisfiable. A SAT solver is a program that decides the SAT problem. If the SAT problem is satisfiable, the solver returns* `SAT` *and a model, otherwise* `UNSAT`.

While deciding the SAT problem is NP-complete, SAT solvers are known to perform very efficiently on many real-world problems [140]. However, due to their restriction to propositional logic, their usefulness for program synthesis is limited. To address this limitation, synthesizer based on logical reasoning often formulate their decision problems in satisfiability modulo theories [91, 102, 104, 123, 184, 185].

*Satisfiability modulo theories (SMT)* expand the former concepts by including additional theories. An SMT formula combines propositional logic and theories such as the theories of bit vectors and arrays [51], reals [87], strings and regular expressions [143] and binary floating-points [188]. To extend satisfiability to SMT, we ask in the *SMT problem* if there exists a model that solves a given formula $\varphi$. An *SMT solver* is a program that decides the SMT problem by combining a SAT solver with a solver for the underlying theory.

**Definition 2.4 (SMT problem, SMT solver).** *The SMT problem is a decision problem that asks if a formula of propositional logic $\varphi$ and of a theory $T$ is satisfiable. An SMT solver is a program that decides the SMT problem. If the SMT problem is satisfiable, the solver returns* `SAT` *and a model, otherwise* `UNSAT`.

Satisfiability modulo theories are at least NP-complete and, in the worst-case scenario, undecidable. However, often NP-complete fragments of theories are sufficient to solve many real-world problems effectively [74].

For program synthesis, the theory of fixed-size bit vectors is frequently used. This theory represents variables and constants as vectors of bits; it also defines arithmetic as well as bitwise operations over bit vectors (e. g., addition, subtraction, shifts and logical operations). As a consequence, it provides the basis for synthesizing all kinds of bit-vector programs.

### 2.2.2 Search Space and Methods

In this section, we discuss several techniques of SMT-based program synthesis. Therefor, we consider the following *simplified* syntax of bit-vector arithmetic which we will use to illustrate different program synthesis concepts:

**Definition 2.5 (syntax of simplified bit-vector arithmetic, formula).** *For a fixed-size bit vector of size $l$, the grammar $(\{S\}, \Sigma = V \cup O, P, S)$ with the variables $V = \{x, y\}$, the set of arithmetic symbols $O = \{+, -\}$ and the production rules*

$$P = \{S \rightarrow x \mid y \mid S\ S\ + \mid S\ S\ -\}$$

*defines the syntax of simplified bit-vector arithmetic. A formula $\varphi$ is a well-formed sentence with respect to the syntax.*

Given this simplified bit-vector arithmetic, the search space consists of all bit-vector formulas that can be derived from the grammar. Therefore, $x$, $y\ y\ +$ and $x\ y\ -$ are formulas, while $y\ \neg$ is not, since $\neg$ is not a valid symbol in the grammar. In the following, we illustrate different search techniques to explore the search base.

**Enumerative Synthesis.** The enumerative synthesis approach takes as input a logical specification that describes the desired program behavior. Then, it explores the search space via exhaustive enumeration, in which the synthesizer—beginning with the starting symbol—incrementally applies different derivation rules in some order. Often, domain-specific heuristics are used to prune the search space (e. g., enforcing an order for commutative operators). For every terminal formula, it queries an SMT solver to check if the derived formula is semantically equivalent to the provided logical specification. While this approach works well for small formulas, it grows exponentially, making this search method prohibitively expensive [36, 102].

**Component-based Synthesis.** In component-based synthesis [104, 123] a corpus of base components—core expressions of the synthesis language such as $x\ y\ +$ and $x\ y\ -$—build a so-called *library*. Given a logical specification as user intent and a logical model of control flow, an SMT solver tries to find a permutation of library components that fulfill the provided specification. As a consequence, the synthesizer can efficiently prune the search space due to the underlying SMT solver. However, such a synthesizer can only find programs if enough core components are included in the library. For instance, it can only find $x\ x\ +\ x\ x\ +\ +$ if it includes $x\ x\ +$ at least twice in the library.

**Oracle-guided Synthesis.** Instead of providing a full logical program specification as input, a partial program is provided in the form of its input-output (I/O) behavior. Therefor, a program is used as an oracle by querying it with a vector of randomly generated inputs and observing its outputs. While this approach also works when a logical specification is provided (which can be used as an oracle), it especially enables the usage of synthesis-based approaches if no concrete specification is given or too complex to be modeled. The disadvantage of this method is that the synthesized program is only as precise as the observed input-output behavior; thus, only observable behavior can be synthesized[91, 123].

**Counter-example Guided Synthesis.** To tackle the limitations of oracle-guided synthesis, counter-example guided synthesis has been introduced. If the synthesizer finds different programs leading to the same input-output behavior, it uses an SMT solver to check whether these two programs are semantically equivalent. If not, the SMT solver

returns a *distinguishing input*, which is provided to the I/O oracle. After obtaining the output, the new I/O sample is added to the partial program specification and the synthesis process restarts until the synthesizer cannot find any more distinguishing inputs [91, 104, 123].

**Template-based Synthesis.** In scenarios where domain knowledge is applicable, the search space can be reduced dramatically via template-based synthesis. In these cases, as part of user intent, a template is provided to the synthesizer as a starting point. For instance, if the synthesizer has to synthesize $x\ x\ +\ x\ x\ +\ +$ and we know that three additions are part of the semantics, we can provide a template such as $S\ S\ +\ S\ +\ +$, where the SMT solver only has to locate the variable assignment instead of exploring the whole search space [91].

To summarize, there are different search strategies on how (1) user intent can be provided, (2) the search space is structured and (3) synthesizers operate. In SMT-based approaches, the user intent is usually defined either as a complete logical specification or as a partial program in form of input-output samples. The search space generated by our example grammar is, theoretically, infinite; however, SMT-based approaches mostly operate on a large but finite search space. While the search space consists of all possible permutations of its core components, it is drastically reduced with template-based synthesis if domain knowledge can be applied. Each of these techniques can be applied on their own, but often a combination of different aspects is possible. For instance, Godefroid [91] applies a counter-example and oracle-guided approach whereby the search space is reduced via provided templates. Gulwani et al. use a component-based and counter-example guided approach [104]. As we will see in the next section, parts of these techniques can also be adapted for stochastic approaches, with and without SMT solvers.

## 2.3 Stochastic Synthesis

While SMT-based program synthesis uses logical reasoning to construct semantically correct programs, *stochastic synthesis* approximates program equivalence as part of a stochastic optimization problem, where a cost function guides the search. This also allows us to find partial programs, while SMT solvers either find a concrete solution or no solution at all. Still, the (partial) result of a stochastic synthesizer can be used in combination with an SMT solver to verify its correctness. In this section, we describe the main ideas of stochastic optimization applied to program synthesis, followed by a discussion of suitable algorithms.

### 2.3.1 Stochastic Optimization

The goal of stochastic optimization is to locate the global optimum of a discrete search space. While it is applicable to minimization as well as maximization problems, we focus on minimization. The following definitions are based on work by Hoos et al. [117] and Crama et al. [67].

For a combinatorial problem, the space of all possible combinations is called *search space* or *state space*. From this state space, an *objective function* or *cost function* maps states from the state space to real numbers.

**Definition 2.6 (search space, state, objective function).** *The search space or state space $S$ is the space of all possible combinations for a given combinatorial problem. An element $s \in S$ is referred to as a state. An objective function $f : S \to \mathbb{R}$ maps elements from $S$ to real numbers.*

Based on the objective function, we can introduce the concept of global optima. A *global optimum* is a state for which the cost function becomes minimal. To solve the *stochastic optimization problem*, we try to minimize the cost function by finding a global optimum.

**Definition 2.7 (global optimum, stochastic optimization problem).** *Given a tuple $(S, f)$, we call an element $s \in S$ a global optimum if $f(s) \leq f(s')$ for all $s' \in S$. The stochastic optimization problem describes the search for a global optimum.*

For stochastic optimization, we assume the search space to be structured so that it can be explored to find the global optimum. For a given state from the search space, slight *mutations* are assumed to generate „nearby“ states that are mapped to similar values by the objective function. For this, we introduce the abstract concept of *neighborhood* as follows:

**Definition 2.8 (neighborhood).** *Given a tuple $(S, f)$ and a state $s \in S$. We call a slightly derived state $s' \in S$ neighbor of $s$. Furthermore, we call the set of all neighbors $N(s) \subseteq S$ neighborhood of $s$.*

The naive process of locating the global optimum is to gradually explore the neighborhood and follow decreasing values obtained by the objective function. However, for many real-world problems, this approach often fails due to the inherent nature of the search space's structure. Instead, it walks towards optimums in the neighborhood. Such an optimum is referred to as *local optimum*.

**Definition 2.9 (local optimum).** *Given a tuple $(S, f)$. We call a state $s \in S$ local optimum if $f(s) \leq f(s')$ holds for all $s' \in N(s)$.*

To formulate program synthesis as a stochastic optimization problem, we consider the space of all possible programs as the search space. If two programs are syntactically similar, they are neighbors. For every program in the search space, we can use an objective function that evaluates domain-specific characteristics—based on the user intent—and assigns a real number known as *score* or *reward* to it. The closer the program relates to the user intent, the smaller is its reward. A global optimum is the desired program described by the user intent. In this setting, a synthesizer aims to minimize the objective function with respect to the user intent.

For example, consider all syntactically correct programs of simplified bit-vector arithmetic as the search space. Note that it is infinite if we assume that the synthesizer uses the grammar in Definition 2.5 to generate program candidates. Two syntactically similar programs—such as $x\ x\ +$ and $x\ y\ +$—are considered to be neighbors. Assume

that the user intent is provided as a set of input-output pairs. Then, the objective function might calculate a reward by comparing the program candidate's I/O behavior to the ones provided as specification; the synthesizer aims to find the program with the closest input-output behavior related to the specification. This global optimum might match the provided specification exactly; otherwise, the global optimum is the best partial program that approximates the specification.

### 2.3.2 Search Methods

Up until now, we discussed how to model program synthesis as a stochastic optimization problem. Given this scenario, we now introduce two different search methods for stochastic optimization: local search and Monte Carlo Tree Search.

#### 2.3.2.1 Local Search

*Local search* comprises a family of algorithms that use the neighborhood concept to explore the search space. Starting with a random state, we inspect its neighborhood by applying minor modifications to the state, evaluating it with the objective function and preferably following decreasing values. The naive approach that was sketched in the section before is known as *hill climbing* and is formally presented in Algorithm 1. It only accepts states with a lower reward until some upper loop bound $n$ is reached.

**Algorithm 1:** Basic local search algorithm. Also known as hill climbing.

```
  Result: s is a synthesized program
1 s ← random_state()
2 for i ← 0 to n do
3     s' ← mutate(s)
4     if f(s') < f(s) then
5         s ← s'
```

The main problem of hill climbing is that it quickly gets stuck in local optima without any way to escape. As a consequence, many other algorithms extend hill climbing with an *acceptance function* that probabilistically accepts states with higher scores. This concept is illustrated in Algorithm 2, in which a worse state is accepted if a random value between 0 and 1 (`RANDOM()`) is smaller than a value of the acceptance function `accept` that is based on the rewards of the current and the previous state. Still, it always remembers and returns $s_{min}$, the best state that has been found.

As a consequence, the acceptance function allows the local search to explore different locations if it is stuck in local optima. Figure 2.1 illustrates this process on the example of finding the darkest area in a given map. Starting at an initial state ($s_0$), the algorithm always accepts a candidate with a better score than the current one (the green arrows such as from $s_0$ to $s_1$ and from $s_3$ to $s_4$). If the score is worse, we accept the worse candidate with a probability (the red arrow from $s_2$ to $s_3$) that depends on the acceptance function and how much worse the candidate is. In case the candidate

**Algorithm 2:** Generic local search with an acceptance function.

**Result:** $s_{min}$ is a synthesized program

```
1 s ← random_state()
2 s_min ← s
3 for i ← 0 to n do
4     s' ← mutate(s)
5     if f(s') < f(s) or RANDOM() < accept(f(s'), f(s)) then
6         s ← s'
7         if f(s) < f(s_min) then
8             s_min ← s
```

is discarded (e. g., the crossed-out red arrow at $s_4$), we pick another one in the local neighborhood. This allows the algorithm to escape from local optima. The algorithm terminates after a specified number of $n$ iterations.

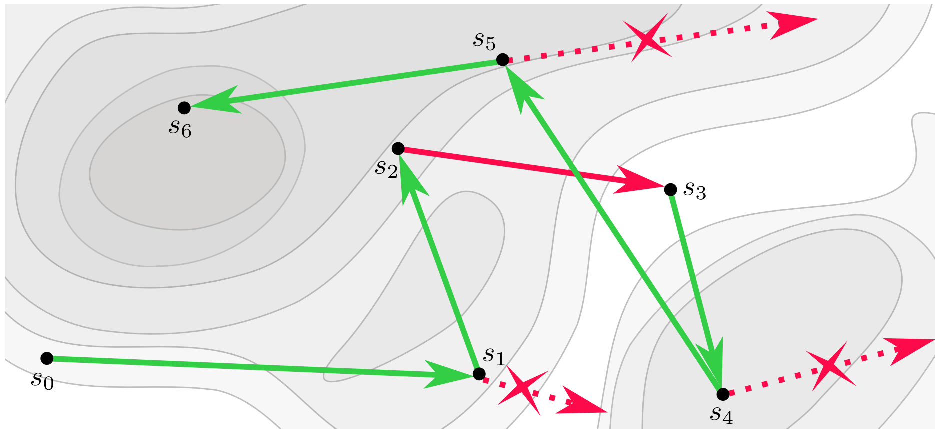


**Figure 2.1:** Local search with an acceptance function approximates a global optimum (the darkest area in the map).

In the following, we discuss two different acceptance functions: Metropolis criterion and Simulated Annealing.

**Metropolis Criterion.** The *Metropolis* criterion [156] is a common acceptance criterion. Given the current state $s'$ and the previous state $s$, it is calculated as

$$\min\left(1, \exp\left(-\frac{f(s) - f(s')}{T}\right)\right) \tag{2.1}$$

where $T$ is a control parameter referred to as *temperature*. Intuitively, the further $f(s')$ and $f(s)$ are apart from each other, the more unlikely the worse state is accepted. If $f(s') = f(s)$, the current state is always accepted. The temperature is used to increase and decrease the acceptance probability of worse states. The greater the value of $T$, the more likely worse states are accepted. $T$ may be a constant, but it also can increase or

decrease over time. A scenario in which it decreases over time is known as Simulated Annealing.

**Simulated Annealing.** *Simulated Annealing* is a variation of the Metropolis criterion in which the search is guided by a falling temperature $T$ that decreases the probability of accepting worse candidates over time [128, 133]. In Algorithm 2, this can be realized by depending the temperature on the upper bound $n$, for instance as follows:

$$T = \frac{n - i}{n} \tag{2.2}$$

The lower the loop counter, the higher is the temperature and the more likely the algorithm accepts a significantly worse candidate solution. This allows the algorithm to escape from local optima while the temperature is high; for low temperatures (loop counters closer to 0), it mainly accepts better candidate solutions.

#### 2.3.2.2 Monte Carlo Tree Search

While local search starts its stochastic exploration of the search space from a random state, there exist other algorithms that use a manually defined *initial state*—such as the start symbol of a context-free gramar—as starting point. One such algorithm is Monte Carlo Tree Search. *Monte Carlo Tree Search (MCTS)* is a stochastic, best-first tree search algorithm that directs the search towards an optimal decision without requiring much domain knowledge. The algorithm builds a search tree through reinforcement learning by performing random simulations—similar to an objective function—that estimate the quality of a node [49]. Since it follows preferably nodes with a good score, the tree grows asymmetrically. MCTS has had significant impact in artificial intelligence for computer games [82, 153, 192, 207], especially in the context of Computer Go [89, 201]. Thus, its inner workings are often explained using game-related vocabulary.

In an MCTS tree, each node represents a game state; a directed link from a parent node to its child node represents a move in the game's domain. The core algorithm iteratively builds the decision tree in four main steps that are also illustrated in Figure 2.2: (1) The *selection* step starts at the root node and successively selects the most promising child node until an *expandable* leaf (i. e., a non-terminal node that has unvisited children) is reached. (2) Afterward, one or more unvisited child nodes are added to the tree in the *expansion* step. (3) In the *simulation* step, node rewards are determined for the new nodes through random playouts. Therefor, consecutive game states are randomly derived until a terminal state (i. e., the end of the game) is reached; the game's outcome is represented by a reward. (4) Finally, the node rewards are propagated backward through the selected nodes to the root in the *backpropagation* step. The algorithm terminates if either a specified time or iteration limit is reached or if an optimal solution is found [49, 58].

Selecting the most promising child node can be treated as a so-called *multi-armed bandit problem*, in which a gambler tries to maximize the sum of rewards by choosing one out of many slot machines with an unknown probability distribution. Applied to MCTS,

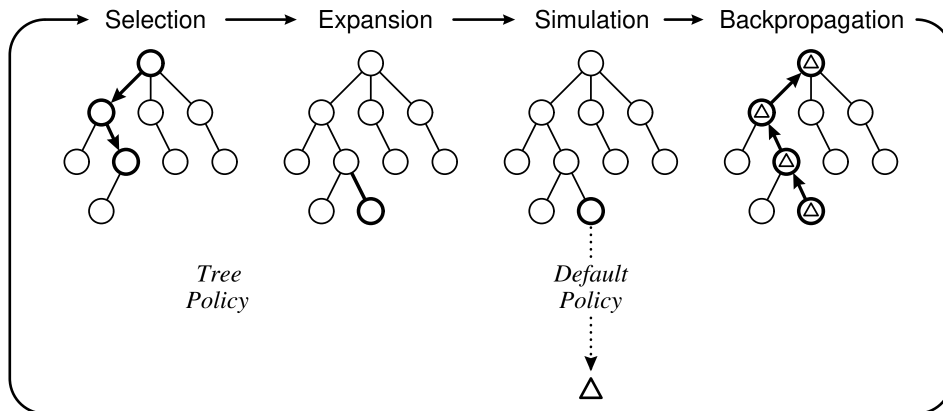


**Figure 2.2:** Illustration of a single MCTS round (taken from Browne et al. [49]).

the *Upper Confidence Bound for Trees* (UCT) [49, 89, 137] provides a good balance between exploration and exploitation. It is obtained by

$$\overline{X}_j + C\sqrt{\frac{\ln n}{n_j}}, \tag{2.3}$$

where $\overline{X}_j$ represents the average reward of the child node $j$, $n$ the current node's number of visits, $n_j$ the visits of the child node and $C$ the exploration constant. The average reward is referred to as *exploitation parameter*: if $C$ is decreased, the search is directed towards nodes with a higher reward. Increasing $C$, instead, leads to an intensified exploration of nodes with few simulations.

## 2.4 Program Synthesis in the Context of This Work

After discussing different scenarios for SMT-based and stochastic program synthesis, we now give a high-level overview of how we synthesize behavioral substitutes in the following chapters. In general, a behavioral substitute refers to the output of a synthesizer that is used to reason about program behavior in various contexts.

In Chapter 3, we simplify obfuscated code via program synthesis by learning bit-vector formulas with the same input-output behavior. In an oracle-guided approach, we use Monte Carlo Tree Search to generate bit-vector programs based on a context-free grammar and guide the search via an objective function that uses different distance metrics for bit-vectors.

In Chapter 4, we synthesize the structure of inputs and guide a fuzzer using structure-aware mutations to find vulnerabilities in language interpreters and parsers. In this scenario, we use a feedback-driven fuzzer as oracle. Feedback in the form of triggered code coverage allows us to extract structural patterns of the target language by removing parts from inputs that are irrelevant for the observed coverage.

In Chapter 5, we synthesize Boolean predicates that isolate crashing behavior from a set of crashing and non-crashing inputs. These predicates are used to pinpoint the root cause of a crash. To generate them, we use a template-based synthesis approach to learn different predicate types such as control-flow or register predicates. Therefor, we use a statistical approach to locate the best predicates that distinguish crashes from non-crashes in a finite set of trace information representing crashing and non-crashing program behavior.

# Chapter 3

# Expression Synthesis to Simplify Obfuscated Code

## 3.1 Introduction

*Code obfuscation* describes the process of applying an obfuscating transformation to an input program to obtain an obfuscated copy of the program. Said copy should be more complex than the input program such that an analyst cannot easily reason about it. An obfuscating transformation is further desired to be *semantics-preserving*, i. e., it must not change observable program behavior [64]. Code obfuscation can be leveraged in many application domains, for example in software protection solutions to prevent illegal copies, or in malicious software to impede the analysis process. In practice, different kinds of obfuscation techniques are used to hinder the analysis process. Most notably, industry-grade obfuscation solutions are typically based on *Virtual Machine (VM)*-based transformations [168, 202, 208, 213], which are considered one of the strongest obfuscating transformations available [35]. While these protections are not perfect and in fact are broken regularly, attacking them is still a time-consuming task that requires highly specific domain knowledge of the individual Virtual Machine implementation. Consequently, for example, this gives game publishers a head-start in which enough revenue can be generated to stay profitable. On the other hand, obfuscated malware stays under the radar for a longer time, until concrete analysis results can be used to effectively defend against it.

To deal with this problem, prior research has explored many different approaches to enable deobfuscation of obfuscated code. For example, Rolles proposes static analysis to aid in deobfuscation of VM-based obfuscation schemes [183]. However, it incorporates specific implementation details an attacker has to know a priori. Further, static analysis of obfuscated code is notoriously known to be intractable in the general case [64]. Hence, recent deobfuscation proposals have shifted more towards dynamic analysis [66, 224, 225]. Commonly, they produce an execution trace and use techniques such as (dynamic) taint analysis or symbolic execution to distinguish input-dependent instructions. Based on their results, the program code can be reduced to only include relevant, input-

dependent instructions. This effectively strips the obfuscation layer. Even though such deobfuscation approaches sound promising, recent work proposes several ways to effectively thwart underlying techniques, such as symbolic execution [35]. For this reason, it suggests itself to explore distinct techniques that may be leveraged for code deobfuscation.

In this chapter, we propose an approach orthogonal to prior work on approximating the underlying semantics of obfuscated code. Instead of manually analyzing the instruction handlers used in virtualization-based (VM) obfuscation schemes in a complex and tedious manner [183] or learning merely the bytecode decoding (not the semantics) of these instruction handlers [199], we aim at learning the *semantics* of VM-based instruction handlers in an automated way. Furthermore, our goal is to develop a generic framework that can deal with different use cases. Naturally, this includes constructs close to obfuscation, such as Mixed Boolean-Arithmetic (MBA), different kinds of VM-based obfuscation schemes, or even analysis of code chunks (so called *gadgets*) used in Return-oriented Programming (ROP) exploits.

To this extend, we explore how *program synthesis* can be leveraged to tackle this problem. Broadly speaking, program synthesis describes the task of automatically constructing programs for a given specification. While there exists a variety of program synthesis approaches [102], we focus on SMT-based and stochastic program synthesis in the following, given its proven applicability to problem domains close to trace simplification and deobfuscation. SMT-based program synthesis constructs a loop-free program based on first-order logic constraints whose satisfiability is checked by an SMT solver. For *component-based* synthesis, components are described that build the instruction set of a synthesized program; for instance, components may be bitwise addition or arithmetic shifts. The characteristics of a well-formed program such as the interconnectivity of components are defined and the semantics of the program are described as a logical formula. Then, an SMT solver returns a permutation of the components that forms a well-encoded program following the previously specified intent [104, 123], if it is satisfiable, i. e., such a permutation *does* exist.

Instead of relying on a logical specification of program intent, *oracle-guided* program synthesis uses an input-output (I/O) oracle. Given the outputs of an I/O oracle for arbitrary program inputs, program synthesis learns the oracle's semantics based on a finite set of I/O samples. The oracle is iteratively queried with *distinguishing* inputs that are provided by an SMT solver. Locating distinguishing inputs is the most expensive task in this approach. The resulting synthesized program has the same input-output behavior as the I/O oracle [123]. Contrary to SMT-based approaches that only construct semantically correct programs, stochastic synthesis *approximates* program equivalence and thus remains faster. In addition, it can also find partial correct programs. Program synthesis is modeled as heuristic optimization problem, where the search is guided by a cost function. It determines, for instance, output similarity of the synthesized expression and the I/O oracle for same inputs [193].

As program synthesis is indifferent to code complexity, it can synthesize arbitrarily obfuscated code and is only limited by the underlying code's *semantic* complexity. We demonstrate that a stochastic program synthesis algorithm based on Monte Carlo

Tree Search (MCTS) achieves this in a scalable manner. To show feasibility of our approach, we automatically learned the semantics of 489 out of 500 MBA-obfuscated random expressions. Furthermore, we synthesize the semantics of arithmetic instruction handlers in two state-of-the art commercial virtualization-based obfuscators with a success rate of more than 94%. Finally, to show applicability to areas more focused on security aspects, we further automatically learn the semantics of ROP gadgets.

**Contributions** In summary, we make the following contributions:

- We introduce a generic approach for trace simplification based on program synthesis to obtain the semantics of different kinds of obfuscated code. We demonstrate how Monte Carlo Tree Search (MCTS) can be utilized in program synthesis to achieve a scalable and generic approach.
- We implement a prototype of our method in a tool called SYNTIA. Based on I/O samples from assembly code as input, SYNTIA can apply MCTS-based program synthesis to compute a simplified expression that represents a deobfuscated version of the input.
- We demonstrate that SYNTIA can be applied in several different application domains such as simplifying MBA expressions by learning their semantics, learning the semantics of arithmetic VM instruction handlers and synthesizing the semantics of ROP gadgets.

## 3.2 Challenges in Code Deobfuscation

Before presenting our approach to utilize program synthesis for recovering the semantics of obfuscated code, we first review several concepts and techniques we use throughout the rest of the chapter.

### 3.2.1 Obfuscation

In the following, we discuss several techniques that qualify as an obfuscating transformation, namely virtualization-based obfuscation, Return-oriented Programming and Mixed Boolean-Arithmetic.

#### 3.2.1.1 Virtualization-based Obfuscation

Contemporary software protection solutions such as VMProtect [213], Themida [168], and major game copy protections such as SecuROM base their security on the concept of *Virtual Machine-based* obfuscation (also known as *virtualization-based* obfuscation [183]).

Similar to system-level Virtual Machines (VMs) that emulate a whole system platform, process-level VMs emulate a foreign instruction set architecture (ISA). The core idea is to translate parts of a program, e. g., a function $f$ containing intellectual property, from its native architecture—say, Intel x86—into a custom VM-ISA. The obfuscator then embeds both the *bytecode* of the virtualized function (its instructions encoded

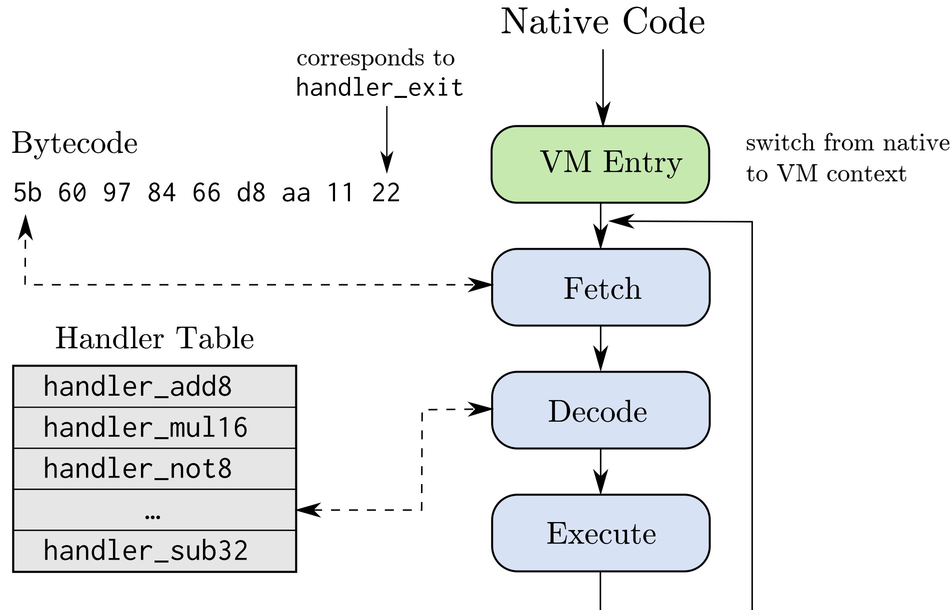


**Figure 3.1:** The Fetch–Decode–Execute cycle of a Virtual Machine. Native code calls into the VM, upon which startup code is executed (VM entry). It performs the context switch from native to VM context. Then, the next instruction is fetched from the bytecode stream, mapped to the corresponding handler using the handler table (decoding) and, finally, the handler is executed. The process repeats for subsequent VM instructions in the bytecode until the exit handler is executed, which returns back to native code.

for the VM-ISA) along with an *interpreter* for the new architecture into the target binary whilst removing the function's original, native code. Every call to $f$ is then replaced with an invocation of the interpreter. This effectively thwarts any naive reverse engineering tool operating on the native instruction set and forces an adversary to analyze the interpreter and re-translate the interpreted bytecode back into native instructions. Commonly, the interpreter is heavily obfuscated itself. As VM-ISAs can be arbitrarily complex and generated uniquely upon protection time, this process is highly time-consuming [183].

**Components.** The *(VM) context* holds internal variables of the VM-ISA such as general-purpose registers or the virtual instruction pointer. It is initialized by sequence called *VM entry*, which handles the context switch from native code to bytecode.

After initialization, the *VM dispatcher* fetches and decodes the next instruction and invokes the corresponding *handler* function by looking it up in a global *handler table* (depicted in Figure 3.1). The latter maps indices, obtained from the instruction's bytecode in the decoding step, to handlers addresses. In its most simple implementation, all handler functions return to a central dispatching loop which then dispatches the next handler. Eventually, execution flow reaches a designated handler, *VM exit*, which performs the context switch back to the native processor context and transfers control back to native code.

**Custom ISA.** The design of the target VM-ISA is entirely up to the VM designer. Still, to maximize the amount of handlers an analyst has to reverse engineer, VMs often opt for reduced complexity for the individual handlers, akin to the RISC design principle. To exemplify, consider the following Intel x86 code:

```
mov eax, dword ptr [0x401000 + ebx * 4]
pop dword ptr [eax]
```

This might get translated into VM-ISA as follows:

```
vm_mov   T0, vm_context.real_ebx
vm_mov   T1, 4
vm_mul   T2, T0, T1
vm_mov   T3, 0x401000
vm_add   T4, T2, T3
vm_load  T5, dword(T4)
vm_mov   vm_context.real_eax, T5
vm_mov   T6, T5
vm_mov   T7, vm_context.real_esp
vm_add   T8, T7, T1
vm_mov   vm_context.real_esp, T8
vm_load  T9, dword(T7)
vm_store dword(T6), T9
```

It favors many small, simple handlers over fewer more complicated ones.

**Bytecode Blinding.** In order to prevent global analysis of instructions, the bytecode *bc* of each VM instruction is blinded based on its instruction type, i. e., its corresponding handler $h$, at protection time. Likewise, each handler *unblinds* the bytecode before decoding its operands: $(bc, vm_key) \leftarrow \mathsf{unblind}_h(blinded_bc, vm_key)$.

The routine is parameterized for each handler $h$ and updates a global key register in the VM context. Consequently, instruction decoding can be *flow-sensitive*: An adversary is unable to patch a single VM instruction without re-blinding all subsequent instructions. This, in turn, requires her to extract the unblinding routines from every handler involved. The individual unblinding routines commonly consist of a combination of arithmetic and logical operations.

**Handler Duplication.** In order to easily increase analysis complexity, common VMs *duplicate* handlers such that the same virtual instruction can be dispatched by multiple handlers. In presence of bytecode blinding, these handlers' semantics only differ in the way they unblind the bytecode, but perform the same operation on the VM context.

**Architectures.** In his paper about interpretation techniques, Klint denotes the aforementioned concept using a central decoding loop as the "classical interpretation method" [135]. An alternative is proposed by Bell with *Threaded Code* (TC) [39]: He suggests inlining the dispatcher routine into the individual handler functions such

that handlers execute in a chained manner, instead of returning to a central dispatcher. Nevertheless, the dispatcher still indexes a global handler table.

In Klint's paper, however, he describes an extension of TC, *Direct Threaded Code* (DTC). As in the TC approach, the dispatcher is appended to each handler. The handler table, though, is inlined into the *bytecode* of the instruction. Each instruction now directly specifies the address of its corresponding handler. This way, in presence of bytecode blinding, not all handler addresses are exposed immediately, but only those used on a certain path in the bytecode.

**Attacks.** Several academic works have been published that propose novel attacks on virtualization-based obfuscators [66, 183]. Section 3.6.3 discusses and classifies them. In addition, it draws a comparison to our approach.

#### 3.2.1.2 Return-oriented Programming

In *Return-oriented Programming* (ROP) [139, 198], shellcode is expressed as a so-called ROP chain, a list of references to *gadgets* and parameters for those. In the preliminary step of an attack, the adversary makes `esp` point to the start of the chain, effectively triggering the chain upon function return. Gadgets are small, general instruction sequences ending on a `ret` instruction; other flavors propose equivalent instructions. Concrete values are taken from the ROP chain on the stack. As an example, consider the gadget `pop eax; ret`: It takes the value on top of the stack, places it in `eax` and, using the `ret` instruction, dispatches the next gadget in the chain. By placing an arbitrary immediate value `imm32` next to this gadget's address in the chain, an attacker effectively encodes the instruction `mov eax, imm32` in her ROP shellcode. Depending on the gadget space available to the attacker, this technique allows for arbitrary computations [170, 196].

Automated analysis of ROP exploits is a desirable goal. However, its unique structure poses various challenges compared to traditional shellcode detection. In their paper, Graziano et al. outline them and propose an analysis framework for code-reuse attacks [100]. Amongst others, they mention challenges such as verbosity of the gadgets, stack-based chaining, lack of immediates, and the distinction of function calls and regular control flow. Further, they stress how an accurate emulation of gadgets is important for addressing these challenges. Considering the aforementioned challenges, at its core, Return-oriented Programming can be seen as an albeit weaker flavor of obfuscated code. In particular, the chained invocation of gadgets is reminiscent of handlers in VM-based obfuscation schemes following the threaded code principle.

In addition to its application to exploitation, ROP has seen other fields of applications such as rootkit development [214], software watermarking [151], steganography [150], and code integrity verification [30], which reinforces the importance of automatic ROP chain analysis.

#### 3.2.1.3 Mixed Boolean-Arithmetic

Zhou et al. propose transformations over Boolean-arithmetic algebras to hide constants by turning them into more complex, but semantically equivalent expressions, so called MBA expressions [80, 236]. In Section 3.6.2, we provide details on their proposal of MBA expressions and show how our approach is still able to simplify them.

### 3.2.2 Trace Simplification

Due to the complexity of static analysis of obfuscated code, many deobfuscation approaches proposed recently make use of dynamic analysis [66, 100, 100, 199, 225]. Notably, they operate on *execution traces* that record instruction addresses and accompanying metadata, e. g., register content, along a concrete execution path of a program. Subsequently, trace simplification is performed to strip the obfuscation layer and simplify the underlying code. Depending on the approach, multiple traces are used for simplification or one single trace is reduced independently.

Coogan et al. [66] propose *value-based dependence* analysis of a trace in order to track the flow of values into system calls using an equational reasoning system. This allows them to reduce the trace to those instructions *relevant* to the previously mentioned value flow.

Graziano et al. [100] mainly apply standard compiler transformations such as dead code elimination or arithmetic simplifications to reduce the trace.

Yadegari et al. [225] use bit-level taint analysis to identify instructions relevant to the computation of outputs. For subsequent simplification, they introduce the notion of *quasi-invariant* locations with respect to an execution. These are locations that hold the same value at every use in the trace and can be considered constants when performing constant propagation. Similarly, they use several other compiler optimizations and adapt them to make use of information about *quasi-invariance* to prevent over-simplification.

## 3.3 Design

Given an instruction trace, we dissect the instruction trace into *trace windows* (i. e., subtraces) and aim at learning their high-level semantics which can be used later on for further analysis. In the following, we describe our approach which is divided into three distinct parts:

1. *Trace Dissection.* The instruction trace is partitioned into unique sequences of assembly instructions in a (semi-)automated manner.
2. *Random Sampling.* We derive random input-output pairs for each trace window. These pairs describe the trace window's semantics.
3. *Program Synthesis.* Expressions that map all provided inputs to their corresponding outputs are synthesized.

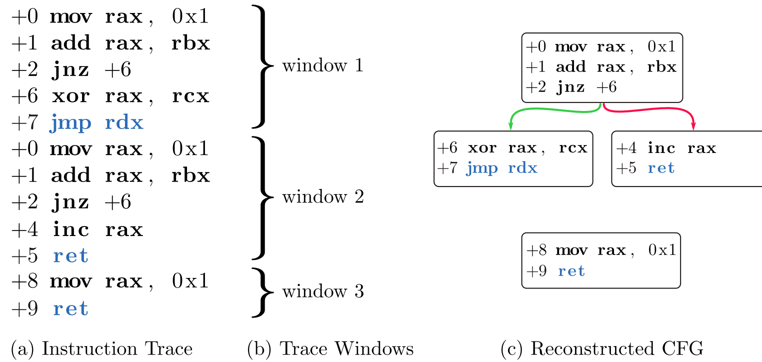


**Figure 3.2:** Dissecting a given trace (a) into several trace windows (b). The trace windows can be used to reconstruct a (possibly disconnected) control-flow graph (c).

### 3.3.1 Trace Dissection

The choice of trace window boundaries highly impacts later analysis stages. Most notably, it affects synthesis results: if a trace window ends at an *intermediary* computation step, the synthesized formula is not necessarily succinct or meaningful at all, as it includes spurious semantics.

Yet, we note how trace dissection of ROP chains and VM handlers lends itself to a simple heuristic. Namely, we split traces at indirect branches. In the ROP case, this describes the transition between two gadgets (commonly, on a `ret` instruction), whereas for VM handlers it distinguishes the invocation of the next handler (cf. Section 3.6.3). Figure 3.2 illustrates the approach. Given concrete trace window boundaries, we can reconstruct a control-flow graph consisting of multiple connected components. A *trace window* then describes a particular path through a connected component.

### 3.3.2 Random Sampling

The goal of random sampling is to derive input-output relations that describe the semantics of a trace window. This happens in two steps: First, we determine the inputs and outputs of the trace window. Then, we replace the inputs with random values and obverse the outputs.

Generally speaking, we consider register and memory reads as inputs and register and memory writes as outputs. For inputs, we apply a read-before-write principle: inputs are only registers/memory locations that are read before they have been written; for outputs, we consider the last writes of a register/memory location as output.

```
mov rax, [rbp + 0x8]
add rax, rcx
mov [rbp + 0x8], rax
add [rbp + 0x8], rdx
```

Following this principle, the code above has three inputs and two outputs: The inputs are the memory read $M_0$ in line 1, `rcx` (line 2) and `rdx` (line 4). The two outputs are $o_0$ (line 2) and $o_1$ (line 4).

In the next step, we generate random values and obverse the I/O relationship. For instance, we obtain the outputs $(7, 14)$ for the input tuple $(2, 5, 7)$; for the inputs $(1, 7, 10)$, we obtain $(8, 18)$.

By default, we use register locations as well as memory locations as inputs and outputs. However, we support the option to reduce the inputs and outputs to either register or memory locations. For instance, if we know that registers are only used for intermediate results, we may ignore them since it reduces the complexity for the synthesis.

### 3.3.3 Synthesis

This section demonstrates how we synthesize the semantics of assembly code; we discuss the inner workings of our synthesis approach in the next section.

After we obtained the I/O samples, we combine the different samples and synthesize each output separately. These synthesis instances are mutually independent and can be completely parallelized.

To exemplify, for the I/O pairs above, we search an expression that transforms $(2, 5, 7)$ to 7 and $(1, 7, 10)$ to 8 for $o_0$; for $o_1$, the expression has to map $(2, 5, 7)$ to 14 and $(1, 7, 10)$ to 18. Then, the synthesizer finds $o_0 = M_0 + \texttt{rcx}$ and $o_1 = M_0 + \texttt{rcx} + \texttt{rdx}$.

## 3.4 Program Synthesis

In the last section, we demonstrated how we obtain I/O samples from assembly code and apply program synthesis to that context. This section describes our algorithm in detail; we show how we find an expression that maps all inputs to their corresponding outputs for all observed samples. We use Monte Carlo Tree Search, since it has been proven to be very effective when working on infinite decision trees without requiring much domain knowledge.

We consider program synthesis as a single-player game whose purpose is to synthesize an expression whose input-output behavior is as close as possible to given I/O samples. In essence, we define a context-free grammar that consists of terminal and non-terminal symbols. (Partially) derived words of the grammar are *game states*; the grammar's production rules represent the *moves* of the game. *Terminal nodes* are expressions that contain only terminal symbols; these are *end states* of the game.

Given a maximum number of iterations and I/O samples, we iteratively apply the four MCTS steps (cf. Section 2.3.2.2), until we find a solution or we reach the timeout. Starting with a non-terminal expression as *root node*, we *select* the most-promising expandable node. A node is *expandable*, if there still exist production rules that have not been applied to this node. We choose a production rule randomly and expand the selected node. To evaluate the quality of the new node, we perform a *random playout*: First, we randomly derive a terminal expression by successively applying random production rules. Then, we evaluate the expressions based on the inputs from the I/O pairs and compare the output similarity. The similarity score is the node *reward*.

A reward of 1 ends the synthesis, since the input-output behavior is the same for the provided samples. Finally, we *propagate* the reward back to the root.

In the following, we give details on node selection, our grammar, random playouts and backpropagation. Finally, we discuss the algorithm configuration and parameter tuning. To demonstrate the different steps of our approach, we use the following running example throughout this section:

**Example 3.1 (I/O relationship).** *Working with bit-vectors of size* 3 *(i. e., modulo* $2^3$*), we observe for an expression with two inputs and one output the I/O relations:* $(2, 2) \rightarrow 4$ *and* $(4, 5) \rightarrow 1$*. A synthesized expression that maps the inputs to the corresponding outputs is* $f(a, b) = a + b$*.*

### 3.4.1 Node Selection

Since we have an infinite search space for program synthesis, node selection must be a trade-off between exploration and exploitation. The algorithm has to explore different nodes such that several promising and non-promising candidates are known. On the other hand, it has to follow more promising candidates to find deeper expressions. As described in Section 2.3.2.2, the UCT (cf. Equation 2.3) provides a good balance between exploitation and exploration for many MCTS applications.

However, we observed that it does not work for our use case: if we set the exploration constant $C$ to a higher value (focus on exploration), it does not find deeper expressions; if we set $C$ to a lower value, MCTS gets lost in deep expressions. To solve this problem, we use an adaption of UCT that is known as *Simulated Annealing UCT* (SA-UCT) [187]. The main idea of SA-UCT is to use the characteristics of Simulated Annealing (cf. Section 2.3.2.1) and apply it to UCT. SA-UCT is obtained by replacing the exploration constant $C$ by a variable $T$ with

$$T = C\frac{N - i}{N}, \tag{3.1}$$

where $N$ is the maximum number of MCTS iterations and $i$ the current iteration. Then, SA-UCT is defined as

$$\overline{X}_j + T\sqrt{\frac{\ln n}{n_j}}. \tag{3.2}$$

$T$ decreases over time, since $\frac{N-i}{N}$ converges to 0 for increasing values of $i$. As a result, MCTS places the emphasis on exploration in the beginning; the more $T$ decreases, the more the focus shifts to exploitation.

### 3.4.2 Grammar

Game states are represented by sentential forms of a context-free grammar that describes valid expressions of our high-level abstraction. We introduce a terminal symbol for

each input (which corresponds to a variable that stores this input) and each valid operator (e. g., addition or multiplication). For every data type that can be computed we introduce one non-terminal symbol (in our running example, we only use a single non-terminal value $U$ that represents an unsigned integer). The production rules describe how we can derive expressions in our high-level description. Since the sentential forms represent partial expressions, we will use the term expression to denote the (partial) expression that is represented by a given sentential form. Sentences of the grammar are final states in the game since they do not allow any further moves (derivations). They represent expressions that can be evaluated. We represent expressions in *Reverse Polish Notation* (RPN).

**Example 3.2.** *The grammar in our previous example has two input symbols $V = \{a, b\}$, since each I/O sample has two inputs. If the grammar supports addition and multiplication $O = \{+, *\}$, there are four production rules: $R = \{U \to U\ U\ +\ |\ U\ U\ *\ |\ a\ |\ b\}$. An unsigned integer expression $U$ can be mapped to an addition or multiplication of two such expressions or a variable. The final grammar is $(\{U\}, \Sigma = V \cup O, R, U)$.*

**Synthesis Grammar.**

Our grammar is designed to synthesize expressions that represent the semantics of bit-vector arithmetic, especially for the x86 architecture. For every data type ($U_8$, $U_{16}$, $U_{32}$ and $U_{64}$), we define the set of operations as $O = \{+,-,*,/_s,/,\%_s,\%,\wedge,\vee,\oplus,\ll,\gg,\gg_a,-_1,\neg,$ `sign_ext`, `zero_ext`, `extract`, $++$, $1\}$, where the operations are binary addition, subtraction, multiplication, signed/unsigned division, signed/unsigned remainder, bitwise and/or/xor, logical left shift, logical/arithmetic right shift as well as unary minus and complement. The unary operations `sign_ext` and `zero_ext` extend smaller data types to signed/unsigned larger data types. Conversely, the unary operator `extract` transforms larger data types into smaller data types by extracting the respective least significant bits. Since the x86 architecture allows register concatenation (e. g., for division), we employ the binary operator $++$ to concatenate two expressions of the same data type. Finally, to synthesize expressions such as increment and decrement, we use the constant 1 as niladic operator. The input set $V$ consists of $|V| = n$ variables, where $n$ represents the number of inputs.

**Tree Structure.** The sentential form $U$ is the root node of the MCTS tree. Its child nodes are other expressions that are produced by applying the production rules to a single non-terminal symbol of the parent. The expression depth (referred to as *layer*) is equivalent to the number of derivation steps, as depicted in Figure 3.3.

**Example 3.3.** *The root node $U$ is an expression of layer 0. Its children are $a$, $b$, $U\ U\ +$, and $U\ U\ *$, where $a$ and $b$ are terminal expressions of layer 1. Assuming that the right-most $U$ in an expression is replaced, the children of $U\ U\ +$ are $U\ b\ +$, $U\ a\ +$, $U\ U\ U\ +\ +$, and $U\ U\ U\ *\ +$. To obtain the layer 3 expression $b\ a\ +$, the following derivation steps are applied: $U \Rightarrow U\ U\ + \Rightarrow U\ a\ + \Rightarrow b\ a\ +$.*

To direct the search towards outer expressions, we replace the *top-most-right-most* non-terminal. If we, instead, substitute always the *right-most* non-terminal only, then the search would be guided towards most-promising subexpressions. If the expression is too nested, the synthesizer would find the partial subexpression but not the whole

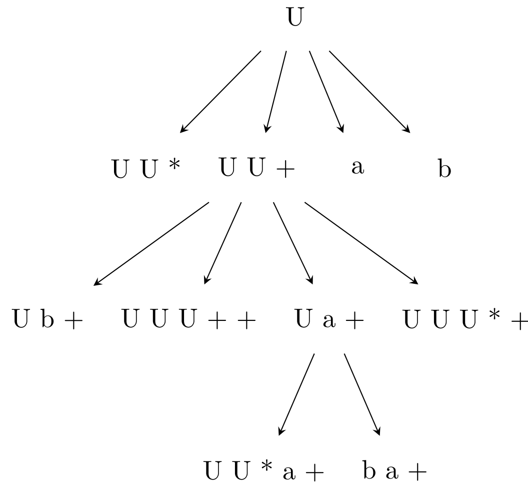


**Figure 3.3:** An MCTS tree for program synthesis that grows towards the most-promising node $b\ a\ +$, the right-most leaf in layer 3.

expression. The top-most-right-most derivation is illustrated in Figure 3.4, which shows the abstract syntax tree (AST) of an expression.

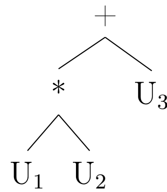


**Figure 3.4:** The left-most $U$ in $U_3\ U_2\ U_1\ *\ +$ is the top-most-right-most non-terminal in the abstract syntax tree. (The indices are provided for illustrative purposes only.)

**Example 3.4.** *The expression* $(U + (U * U))$ *is represented as* $U\ U\ U\ *\ +$*. If we successively replace the right-most* $U$*, the algorithm is unlikely to find expressions such as* $((a+b)+(b*(b*a)))$*, since it is directed into the subexpression with the multiplication. Instead, replacing the top-most-right-most non-terminal directs the search to the top-most addition and then explores the subexpressions.*

### 3.4.3 Random Playout

One of the key concepts of MCTS are random playouts. They are used to determine the outcome of a node; this outcome is represented by a reward. In the first step, we randomly apply production rules to the current node, until we obtain a terminal expression. To avoid infinite derivations, we set a maximum playout depth. This maximum playout depth defines how often a non-terminal symbol can be mapped to rules that contain non-terminal symbols; at the latest we reached the maximum, we

map non-terminals only to terminal expressions. This happens in a top-most-right-most manner. Afterward, we evaluate the expression for all inputs from the I/O samples.

**Example 3.5.** *Assuming a maximum playout depth of* $2$ *and the expression* $U\ U\ *$*, the first top-most-right-most* $U$ *is randomly substituted with* $U\ U\ *$*, the second one with* $U\ U\ +$*. After that, the remaining non-terminal symbols are randomly replaced with variables:* $U\ U\ * \Rightarrow U\ U\ U\ *\ * \Rightarrow U\ U\ +\ U\ U\ *\ * \Rightarrow \cdots \Rightarrow a\ a\ +\ b\ a\ *\ *$*. A random playout for* $U\ U\ +$ *is* $a\ b\ b\ +\ +$*.*

*For the I/O sample* $(2, 2) \rightarrow 4$*, we evaluate* $g(2, 2) = 0$ *for* $g(a, b) = ((a + a) * (b * a)) \mod (2^8)$ *and* $h(2, 2) = 6$ *for* $h(a, b) = (a + (b + b)) \mod 2^8$*.*

We set terminal nodes to inactive after their evaluation, since they already are end states of the game; there is no possibility to improve the node's reward by random playouts. As a result, MCTS will not take these nodes into account in further iterations. The node's reward is the similarity of the evaluated expressions and the outputs from the I/O samples. We describe in the following section how to measure the similarity to the outputs.

### 3.4.4 Measuring Similarity of Outputs

To measure the similarity of two outputs, we compare values with different metrics: arithmetic distance, Hamming distance, count leading zeros, count trailing zeros, count leading ones and count trailing ones. While the numeric distance is a reliable metric for arithmetic operations, it does not work well with overflows and bitwise operations (e. g., xor and shifts). In turn, the Hamming distance addresses these operations since it states in how many bits two values differ. Finally, the leading/trailing zeros/ones are strong indicators that two values are in the same range. We scale each result between a value of 0 and 1. Since the different metrics compensate each other, we set the total similarity reward to the average reward of all metrics.

**Example 3.6.** *Considering I/O pair* $(2, 2) \rightarrow 4$*, the output similarities for* $g$ *and* $h$ *(as defined in Example 3.5) are* $\text{similarity}(4, 0)$ *and* $\text{similarity}(4, 6)$*. Limiting to the metrics of Hamming distance and count leading zeros (`clz`), we obtain* $\text{hamming}(4, 0) = \text{hamming}(4, 6) = 0.67$*,* $\text{clz}(4, 0) = 0$ *and* $\text{clz}(4, 6) = 1.0$*. Therefore, the average similarities are* $\text{similarity}(4, 0) = 0.335$ *and* $\text{similarity}(4, 6) = 0.835$*. Related to the random playouts, the evaluated node* $U\ U\ +$ *has a higher reward than* $U\ U\ *$*.*

During a random playout, we calculate the similarity for all I/O samples. The final node reward is the average score of all similarity rewards. A reward of 1 finishes program synthesis, since the evaluated expression produces exactly the outputs from the I/O samples.

### 3.4.5 Backpropagation

After obtaining a score by random playout, we do the following for the selected node and all its parents, up to the root: (1) We update the node's average reward. This reward is averaged based on the node's and its successors' total number of random

playouts. (2) If the node is fully expanded and its children are all inactive, we set the node to inactive. (3) Finally, we set the current node to its parent node.

### 3.4.6 Expression Simplification

Since MCTS performs a stochastic search, synthesized expressions are not necessary in their shortest form. Therefore, we apply some basic standard expression simplification rules. For example, if the synthesizer constructs integer values as $((1 \ll 1) \ll (1 + (1 \ll 1)))$, we can reduce them to the value 16.

### 3.4.7 Algorithm Configuration

Two main factors define the algorithm's success that cannot be influenced by the user: the number of input variables and the complexity (e. g., depth) of the expression to synthesize. Contrary, there exist four parameters that can be configured by a user to improve the effectiveness and speed: the initial SA-UCT value, the number of I/O samples, the maximum number of MCTS iterations and the maximum playout depth.

The SA-UCT parameter $T$ configures the trade-off between exploration and exploitation and depends on the maximum number of MCTS iterations; if the maximum number of MCTS iterations is low, the algorithm focuses on exploiting promising candidates within a small period of time. The same holds for small initial values of $T$.

A large number of variables or a higher expression depth requires more MCTS iterations. Besides the maximum number of MCTS iterations, the maximum playout depth provides more accuracy since it is more probable to hit deeper expressions or more influencing variables with deeper playouts. On the other hands, deeper playouts have an impact on the execution time.

Since random playouts are performed for every node and for every I/O pair, the number of I/O samples has a significant impact on the execution time. In addition, it effects the number of false positives, since there are less expressions that have the same I/O behavior for a larger number of I/O samples. Finally, the MCTS synthesis is more precise since the different node rewards are expected to be informative.

Since the search space for finding good algorithm configurations for different complexity classes is large, we approximate an optimal solution by Simulated Annealing. We present the details and results in Section 3.6.1.

## 3.5 Implementation

We implemented a prototype implementation of our approach in our tool Syntia, which is written in Python. For trace generation and random sampling, we use the *Unicorn Engine* [177], a CPU emulator framework. To analyze assembly code (e. g., trace dissection), we utilize the disassembler framework *Capstone* [178]. Furthermore, we use the SMT solver *Z3* [158] for expression simplification.

Initially, Syntia expects a memory dump, a start and an end address as input. Then, it emulates the program and outputs the instruction trace. Then, the user has the opportunity to define its own rules for trace dissection; otherwise, Syntia dissects the trace at indirect control transfers. Additionally, the user has to decide if register and/or memory locations are used as inputs/outputs and how many I/O pairs shall be sampled. Syntia traces register and memory modifications in each trace window, derives the inputs and outputs and generates I/O pairs by random sampling. The last step can be parallelized for each trace window. Finally, the user defines the synthesis parameters. Syntia creates a synthesis tasks for each (trace window, output) pair. The synthesis tasks are performed in parallel. The synthesis results are simplified by Z3's term-rewriting engine.

## 3.6 Experimental Evaluation

In the following, we evaluate our approach in three areas of application. The experiments have been evaluated on a machine with two Intel Xeon E5-2667 CPUs (in total, 12 cores and 24 threads) and 96 GiB of memory. However, we never have used more than 32 GiB of memory even though parallel I/O sampling for many trace windows can be memory intensive; synthesis itself never used more than 6 GiB of memory.

### 3.6.1 Parameter Choice

As described in Section 3.4.7, we approximate an optimal algorithm configuration with Simulated Annealing. To compute preferably representative results, we generate a set of $1,200$ *randomly* generated expressions. We divide this set into three classes with 400 expressions each; to prevent overfitting the parameters on a fixed set of inputs, the experiments of each class are performed with distinct input samples.

In each iteration, Simulated Annealing synthesizes the $1,200$ expressions under the same configuration. We set a timeout of 120 seconds for each synthesis task and prune non-successful tasks by a constant factor of the timeout. As a result, Simulated Annealing optimizes towards a high success rate for synthesis tasks and a minimal average time. Table 3.1 lists the initial algorithm configuration and the parameter boundaries.

**Table 3.1:** Initial Simulated Annealing configuration and the parameter's lower/upper bounds.

| **parameter** | **initial** | **lower bound** | **upper bound** |
|---|---:|---:|---:|
| SA-UCT | 1.0 | 0.7 | 2.0 |
| # MCTS iterations | 2,000 | 500 | 50,000 |
| # I/O samples | 30 | 10 | 60 |
| playout depth | 1 | 0 | 2 |

We aim at determining optimal parameters for different complexity classes. Classes are distinguished by the number of variables and by the expression's layer. Table 3.2

**Table 3.2:** Parameter choices for different complexity classes that depend on the expression layer and the number of variables. The parameters are the SA-UCT parameter (SA), the maximum number of MCTS iterations (# iter), the number of I/O samples (# I/O) and the playout depth (PD).

| | # variables | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **2** | | | | **5** | | | | **10** | | | | **20** | | | |
| **layer** | **SA** | **# iter** | **# I/O** | **PD** | **SA** | **# iter** | **# I/O** | **PD** | **SA** | **# iter** | **# I/O** | **PD** | **SA** | **# iter** | **# I/O** | **PD** |
| 3 | 1.42 | 40,569 | 20 | 0 | 1.55 | 32,375 | 17 | 0 | 1.74 | 42,397 | 20 | 1 | 1.38 | 28,089 | 18 | 1 |
| 5 | 1.84 | 35,399 | 14 | 0 | 1.11 | 28,792 | 23 | 0 | 1.29 | 27,365 | 23 | 0 | 0.92 | 34,050 | 12 | 0 |
| 7 | 1.25 | 28,363 | 20 | 0 | 1.01 | 30,838 | 23 | 0 | 1.23 | 15,285 | 22 | 0 | 1.42 | 11,086 | 22 | 0 |

illustrates the final configurations for 12 different complexity classes after 50 Simulated Annealing iterations. While the I/O samples and the playout depth are mostly in a similar range (0 and 20), there is a larger scope for the SA-UCT parameter and the maximum number of MCTS iterations. Especially for higher complexity classes, this is due to the optimization towards a high success rate within 120 seconds. The latter parameters strive towards larger values without this timeout.

Generally, the parameter configurations set a focus on exploration instead of exploitation. We follow this observation and adapt the configuration based on our problem statements. To describe a configuration, we provide a configuration vector of the form (SA-UCT, #iter, #I/O, PD).

### 3.6.2 Mixed Boolean-Arithmetic

Zhou et al. proposed the concept of *MBA expressions* [236]. By transforming simpler expressions and constants into MBA expressions over Boolean-arithmetic algebras, they can generate semantically-equivalent, but much more complex code which is arguably hard to reverse engineer. Effectively, this obfuscating transformation allows them to hide formulas and constants in plain code. In their paper, they define a Boolean-arithmetic algebra as follows:

**Definition 3.1 (Boolean-arithmetic algebra [236]).** *With $n$ a positive integer and $\mathsf{B} = \{0, 1\}$, the algebraic system $(\mathsf{B}^n, \wedge, \vee, \oplus, \neg, \leq, \geq, >, <, \leq^s, \geq^s, >^s, <^s, \neq, =, \gg^s, \gg, \ll, +, -, \cdot)$, where $\ll, \gg$ denote left and right shifts, $\cdot$ (or juxtaposition) denotes multiply, and signed compares and arithmetic right shift are indicated by $^s$, is a* Boolean-arithmetic algebra (BA-algebra), $\mathsf{BA}[n]$. *$n$ is the* dimension *of the algebra.*

Specifically, they highlight how $\mathsf{BA}[n]$ includes, amongst others, the Boolean algebra $(\mathsf{B}^n, \wedge, \vee, \neg)$ as well as the integer modular ring $\mathbb{Z}/(2^n)$. As a consequence, Mixed Boolean-Arithmetic (MBA) expressions over $\mathsf{B}^n$ are hard to simplify in practice. In general, we note that reducing a complex expression to an equivalent, but simpler one by, e. g., removing redundancies, is considered NP-hard [144].

Zhou et al. represent MBA expressions as polynomials over $\mathsf{BA}[n]$. While polynomial MBA expressions are conceptually not restricted in terms of complexity, Zhou et al. define *linear MBA expressions* as those polynomials with degree 1. In particular,

**Table 3.3:** Trace window statistics and synthesis performance for Tigress (MBA), VMProtect (VMP), Themida (flavor Tiger White, TM), and ROP gadgets.

| | **MBA** | **VMP** | **TM** | **ROP** |
|---|---|---|---|---|
| #trace windows | 500 | 12,577 | 2,448 | 78 |
| #unique windows | 500 | 449 | 106 | 78 |
| #instructions per window | 116 | 49 | 258 | 3 |
| #inputs per window | 5 | 2 | 15 | 3 |
| #outputs per window | 1 | 2 | 10 | 2 |
| #synthesis tasks | 500 | 1,123 | 1,092 | 178 |
| I/O sampling time (s) | 110 | 118 | 60 | 17 |
| overall synthesis time (s) | 2,020 | 4,160 | 9,946 | 829 |
| synthesis time per task (s) | 4.0 | 3.7 | 9.1 | 4.7 |

$f(x, y) = x - (x \oplus y) - 2(x \vee y) + 12564$ is a linear MBA expression, whereas $f(x, y) = x + 9(x \vee y)yx^3$ is not.

**Implementation in Tigress.** In practice, MBA expressions are used in the Tigress C Diversifier/Obfuscator by Collberg et al. [65] which uses the technique to encode integer variables and expressions in which they are used [61]. Further, Tigress also supports common arithmetic encodings to increase an expression's complexity, albeit not based on MBAs [62].

For example, the rather simple expression $x + y + z$ is transformed into the layer 23 expression $(((x \oplus y) + ((x \wedge y) \ll 1)) \vee z) + (((x \oplus y) + ((x \wedge y) \ll 1)) \wedge z)$ using its arithmetic encoding option. In a second transformation step, Tigress encodes it into a linear MBA expression of layer 383 (omitted due to complexity). Such expressions are hard to simplify symbolically.

**Evaluation Results.** We evaluated our approach to simplify MBA expressions using Syntia. As a testbed, we built a C program which calls 500 *randomly* generated functions. Each of these random functions takes 5 input variables and returns an expression of layer 3 to 5. Then, we applied the arithmetic encoding provided by Tigress, followed by the linear MBA encoding. The resulting program contained expressions of up to $2,821$ layers, the average layer being 156. The arithmetic encoding is applied to highlight that our approach is invariant to the code's increased *symbolic* complexity and is only concerned with semantical complexity.

Based on a concrete execution trace it can be observed that the 500 functions use, on average, 5 memory inputs (as parameters are passed on the stack) and one register output (the register containing the return value). Table 3.3 shows statistics for the analysis run using the configuration vector $(1.5, 50000, 50, 0)$. The first two components indicate a strong focus on *exploration* in favor of exploitation; due to the small number of synthesis tasks, we used 50 I/O samples to obtain more precise results.

The sampling phases completed in less than two minutes. Overall, the 500 synthesis tasks were finished in about 34 minutes, i. e., in 4.0 seconds per expression. We were

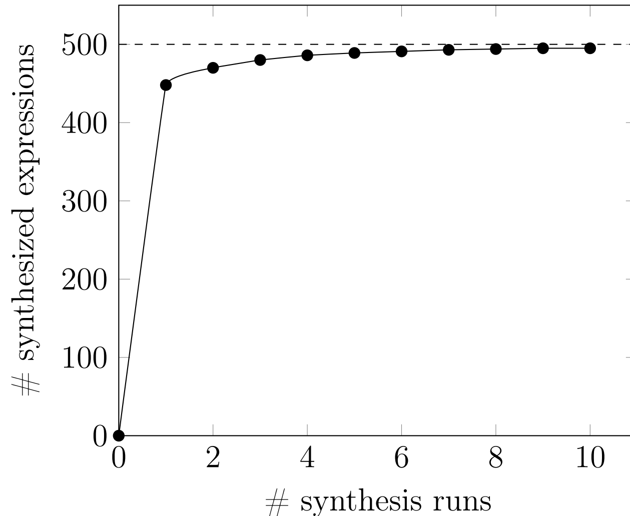


**Figure 3.5:** Subsequent synthesis runs increase the number of synthesized MBA expressions. Each point represents the average cumulative number of synthesized expressions from 15 separate experiments.

able to synthesize 448 out of 500 expressions (89.6%). The remaining expressions are not found due to the probabilistic nature of our algorithm; after 4 subsequent runs, we synthesized 489 expressions (97.8%) in total.

To get a better feeling for this probabilistic behavior, we compared the cumulative numbers of synthesized MBA expressions for 10 subsequent runs. Figure 3.5 shows the results averaged over 15 separate experiments. On average, the first run synthesizes 89.6% (448 expressions) of the 500 expressions. A second run yields 22 new expressions (94.0%), while a third run reveals 10 more expressions (96.0%). While converging to 500, the number of newly synthesized expressions decreases in subsequent runs. Comparing the fifth and the eighth run, we only found 5 new expressions (from 489 to 494). After the ninth run, Syntia synthesized 495 (99.0%) of the MBA expressions.

### 3.6.3 VM Instruction Handler

As introduced in Section 3.2.1.1, an instruction handler of a Virtual Machine implements the effects of an atomic instruction according to the custom VM-ISA. It operates on the VM context and can perform arbitrarily complex tasks. As handlers are heavily obfuscated, manual analysis of a handler's semantics is a time-consuming task.

**Attacking VMs.** When faced with virtualization-based obfuscations, an attacker typically has two options. For one, she can analyze the interpreter and, for each handler, extract all information required to *re-translate* the bytecode back to native instruction. Especially in face of handler duplication and bytecode blinding, this requires her to precisely capture all effects produced by the handlers. This includes both the high-level semantics with regard to input and output variables *as well as* the individual unblinding routines. In his paper, Rolles discusses how this type of attack requires complete understanding of the VM and therefore has to be repeated for each virtualization

obfuscator [183]. Thus, we note that this attack does not lend itself easily to full automation. Another approach is to perform analyses on the *bytecode level*. The idea is that while an attacker cannot learn the full semantics of the original code, the analysis of the interaction of handlers itself reveals enough information about the underlying code. This allows the attacker to skip details like bytecode blinding as she only requires the high-level semantics of a handler. Sharif et al. successfully mounted such an attack to recover the CFG of the virtualized function [199], but do not take semantics other than virtual instruction pointer updates into account.

We recognize the latter approach as promising and note how SYNTIA allows us to *automatically* extract the high-level semantics of arithmetical and logical instruction handlers. This is achieved by operating on an execution trace through the interpreter and simplify its individual handlers—as distinguished by trace window boundaries—using program synthesis. Especially, we highlight how obtaining the semantics of *one* handler automatically yields information about the underlying native code at *all points* of the trace where this specific handler is used to encode equivalent virtualized semantics.

**Evaluation Setup.** We evaluated SYNTIA to learn the semantics of arithmetic and logical VM instruction handlers in recent versions of VMProtect [213] (v3.0.9) and Themida [168] (v2.4.5.0). To this end, we built a program that covers bit-vector arithmetic for operand widths of 8, 16, 32, and 64 bit. Since we are interested in analyzing effects of the VM itself, using a synthetic program does not distort our results. For verification, we manually reverse engineered the VM layouts of VMProtect and Themida. Note that the commercial versions of both protection systems have been used to obfuscate the program. These are known to provide better obfuscation strength compared to the evaluation versions.

We argue that our evaluation program is representative of *any* program obfuscated with the respective VM-based obfuscating scheme. As seen in Section 3.2.1.1, common instructions map to a plethora of VM handlers. Consequently, if we succeed in recovering the semantics of these integral building blocks, we are at the same time able to recover other variations of native instructions using these handlers as well.

This motivates the design of our evaluation program, which aims to have a wide coverage of all possible arithmetic and logical operations. We note that this may not be the case for *real-world* test cases, which may not trigger all interesting VM handlers. To this extent, our evaluation program is, in fact, *more representative* than, e. g., malware samples.

#### 3.6.3.1 VMProtect

In its current version, VMProtect follows the *Direct Threaded Code* design principle (cf. Section 3.2.1.1). Each handler directly invokes the next handler based on the address encoded directly in the instruction's bytecode. Hence, reconstructing the handlers requires an instruction trace. Also, this impacts trace dissection: since VM handlers dispatch the next handler, they end with an indirect jump. Unsurprisingly, SYNTIA could automatically dissect the instruction trace into trace windows that

represent a single VM handler. As evident from Table 3.3, there are 449 *unique* trace windows out of a total of 12,577 in the instruction trace.

Further, VMProtect employs *handler duplication*. For example, the 449 instruction handlers contain 12 instances performing 8-bit addition, 11 instances for each of addition (for each flavor of 16-, 32-, 64-bit), `nor` (8-, 64-bit), left and right shift (32-, 64-bit); amongst multiple others. If SYNTIA is able to learn one instance in each group, it is safe to assume that it will successfully synthesize the full group, as supported by our results.

Similarly, the execution trace is made up of all possible handlers and some of them occur multiple times. Hence, if we correctly synthesize semantics for, e. g., a 64-bit addition, this immediately yields semantics for 772 trace windows (6.2% of the full trace, 32.0% of all arithmetic and logical trace windows in the trace). Equivalent reasoning applies to 16-bit `nor` operations in our trace (3.6% of the full trace, 18.8% of all arithmetic and logical trace windows). In total, our results reveal semantics for 19.7% of the full execution trace (2,482 out of 12,577 trace windows). Manual analysis suggests that the remaining trace semantics mostly consists of control-flow handling and stack operations. These are especially used when switching from the native to the VM context and amount for a large part of the execution trace.

On average, an individual instruction handler consists of 49 instructions. As VMProtect's VM is stack-based, binary arithmetic handlers pop two arguments from the stack and push the result onto the stack. This tremendously eases identification of inputs and outputs. Therefore, we mark memory operands as inputs and outputs and use the configuration vector $(1.5, 30000, 20, 0)$ for the synthesis. The sampling phase finished in less than two minutes. Overall, the 1,123 synthesis tasks completed in less than an hour, which amounts to merely 3.7 seconds per task. In total, in our first run, we automatically identified 190 out of 196 arithmetical and logical handlers (96.9%). The remaining 6 handlers implement 8-bit divisions and shifts. Due to their representation in x86 assembly code, SYNTIA needs to synthesize more complex expressions with nested data type conversions. As the analysis is probabilistic in nature, we scheduled five more runs which yielded 4 new handlers. Thus, we are able to automatically pinpoint 98.9% of all arithmetic and logical instruction handlers in VMProtect.

#### 3.6.3.2 Themida

The protection solution Themida supports three basic VM flavors, namely, Tiger, Fish, and Dolphin. Each flavor can further be customized to use one of three obfuscation levels, in increasing complexity: White, Red, and Black. We note that related work on deobfuscation does not directly mention the exact configuration used for Themida. In hopes to be comparable, we opted to use the default flavor Tiger, using level White, in our evaluation. Unlike VMProtect, Tiger White uses an explicit handler table while inlining the dispatcher routine; i. e., it follows the *Threaded Code* design principle (cf. Section 3.2.1.1). Consequently, trace dissection again yields one trace window per instruction handler. Even though the central handler table lists 1,111 handlers, we identified 106 *unique* trace windows along the concrete execution trace.

Themida implements a register-based architecture and stores intermediate computations in one of many register available in the VM context. This, in turn, affects the identification of input and output variables. While in the case of VMProtect, inputs and outputs are directly taken from two slots on the stack, Themida has a significantly higher number of potential inputs and outputs (i. e., all virtual registers in the VM context, 10 to 15 in our case).

Tiger White supports handlers for addition, subtraction, multiplication, logical left and right shift, bitwise operations and unary subtraction; each for different operand widths. In contrast to VMProtect, handlers are neither duplicated nor do they occur multiple times in the execution trace. Hence, the trace itself is much more compact, spanning 2,448 trace windows in total; roughly 5 times shorter than VMProtect's. Still, Themida's handlers are much longer, with 258 instructions on average.

We ran the analysis using the configuration vector $(1.8, 50000, 20, 0)$. Due to the higher number of inputs, this configuration—in comparison to the previous section—sets a much higher focus on exploration as indicated by higher values chosen for the first two parameters. Sampling finished in one minute, whereas the synthesis phase took around 166 minutes. At 1,092 synthesis tasks, this amounts to roughly 9.1 seconds per task. Eventually, we automatically learned the semantics of 34 out of 36 arithmetic and logical handlers (94.4%). The remaining handlers (8-bit subtraction and logical or) were not found as we were unable to complete the sampling phase due to crashes in Unicorn engine.

### 3.6.4 ROP Gadget Analysis

We further evaluated SYNTIA on ROP gadgets, specifically, on four samples that were thankfully provided by Debray [225]. They implement bubble sort, factorials, Fibonacci, and matrix multiplication in ROP. To have a larger set of samples, we also used a CTF challenge [174] that has been generated by the ROP compiler Q [196] and another Fibonacci implementation that has been generated with ROPC [170].

SYNTIA automatically dissected the instruction traces into 156 individual gadgets. Since many gadgets use exactly the same instructions, we unified them into 78 unique gadgets. On average, a gadget consists of 3 instructions with 3 inputs and 2 outputs (register and memory locations).

Due to the small numbers of inputs and synthesis tasks, we chose the configuration vector $(1.5, 100000, 50, 0)$ that sets a very strong focus on exploration while accepting a higher running time. Especially, we experienced both effects for the maximum number of MCTS iterations.

SYNTIA synthesized partial semantics for 97.4% of the gadgets in less than 14 minutes; in total, we were successful in 163 out of the 178 (91.5%) synthesis tasks. Our synthesis results include 58 assignments, 17 binary additions, 5 ternary additions, 4 unary minus, 4 binary subtractions, 4 register increments/decrements, 2 binary multiplications and 1 bitwise and. In addition, we found 68 stack pointer increments due to `ret` statements.

The results do not include larger constants or operations such as `ror` as they are not part of our grammar.

## 3.7 Discussion

In the following, we discuss different aspects of program synthesis for trace simplification and MCTS-based program synthesis. Furthermore, we point out limitations of our approach as well as future work.

**Program Synthesis for Trace Simplification.** Current research on deobfuscation [66, 199, 224, 225] operates on instruction traces and uses a mixed approach consisting of symbolic execution [224] and taint analysis [223]; two approaches that require a precise analysis of the underlying code. While techniques exist that defeat taint analysis [54, 191], recent work shows that symbolic execution can similarly be attacked [35].

Program synthesis is an orthogonal approach that operates on a purely semantical level as opposed to (binary) code analysis; it is oblivious to the underlying code constructs. As a result, syntactical aspects of code complexity such as obfuscation or instruction count do not influence program synthesis negatively. It is merely concerned with the complexity of the code's *semantics*. The only exception where code-level artifacts matter is the generation of I/O samples; however, this can be realized with small overhead compared to regular execution time using dynamic binary instrumentation [163, 172].

Commonly, instruction traces contain repetitions of unique trace windows that can be caused by loops or repeated function calls to the same function. By synthesizing these trace windows, the synthesized semantics pertain for all appearances on the instruction trace; the more frequently these trace windows occur in the trace, the higher the percentage of *known* semantics in the instruction trace. We stress how VM-based obfuscation schemes do this to the extreme: a relatively small number of unique trace windows are used over the whole trace.

In general, the synthesis results may not be precise semantics since we approximate them based on I/O samples. If these do not reflect the full semantics, the synthesis misses edge cases. For instance, we sometimes cannot distinguish between an arithmetic and a logical right shift if the random inputs are no *distinguishing* inputs. We point out that this is not necessarily a limitation, since a human analyst might still get valuable insights from the approximated semantics.

As future work, we consider improving trace simplification by a stratified synthesis approach [112]. The main idea is to incrementally synthesize larger parts of the instruction trace based on previous results and successively approximate high-level semantics of the entire trace. Further, we note that the work by Sharif et al. [199] is complementary to our synthesis approach and would also allow us to identify control flow. Likewise, extending the grammar by control-flow operations is another viable approach to tackle this limitation.

**MCTS-based Program Synthesis.** Compared to SMT-based program synthesis, we obtain *candidate* solutions, even if the synthesizer does not find an *exact* result. This is particularly beneficial for applications such as deobfuscation, since a human analyst can sometimes infer the full semantics. We decided to utilize MCTS for program synthesis since it has been proven very effective when operating on large search trees without domain knowledge. However, our approach is not limited to MCTS, other stochastic algorithms are also applicable.

Drawn from the observations made in Section 3.6, we infer that the MCTS approach is much more effective with a configuration that focuses on exploration instead of exploitation. The SA-UCT parameter ensures that paths with a higher reward are explored in-depth in later stages of the algorithm. We still try to improve exploration strategies, for instance with *Nested Monte Carlo Tree Search* [153] and *Monte Carlo Beam Search* [55].

**Limitations.** In general, limits of program synthesis apply to our approach as well. Non-determinism and point functions—Boolean functions that return 1 for exactly one input out of a large input domain—cannot be synthesized practically. This also holds for semantics that have strong confusion and diffusion properties, such as cryptographic algorithms. These are inherently very complex, non-linear expressions with a deep nesting level. Our approach is also limited by the choice of trace window boundaries; ending a trace window in intermediate computation steps may produce formulas that are not *meaningful* at all.

## 3.8 Related Work

We now review related work for program synthesis, Monte Carlo Tree Search and deobfuscation. Furthermore, we describe how our work fits into these research areas.

**Program Synthesis.** Gulwani et al. [104] introduced an SMT-based program synthesis approach for loop-free programs that requires a logical specification of the desired program behavior. Building on this, Jha et al. [123] replaced the specification with an I/O oracle. Upon generation of *multiple* valid program candidates, they derive *distinguishing inputs* that are used for subsequent oracle queries. They demonstrated their use case by simplifying a string obfuscation routine of MyDoom. Godfroid and Taly [91] used an SMT-based approach to learn the formal semantics of CPU instruction sets; for this, they use the CPU as I/O oracle.

Schkufza et al. [193] proved that stochastic program synthesis often outperforms SMT-based approaches. This is mostly due to the fact that common SMT-based approaches effectively enumerate *all* programs of a given size or prove their non-existence. On the other hand, stochastic approaches focus on promising parts of the search space without searching exhaustively. Schkufza et al. use this technique for stochastic superoptimization on the basis of their tool *STOKE*. Recent work by Heule et al. [112] demonstrates a stratified approach to learn the semantics of the x86-64 instruction set, based on STOKE. Their main idea is to re-use synthesis results to synthesize more complex instructions in an iterative manner. To the best of our knowledge, STOKE is

the only other stochastic synthesis tool that is able to synthesize low-level semantics. By design, their code only produces Intel x86 code.

In our case, stochastic techniques have additional properties that are not achieved by previous tools: we obtain partial results that are often already „close" to a real solution and might be helpful for a human analyst who tries to understand obfuscated code. Furthermore, we can encode arbitrary complex function symbols in our grammar (e. g., complex encoding schemes or hash functions); a characteristic that is not easily reproduced by SMT-based approaches.

In the context of non-academic work, Rolles applied some of the above mentioned SMT-based approaches to reverse engineering and deobfuscation [184]. Amongst others, he learned obfuscation rules by adapting peephole superoptimization techniques [36] and extracted metamorphic code using an oracle-guided approach. In his recent work, he performs SMT-based shellcode synthesis [185].

**Monte Carlo Tree Search.** MCTS has been widely studied in the area of AI in games [82, 153, 192, 207]. Ruijl et al. [187] combine Simulated Annealing and MCTS by introducing SA-UCT for expression simplification. Lim and Yoo [146] describe an early exploration on how MCTS can be used for program synthesis and note that it shows comparable performance to genetic programming. We extend the research of MCTS-based program synthesis by applying SA-UCT and introducing node pruning. For our synthesis approach, we designed a context-free grammar that learns the semantics of Intel x86 code.

**Deobfuscation.** Rolles provides an academic analysis of a VM-based obfuscator and outlines a possible attack on such schemes in general [183]. He proposes using static analysis to re-translate the VM's bytecode back into native instructions. This, however, requires minute analysis of *each* obfuscator and hence is time-consuming and prone to minor modifications of the scheme. Kinder is also concerned with (static) analysis of VMs [130]. Specifically, he lifts a *location-sensitive* analysis to be usable in presence of virtualization-based obfuscation schemes. His work highlights how the execution trace of a VM, while performing various computations, always exhibits a recurring set of addresses. As seen in Section 3.6, our approach actually benefits from this side effect. In contrast, Sharif et al. [199] analyze VMs in a *dynamic* manner and record execution traces. In contrast to the work of Rolles, their goal is not to re-translate, but to directly analyze the bytecode itself. Specifically, they aim to reconstruct parts of the underlying code's control flow from the bytecode. This approach is closest to our work as we are, in turn, mostly concerned with arithmetic and logical semantics of a handler.

More recent results include work by Coogan et al. [66] as well as Yadegari et al. [225]. Both approaches seek to deobfuscate code based on execution traces by further making use of symbolic execution and taint tracking. The former approach is focused on the *value flow* to system calls to reduce a trace whereas Yadegari et al. propose a more general approach and aim to produce fully deobfuscated code. However, to counteract symbolic execution-based deobfuscation approaches, Banescu et al. propose novel obfuscating transformations that specifically target their deficiencies [35]. For one, they propose a construct akin to *random opaque predicates* [64] that deliberately explodes the

number of paths through a function. A second technique preserves program behavior of the obfuscated program for specific input invariants *only*, effectively increasing the input domains and thus the search space for symbolic executors.

Guinet et al. present *arybo*, a framework to simplify MBA expressions [101]. In essence, they perform bit-blasting and use a Boolean expression solver that tries to simplify the expression symbolically. Eyrolles [81] describes a symbolic approach that uses pattern matching. Furthermore, she suggests improvements of current MBA-obfuscated implementations that impede these symbolic deobfuscation techniques [80]. To this effect, we also argue that symbolic simplification is inherently limited by the complexity of the input expression. However, we demonstrated that a synthesis-based approach allows fine-tuned simplification, irrespective of *syntactical* complexity, while producing approximate intermediate results.

## 3.9 Conclusion

With our prototype implementation of Syntia we have shown that program synthesis can aid in deobfuscation of real-world obfuscated code. In general, our approach is vastly different in nature compared to proposed deobfuscation techniques and hence may succeed in scenarios where approaches requiring precise code semantics fail.

# Chapter 4

# Input Structure Synthesis to Guide Feedback-driven Fuzzing

## 4.1 Introduction

As the amount of software impacting the (digital) life of nearly every citizen grows, effective and efficient testing mechanisms for software become increasingly important. The publication of the fuzzing framework AFL [231] and its success at uncovering a huge number of bugs in highly relevant software has spawned a large body of research on effective feedback-based fuzzing. AFL and its derivatives have largely conquered automated, dynamic software testing and are used to uncover new security issues and bugs every day. However, while great progress has been achieved in the field of fuzzing, many hard cases still require manual user interaction to generate satisfying test coverage. To make fuzzing available to more programmers and thus scale it to more and more target programs, the amount of expert knowledge that is required to effectively fuzz should be reduced to a minimum. Therefore, it is an important goal for fuzzing research to develop fuzzing techniques that require less user interaction and, in particular, less domain knowledge to enable more automated software testing.

**Structured Input Languages.** One common challenge for current fuzzing techniques are programs which process highly structured input languages such as interpreters, compilers, text-based network protocols or markup languages. Typically, such inputs are consumed by the program in two stages: parsing and semantic analysis. If parsing of the input fails, deeper parts of the target program—containing the actual application logic—fail to execute; hence, bugs hidden „deep“ in the code cannot be reached. Even advanced feedback fuzzers—such as AFL—are typically unable to produce diverse sets of syntactically valid inputs. This leads to an imbalance, as these programs are part of the most relevant attack surface in practice, yet are currently unable to be fuzzed effectively. A prominent example are browsers, as they parse a multitude of highly-structured inputs, ranging from XML or CSS to JavaScript and SQL queries.

Previous approaches to address this problem are typically based on manually provided grammars or seed corpora [33, 78, 167, 186]. On the downside, such methods require

human experts to (often manually) specify the grammar or suitable seed corpora, which becomes next to impossible for applications with undocumented or proprietary input specifications. An orthogonal line of work tries to utilize advanced program analysis techniques to automatically infer grammars [37, 38, 98]. Typically performed as a pre-processing step, such methods are used for generating a grammar that guides the fuzzing process. However, since this grammar is treated as immutable, no additional learning takes place during the actual fuzzing run.

**Our Approach.** In this chapter, we present a novel, fully automated method to fuzz programs with a highly structured input language, without the need for any human expert or domain knowledge. Our approach is based on two key observations: First, we can use code coverage feedback to automatically infer structural properties of the input language. Second, the precise and „correct" grammars generated by previous approaches are actually unnecessary in practice: since fuzzers have the virtue of high test case throughput, they can deal with a significant amount of noise and imprecision. In fact, in some programs (such as `Boolector`) with a rather diverse set of input languages, the additional noise even benefits the fuzz testing. In a similar vein, there are often program paths which can only be accessed by inputs *outside* of the formal specifications, e. g., due to incomplete or imprecise implementations or error handling code.

Instead of using a pre-processing step, our technique is directly integrated in the fuzzing process itself. We propose a set of generalizations and mutations that resemble the inner workings of a grammar-based fuzzer, without the need for an explicit grammar. Our generalization algorithm analyzes each newly found input and tries to identify substrings of the input which can be replaced or reused in other positions. Based on this information, the mutation operators recombine fragments from existing inputs. Overall, this results in synthesizing new, structured inputs without prior knowledge of the underlying specification.

We have implemented a prototype of the proposed approach in a tool called GRIMOIRE[1]. GRIMOIRE does not need any specification of the input language and operates in an automated manner without requiring human assistance; in particular, without the need for a format specification or seed corpus. Since our techniques make no assumption about the program or its environment behavior, GRIMOIRE can be easily applied to closed-source targets as well.

To demonstrate the practical feasibility of our approach, we perform a series of experiments. In a first step, we select a diverse set of programs for a comparative evaluation: we evaluate GRIMOIRE against other fuzzers on four scripting language interpreters (`mruby`, `PHP`, `Lua` and `JavaScriptCore`), a compiler (`TCC`), an assembler (`NASM`), a database (`SQLite`), a parser (`libxml`) and an SMT solver (`Boolector`). Demonstrating that our approach can be applied in many different scenarios without requiring any kind of expert knowledge, such as an input specification. The evaluation results show that our approach outperforms all existing coverage-guided fuzzers; in the case of `Boolector`, GRIMOIRE finds up to 87% more coverage than the baseline (REDQUEEN). Second, we

[1] A *grimoire* is a magical book that recombines magical elements to formulas. Furthermore, it has the same word stem as the Old French word for grammar—namely, *gramaire*.

evaluate GRIMOIRE against state-of-the-art grammar-based fuzzers. We observe that in situations where an input specification is available, it is advisable to use GRIMOIRE in addition to a grammar fuzzer to further increase the test coverage found by grammar fuzzers. Third, we evaluate GRIMOIRE against current state-of-the-art approaches that use automatically inferred grammars for fuzzing and found that we can significantly outperform such approaches. Overall, GRIMOIRE found 19 distinct memory corruption bugs that we manually verified. We responsibly disclosed all of them to the vendors and obtained 11 CVEs. During our evaluation, the next best fuzzer only found 5 of these bugs. In fact, GRIMOIRE found more bugs than all five other fuzzers combined.

**Contributions.** In summary, we make the following contributions:

- We present the design, implementation and evaluation of GRIMOIRE, an approach to fully automatically fuzz highly structured formats with no human interaction.
- We show that even though GRIMOIRE is a binary-only fuzzer that needs no seeds or grammar as input, it still outperforms many fuzzers that make significantly stronger assumptions (e. g., access to seeds, grammar specifications and source code).
- We found and reported multiple bugs in various common projects such as `PHP`, `gnuplot` and `NASM`.

## 4.2 Challenges in Fuzzing Structured Languages

Fuzzing has proven extremely useful as a tool for uncovering bugs in software. Thus, a multitude of corresponding methods has emerged over the last decades. In this chapter, we discuss various existing approaches and their drawbacks. Afterward, we combine advantages of different techniques to derive novel methods for fuzzing structured languages.

### 4.2.1 Black-box Fuzzing

Many fuzzers (so-called black-box fuzzers) do not know any internals of the target application. They generate a random stream of malformed inputs in the hope of causing a fault in the target application. Over time, many such fuzzer were developed. To name just a few, RADAMSA [111], PEACH [78], SULLEY [167] and ZZUF [115] are all well-known examples from this category.

With such a diverse set of fuzzers, widely different strategies are employed to generate a stream of inputs. We further divide black-box fuzzers into two subcategories based on the strategies used, namely structured fuzzers and mutational fuzzers.

*Structured fuzzers* require a specification to create (mostly) valid inputs [169, 211]. This has the advantage that even a „blind“ black-box fuzzer can generate very good test coverage and explore almost all code paths. However, the quality of the testing process depends entirely on the quality of the specification, while building a good specification can be extremely time-consuming.

*Mutational fuzzers* assume a set of interesting inputs *(seeds)* that are iteratively mutated. For example, fuzzers such as RADAMSA [111] flip bits, repeat random byte sequences or add a random value to an arbitrary offset. The effectiveness of mutational fuzzing depends entirely on a well-chosen set of seeds. Due to the nature of these mutations, only tiny changes are applied to inputs. Consequently, only bugs „close" to the seed inputs will be uncovered.

### 4.2.2 Gray-box Fuzzing

While black-box fuzzers were already highly effective at finding bugs, the introduction of AFL [231] caused a downright explosion in research on fuzzing tools. AFL was the first well-known mutational *gray-box fuzzer* or *coverage-guided fuzzer* that uses code coverage information for each generated test case to guide the fuzzing process. Test cases that trigger new coverage are used as seeds for further mutations. Hence, the fuzzer is able to explore the target application in much more detail than a black-box mutation-based fuzzer would be able to. As interesting test cases are stored, mutations can accumulate and the fuzzer is able to find bugs that differ greatly from the original inputs.

The seeds supplied to AFL are stored in a queue of inputs used for testing. AFL itself uses a large loop that (1) picks an input from the queue, (2) mutates it and (3) runs the target application with the mutated input. If executing the mutated input yields new test coverage, the mutated input is stored in the queue, otherwise discarded. Finally, the next input in the queue is processed.

Similar approaches have been explored even before the publication of AFL; however, these approaches were usually rather slow [79]. The success of AFL was due to its highly efficient implementation. One of the biggest problems for fuzzers was effectively computing and comparing test coverage. AFL introduced a very fast approximation scheme based on compile-time instrumentations. Later, the same scheme was adopted to use other instrumentation techniques such as dynamic binary instrumentation [142] or Intel-PT [194].

**Coverage Calculation.** To obtain coverage feedback, AFL uses a compiler pass that inserts instrumentations at the beginning of every basic block. This instrumentation calculates a 16-bit ID for each edge in the control-low graph of the target program. To obtain such an ID efficiently, the compiler pass assigns a random ID to each basic block. Then, the instrumentation calculates

$$\texttt{edge_id} := (\texttt{last_bb_id} * 2) \oplus \texttt{bb_id} \tag{4.1}$$

and sets `last_bb_id` to the current basic block ID. The multiplication by two serves to distinguish self-loops (all loops with a single basic block would use the edge ID zero otherwise) and the direction of the control-flow transfer.

In the last step, the instrumentation uses the edge ID as an index into the so-called „shared map" and increments the corresponding byte in this map. After the execution

is finished, the shared map contains a representation of the edge coverage of the run [233].

**Coverage Comparison.** After AFL obtained this array of edge counts for a given test case, it compares the resulting coverage to the coverage one observed by all previous test cases. To avoid individually comparing test cases, it uses a global bitmap that stores all the coverage that has been previously observed. If any entry in the shared map contains a value higher than that in the global map, the input triggered some new edge coverage and is stored in the queue. The corresponding entry in the global map is also updated to reflect that this new value has been observed.

To be more precise, AFL does not compare the values directly. Instead, it applies a process called *bucketing*: Each byte in the shared map is reduced to a byte where only a single bit is set. Then, to compare the global map against the shared map, AFL only checks whether the shared map contains a bit that is still zero in the global map. This operation can be performed very efficiently for eight bytes at a time.

We distinguish between inputs that triggered an edge that has never been seen before (e. g., where the value in the global map was zero) and those that only encountered a new number of loop iterations for the given edge. When an entry is encountered the first time, we say „the input found a new byte“; otherwise, we say „the input only found new bits“.

**Example 4.1.** *Assume the target contains an edge from a basic block with ID* $0x0010$ *and another basic block with ID* $0x000a$*. The edge ID is calculated—according to Equation 4.1—as* $\mathrm{0x0010} * 2 \oplus \mathrm{0x000}a = 0x002a$*. This edge is executed ten times. After execution, the shared map byte at offset* $0x002a$ *contains the value* $10$*. The bucketing process assigns the bucket* $8$ *(*$0b1000$*). Then, we check whether the fourth bit in the global map entry at offset* $0x002a$ *is set. If not, we store the input in the queue.*

Based on the design of AFL, various other fuzzers were built that try to overcome common challenges in fuzzing. Consider a four-byte header is used to identify a file format or message. A fuzzer that is only guided by code coverage would find it very difficult to guess the correct header. To overcome similar scenarios, different approaches apply a wide variety of techniques. For example, REDQUEEN [34] only uses breakpoints to include values observed during execution in the feedback. ANGORA [59] makes use of taint tracking and gradient inference to tackle the same challenges. Lastly, tools like QSYM [230] and T-FUZZ [171] rely on symbolic execution to overcome hard constraints.

While these approaches are effective at solving certain constraints, they are woefully unprepared to deal with the challenges that are posed by highly structured input languages. In such cases, the „interesting“ program logic is only reached after the input has been parsed. Consequently, a single bit flip in the wrong position will cause the target to discard the input; thus, mutations need to maintain the syntactical validity of the input. While fuzzers like AFL occasionally manage to generate syntactically valid inputs, they cannot systematically explore the set of syntactically valid inputs. One way to overcome this problem is by using a format specification during the fuzzing process.

### 4.2.3 Structured Gray-box Fuzzing

The simple bitwise mutations employed by AFL and similar mutational fuzzers are a poor fit for many formats with a large set of syntactical constraints. If these constraints are checked before the interesting part of the program logic is executed, almost no input will manage to exercise the code we are interested in. Examples for such programs are ample, since nearly all kinds of interpreters, compilers, network protocols and text-based formats fall into this category. In fact, it is not uncommon that a blind fuzzer with an input specification is able to cover more interesting code than AFL with access to coverage guidance. Recently, multiple projects combined the best of both worlds: use of an input specification and coverage feedback. This way, fuzzers like NAUTILUS [33] and AFLSMART [173] are able to produce a much more diverse set of syntactically valid inputs.

However, while they can explore the target application's input space much more thoroughly, they require a specification for the input format. Hence, the approach sometimes demands a significant upfront effort to obtain such a specification, severely limiting the usefulness as a general-purpose tool. Additionally, if the specification lacks some aspects of the input format, the fuzzer will not be able to explore the corresponding area of the target.

### 4.2.4 Grammar Inference

To overcome the need for a preexisting specification, some approaches were developed that aim to learn the input specification (typically in the form of a context-free grammar) directly from the program itself. For example, GLADE [38] assumes a black-box oracle that tells whether an input is syntactically correct or not. Using this oracle, GLADE first tries to generalize existing inputs to regular expressions. From these, it infers recursive replacement rules that form a context-free grammar.

This approach has some serious drawbacks. First of all, it requires both a modified target program serving as a syntax oracle and a set of syntactically valid inputs. Secondly, the approach often takes multiple hours and—as our evaluation shows—the resulting grammars are typically an inferior fit for fuzzing purposes.

To avoid modifying the target application, other approaches employ the unmodified program. In particular, AUTOGRAM [118] applies taint tracking to learn which parts of the input are used at which location in the program. Based on these results, it generalizes a grammar. PYGMALION [98] uses symbolic execution for similar purposes. However, both approaches are limited to recursive descent parsers and also require good seeds.

### 4.2.5 Shortcomings of Existing Fuzzers

In summary, using fuzzing to test highly structured input languages is not an easy task; all discussed approaches dealing with highly structured formats have some serious drawbacks. In all cases, a significant amount of human effort is required: either to acquire an input specification, modify the target to obtain a syntax oracle or to gather

good seed cases. Also, these approaches—especially AUTOGRAM and PYGMALION—would be very hard to adapt to binary targets and are unable to infer the grammars for parsers generated by tools such as `GNU Bison` [84] or `Yacc` [124].

In this chapter, we develop an approach that combines grammar inference with fuzzing. Great care is taken to ensure that the approach seamlessly integrates into the normal fuzzing process and can easily be adapted to other state-of-the-art feedback fuzzers. In particular, our approach neither requires any specific instrumentation beyond what AFL-style fuzzers already provide nor any form of human input or good seeds.

## 4.3 Design

Based on the challenges identified above, we now introduce the design of GRIMOIRE, a fully automated approach that synthesizes the target's structured input language during fuzzing. Furthermore, we present structure-aware mutations that cross significant gaps in the program space. Note that none of the limitations discudded before applies to our approach. To emphasize, our design does not require any previous information about the input structure. Instead, we learn an ad-hoc specification based on the program semantics and use it for coverage-guided fuzzing.

We first provide a high-level overview of GRIMOIRE, followed by a detailed description. GRIMOIRE is based on identifying and recombining fragments in inputs that trigger new code coverage during a normal fuzzing session. It is implemented as an additional fuzzing stage on top of a coverage-guided fuzzer. In this stage, we strip every new input (that is found by the fuzzer and produced new coverage) by replacing those parts of the input that can be modified or replaced without affecting the input's new coverage by the symbol □. This can be understood as a generalization, in which we reduce inputs to the fragments that trigger new coverage, while maintaining information about *gaps* or *candidate positions* (denoted by □). These gaps are later used to splice in fragments from other inputs.

**Example 4.2.** *Consider the input „`if(x>1) then x=3 end`" and assume it was the first input to trigger the coverage for a syntactically correct if-statement as well as for „`x>1`". We can delete the substring „`x=3`" without affecting the interesting new coverage since the if-statement remains syntactically correct. Additionally, the space between the condition and the „`then`" is not mandatory. Therefore, we obtain the generalized input „`if(x>1)□then □end`".*

After a set of inputs was successfully generalized, fragments from the generalized inputs are recombined to produce new candidate inputs. We incorporate various different strategies to combine existing fragments, learned tokens (a special form of substrings) and strings from the binary in an automated manner.

**Example 4.3.** *Assume we obtained the following generalized inputs: „`if(x>1)□then □end`" and „`□x=□y+□`". We can use this information in many ways to generate plausible recombinations. For example, starting with the input „`if(x>1)□then □end`", we can replace the second gap with the second input, obtaining „`if(x>1)□then □x=□y+□end`". Afterward, we choose the slice „`□y+□`" from the second input and splice it into the*

*fourth gap and obtain „`if(x>1)□then □x=□y+□y+□end`“. In a last step, we replace all remaining gaps by an empty string. Thus, the final input is „`if(x>1)then x=y+y+end`“.*

One could think of our approach as a context-free grammar with a single non-terminal input □ and all fragments of generalized inputs as production rules. Using these loose, grammar-like recombination methods in combination with feedback-driven fuzzing, we are able to automatically learn interesting structures.

### 4.3.1 Input Generalization

We try to generalize inputs that produced new coverage (e. g., inputs that introduced new bytes to the bitmap, cf. Section 4.2.2). The generalization process (Algorithm 3) tries to identify parts of the input that are irrelevant and fragments that caused new coverage. In a first step, we use a set of rules to obtain fragment boundaries (Line 3). Consecutively, we remove individual fragments (Line 4). After each step, we check if the reduced input still triggers the same new coverage bytes as the original input (Line 5). If this is the case, we replace the fragment that was removed by a □ and keep the reduced input (Line 6).

**Algorithm 3:** Generalizing an input through fragment identification.

**Data:** `input` is the input to generalize, `new_bytes` are the new bytes of the input, `splitting_rule` defines how to split an input
**Result:** A generalized version of `input`

```
1 start ← 0
2 while start < input.length() do
3     end ← find_next_boundary(input, splitting_rule)
4     candidate ← remove_substring(input, start, end)
5     if get_new_bytes(candidate) == new_bytes then
6         input ← replace_by_gap(input, start, end)
7     start ← end
8 input ← merge_adjacent_gaps(input)
```

**Example 4.4.** *Consider input „`pprint 'aaaa'`“ triggers the new bytes* 20 *and* 33 *because of the pprint statement. Furthermore, assume that we use a rule that splits inputs into non-overlapping chunks of length two. Then, we obtain the chunks „`pp`“, „`ri`“, „`nt`“, „` '`“, „`aa`“, „`aa`“ and „`'`“. If we remove any of the first four chunks, the modified input will not trigger the same new bytes since we corrupted the pprint statement. However, if we remove the fifth or sixth chunk, we still trigger the bytes* 20 *and* 33 *since the pprint statement remains valid. Therefore, we reduce the input to „`pprint '□□'`“. As we have two adjacent □, we merge them into one. The generalized input is „`pprint '□'`“.*

To generalize an input as much as possible, we use several fragmentation strategies for which we apply Algorithm 3 repeatedly. First, we split the input into overlapping chunks of size 256, 128, 64, 32, 2 and 1 to remove large uninteresting parts as early as possible. Afterward, we dissect at different separators such as ‘`.`’, ‘`;`’, ‘`,`’, ‘`\n`’, ‘`\r`’,

‘\t’, ‘#’ and ‘ ’. As a consequence, we can remove one or more statements in code, comments and other parts that did not cause the input’s new coverage. Finally, we split at different kinds of brackets and quotation marks. These fragments can help to generalize constructs such as function parameters or nested expressions. In detail, we split in between of ‘()’, ‘[]’, ‘{}’, ‘<>’ as well as single and double quotes. To guess different nesting levels in between these pairs of opening/closing characters, we extend Algorithm 3 as follows: If the current index `start` matches an opening character, we search the furthermost matching closing character, create a `candidate` by removing the substring in between and check if it triggers the same new coverage. We iteratively do this by choosing the next furthermost closing character—effectively shrinking the fragment size—until we find a substring that can be removed without changing the `new_bytes` or until we reach the index `start`. In doing so, we are able to remove the largest matching fragments from the input that are irrelevant to the input’s new coverage.

Since we want to recombine (generalized) inputs to find new coverage—as we describe in the following section—we store the original input as well as its generalization. Furthermore, we split the generalized input at every □ and store the substrings *(tokens)* in a set; these tokens often are syntactically interesting fragments of the structured input language.

**Example 4.5.** *We map „`if(x>1) then x=3 end`“ to its generalization „`if(x>1)`□`then` □`end`“. In addition, we extract the tokens „`if(x>1)`“, „`then` “ and „`end`“. For the generalized input „□`x=`□`y+`□“, we remember the tokens „`x=`“ and „`y+`“.*

### 4.3.2 Input Mutation

Grimoire builds upon knowledge obtained from the generalization stage to generate inputs that have good chances of finding new coverage. For this, it recombines (fragments of) generalized inputs, tokens and strings (stored in a dictionary) that are automatically obtained from the data section of the target’s binary. On a high level, we can divide our mutations into three standalone operations: input extension, recursive replacement and string replacement.

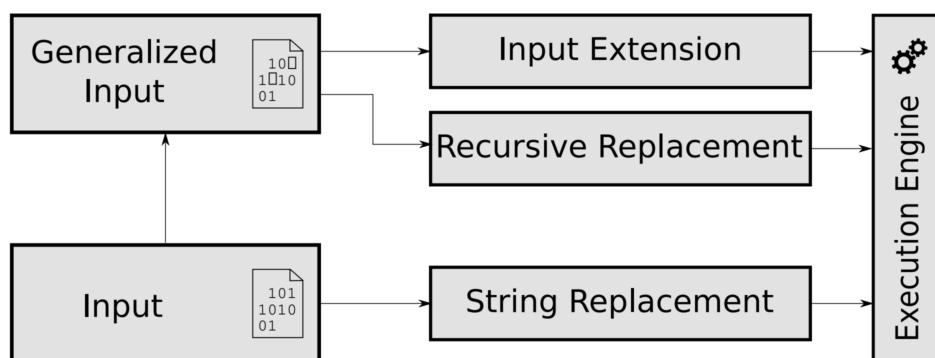


**Figure 4.1:** A high-level overview of our mutations. Given an input, we apply various mutations on its generalized and original form. Each mutation then feeds mutated variants of the input to the fuzzer’s execution engine.

Given the current input from the fuzzing queue, we add these mutations to the so-called *havoc phase* [34] as described in Algorithm 4. First, we use Redqueen's `havoc_amount` to determine—based on the input's performance—how often we should apply the following mutations (in general, between 512 and 1024 times). First, if the input triggered new bytes in the bitmap, we take its generalized form and apply the structure-aware mutations `input_extension` and `recursive_replacement`. Afterward, we take the original input string (accessed by `input.content()`) and apply the `string_replacement` mutation. This process is illustrated in Figure 4.1.

**Algorithm 4:** High-level overview of the mutations introduced in Grimoire.

**Data:** `input` is the current input in the queue, `generalized` is the set of all previously generalized inputs, tokens and strings from the dictionary, `strings` is the provided dictionary obtained from the binary

```
1 content ← input.content()
2 n ← havoc_amount(input.performance())
3 for i ← 0 to n do
4     if input.is_generalized() then
5         input_extension(input, generalized)
6         recursive_replacement(input, generalized)
7     string_replacement(content, strings)
```

Before we describe our mutations in detail, we explain two functions that all mutations have in common—`random_generalized` and `send_to_fuzzer`. The function `random_generalized` takes as input a set of all previously generalized inputs, tokens and strings from the dictionary and returns—based on random coin flips—a random (slice of a ) generalized input, token or string. In case we pick an input slice, we select a substring between two arbitrary □ in a generalized input. This is illustrated in Algorithm 5. The other function, `send_to_fuzzer`, implies that the fuzzer executes the target application with the mutated input. It expects concrete inputs. Thus, mutations working on generalized inputs first replace all remaining □ by an empty string.

**Algorithm 5:** Random selection of a generalized input, slice, token or string.

**Data:** `generalized` is the set of all previously generalized inputs, tokens and strings from the dictionary

**Result:** `rand` is a random generalized input, slice token or string

```
1 if random_coin() then
2     if random_coin() then
3         rand ← random_slice(generalized)
4     else
5         rand ← random_token_or_string(generalized)
6 else
7     rand ← random_generalized_input(generalized)
```

#### 4.3.2.1 Input Extension

The input extension mutation is inspired by the observation that—in highly structured input languages—often inputs are chains of syntactically well-formed statements. Therefore, we extend an generalized input by placing another randomly chosen generalized input, slice, token or string before and after the given one. This is described in Algorithm 6.

**Algorithm 6:** Overview of the input extension mutation.

**Data:** `input` is the current generalized input, `generalized` is the set of all previously generalized inputs, tokens and strings from the dictionary

```
1 rand ← random_generalized(generalized_inputs)
2 send_to_fuzzer(concat(input.content(), rand.content()))
3 send_to_fuzzer(concat(rand.content(), input.content()))
```

**Example 4.6.** *Assume that the current input is „`pprint 'aaaa'`" and its generalization is „`pprint '☐'`". Furthermore, assume that we randomly choose a previous generalization „`☐x=☐y+☐`". Then, we concretize their generalizations to „`pprint '$$'`" and „`x=y+`" by replacing remaining gaps with an empty string. Afterward, we concatenate them and obtain „`pprint '$$'x=y+`" and „`x=y+pprint '$$'`".*

#### 4.3.2.2 Recursive Replacement

The recursive replacement mutation recombines knowledge about the structured input language—that was obtained earlier in the fuzzing run—in a grammar-like manner. As illustrated in Algorithm 7, given a generalized input, we extend its beginning and end by ☐—if not yet present—such that we always can place other data before or behind the input. Afterward, we randomly select $n \in \{2, 4, 8, 16, 32, 64\}$ and perform the following operations $n$ times: First, we randomly select another generalized input, input slice, token or string. Then, we call `replace_random_gap` which replaces an arbitrary ☐ in the first generalized input by the chosen element. Furthermore, we enforce ☐ before and after the replacement such that these ☐ can be subject to further replacements. Finally, we concretize the mutated input and send it to the fuzzer. The recursive replacement mutator has a (comparatively) high likelihood of producing new structurally interesting inputs compared to mutations used by current coverage-guided fuzzers.

**Algorithm 7:** Overview of the recursive replacement mutation.

**Data:** `input` is the current generalized input, `generalized` is the set of all previously generalized inputs, tokens and strings from the dictionary

```
1 input ← pad_with_gaps(input)
2 for i ← 0 to random_power_of_two() do
3 |   rand ← random_generalized(generalized_inputs)
4 |   input ← replace_random_gap(input, rand)
5 send_to_fuzzer(input.content())
```

**Example 4.7.** *Assume that the current input is „*`pprint 'aaaa'`*“. We take its generalization „*`pprint '□'`*“ and extend it to „*`□pprint '□'□`*“. Furthermore, assume that we already generalized the inputs „*`if(x>1)□then □end`*“ and „*`□x=□y+□`*“. In a first mutation, we choose to replace the first □ with the slice „*`if(x>1)□`*“. We extend the slice to „*`□if(x>1)□`*“ and obtain „*`□if(x>1)□pprint '□'□`*“. Afterward, we choose to replace the third □ with „*`□x=□y+□`*“ and obtain „*`□if(x>1)□pprint '□x=□y+□'□`*“. In a final step, we replace the remaining □ with an empty string and obtain „*`if(x>1)pprint 'x=y+'`*“.*

#### 4.3.2.3 String Replacement

Keywords are important elements of structured input languages; changing a single keyword in an input can lead to completely different behavior. GRIMOIRE's string replacement mutation performs different forms of replacements, as described in Algorithm 8. Given an input, it locates all substrings in the input that match strings from the obtained dictionary and chooses one randomly. GRIMOIRE first selects a random occurrence of the matching substring and replaces it with a random string. In a second step, it replaces all occurrences of the substring with the same random string. Finally, the mutation sends both mutated inputs to the fuzzer. As an example, this mutation can be helpful to discover different methods of the same object by replacing a valid method call with different alternatives. Also, changing all occurrences of a substring allows us to perform more syntactically correct mutations, such as renaming of variables in the input.

**Example 4.8.** *Assume the „*`if(x>1)pprint 'x=y+'`*“ and that the strings „if“, „while“, „key“, „pprint“, „eval“, „+“, „=“ and „-“ are in the dictionary. Thus, the string replacement mutation can generate inputs such as „*`while(x>1)pprint 'x=y+'`*“, „*`if(x>1)eval 'x+y+'`*“ or „*`if(x>1)pprint 'x=y-'`*“. Furthermore, assume that the string „x“ is also in the dictionary. Then, the string replacement mutation can replace all occurrences of the variable „x“ in „*`if(x>1)pprint 'x=y+'`*“ and obtain „*`if(key>1)pprint 'key=y+'`*“.*

**Algorithm 8:** Overview of the string replacement mutation.

**Data:** `input` is the input string, `strings` is the provided dictionary obtained from the binary

```
1 sub ← find_random_substring(input, strings)
2 if sub then
3     rand ← random_string(strings)
4     data ← replace_random_instance(input, sub, rand)
5     send_to_fuzzer(data)
6     data ← replace_all_instances(input, sub, and)
7     send_to_fuzzer(data)
```

## 4.4 Implementation

To evaluate the algorithms introduced in this chapter, we built a prototype implementation of our design. Our implementation, called Grimoire, is based on Redqueen's [34] source code. This allows us to implement our techniques within a state-of-the-art fuzzing framework. Redqueen is applicable to both open and closed source targets running in user or kernel space, thus enabling us to target a wide variety of programs. While Redqueen is entirely focused on solving magic bytes and similar constructs which are local in nature (i. e., require only few bytes to change), Grimoire assumes that this kind of constraints can be solved by the underlying fuzzer. It uses global mutations (that change large parts of the input) based on the examples that the underlying fuzzer finds. Since our technique is merely based on common techniques implemented in coverage-guided fuzzers—for instance, access to the execution bitmap—it would be a feasible engineering task to adapt our approach to other current fuzzers, such as AFL.

More precisely, Grimoire is implemented as a set of patches to Redqueen. After finding new inputs, we apply the generalization instead of the minimization algorithm that was used by AFL and Redqueen. Additionally, we extended the havoc stage by structure-aware mutations as explained in Section 4.3. To prevent Grimoire from spending too much time in the generalization phase, we set a user-configurable upper bound; inputs whose length exceeds this bound are not be generalized. Per default, it is set to 16384 bytes. Overall, about 500 lines were written to implement the proposed algorithms.

To support reproducibility of our approach, we open source the fuzzing logic, especially the implementation of Grimoire as well as its interaction with Redqueen at `https://github.com/RUB-SysSec/grimoire`.

## 4.5 Experimental Evaluation

We evaluate our prototype implementation Grimoire to answer the following research questions.

**RQ 1** How does Grimoire compare to other state-of-the-art bug finding tools?

**RQ 2** Is our approach useful even when proper grammars are available?

**RQ 3** How does our approach improve the performance on targets that require highly structured inputs?

**RQ 4** How does our approach perform compared to other grammar inference techniques for the purpose of fuzzing?

**RQ 5** How do our mutators impact fuzzing performance?

**RQ 6** Can Grimoire identify new bugs in real-world applications?

To answer these questions, we perform three individual experiments. First, we evaluate the coverage produced by various fuzzers on a set of real-world target programs. In the

second experiment, we analyze how our techniques can be combined with grammar-based fuzzers for mutual improvements. Finally, we use GRIMOIRE to uncover a set of vulnerabilities in real-world target applications.

### 4.5.1 Measurement Setup

All experiments are performed on an Ubuntu Server 16.04.2 LTS with an Intel i7-6700 processor with 4 cores and 24 GiB of RAM. Each tool is evaluated over 12 runs for 48 hours to obtain statistically meaningful results. In addition to other statistics, we also measure the effect size by calculating the difference in the median of the number of basic blocks found in each run. Additionally, we perform a Mann Whitney U test (using `scipy` 1.0 [125]) and report the resulting $p$-values. All experiments are performed with the tool being pinned to a dedicated CPU in single-threaded mode. Tools other than GRIMOIRE and REDQUEEN require source-code access; we use the fast *clang*-based instrumentation in these cases. Additionally, to ensure a fair evaluation, we provide each fuzzer with a dictionary containing the strings found inside of the target binary. In all cases, except NAUTILUS (which crashed on larger bitmaps), we increase the bitmap size from $2^{16}$ to $2^{19}$. This is necessary since we observe more collisions in the global coverage map for large targets which causes the fuzzer to discard new coverage. For example, in `SQLite` (1.9 MiB), 14% of the global coverage map entries collide [233]. Since we deal with even larger binaries such as `PHP` which is nearly 19 MiB, the bitmap fills up quickly. Based on our empirical evaluation, we observed that $2^{19}$ is the smallest sufficient size that works for all of our target binaries.

Furthermore, we disable the so-called *deterministic stage* [233]. This is motivated by the observation that these deterministic mutations are not suited to find new coverage considering the nature of highly structured inputs. Finally—if not stated otherwise—we use the same uninformed seed that the authors of REDQUEEN used for their experiments: `"ABC...XYZabc...xyz012...789!"$...~+*"`.

As noted by Aschermann et al. [34], there are various definitions of a basic block. Fuzzers such as AFL change the number of basic blocks in a program. Thus, to enable a fair comparison in our experiments, we measure the coverage produced by each fuzzer on the same uninstrumented binary. Therefore, the numbers of basic blocks found and reported in this chapter might differ from other papers. However, they are consistent within all of our experiments.

For our experiments, we select a diverse set of target programs. We use four scripting language interpreters (`mruby-1.4.1` [155], `php-7.3.0` [210], `lua-5.3.5` [120] and `JavaScriptCore, commit „f1312“` [32]) a compiler (`tcc-0.9.27` [40]), an assembler (`nasm-2.14.02` [209]), a database (`sqlite-3.25` [114]), a parser (`libxml-2.9.8` [212]) and an SMT solver (`boolector-3.0.1` [165]). We select these four scripting language interpreters so that we can directly compare the results to NAUTILUS. Note that our choice of targets is additionally governed by architectural limitations of REDQUEEN which GRIMOIRE is based on. REDQUEEN uses Virtual Machine Introspection (VMI) to transfer the target binary—including all of its dependencies—into the Virtual Machine (VM). The maximum transfer size using VMI in REDQUEEN is set to 64 MiB. Programs

such as Python [175], GCC [90], Clang [149], V8 [97] and SpiderMonkey [160] exceed our VMI limitation; thus, we can not evaluate them. We select an alternative set of target binaries that are large enough but at the same time do not exceed our 64 MiB transfer size limit. Hence, we choose JavaScriptCore over V8 and SpiderMonkey, mruby over ruby and TCC over GCC or Clang. Finally, we tried to evaluate GRIMOIRE with ChakraCore [157]. However, ChakraCore fails to start inside of the REDQUEEN Virtual Machine for unknown reasons. Still, GRIMOIRE performs well on large targets such as JavaScriptCore and PHP.

### 4.5.2 State-of-the-Art Bug Finding Tools

To answer **RQ 1**, we perform 12 runs on eight targets using GRIMOIRE and four state-of-the-art bug finding tools. We choose AFL (version 2.52b) because it is a well-known fuzzer and a good baseline for our evaluation. We select QSYM (commit „6f00c3d") and ANGORA (commit „6ff81c6"), two state-of-the-art hybrid fuzzers which employ different program analysis techniques, namely symbolic execution and taint tracking. Finally, we choose REDQUEEN as a state-of-the-art coverage-guided fuzzer, which is also the baseline of GRIMOIRE. As a consequence, we are able to directly observe the improvements of our method. Note that we could not compile libxml for ANGORA instrumentation. Therefore, ANGORA is missing in the libxml plot.

**Table 4.1:** Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against the best alternative. The effect size is the difference of the medians in basic blocks. In most experiments, the effect size is relevant and the changes are highly significant: it is typically multiple orders of magnitude smaller than the usual bound of p < 5.0E-02 (bold).

| Target | Best Alternative | Effect Size ($\Delta = \bar{A} - \bar{B}$) | Effect Size in % of Best | p-value |
|---|---|---|---|---|
| mruby | ANGORA | 3685 | 19.3% | **1.8E-05** |
| TCC | REDQUEEN | 1952 | 22.6% | **7.8E-05** |
| PHP | REDQUEEN | 11238 | 31.6% | **1.8E-05** |
| Boolector | AFL | 7671 | 43.9% | **1.8E-05** |
| Lua | ANGORA | -478 | -8.2% | **4.5E-04** |
| libxml | AFL | 308 | 3.4% | **1.8E-02** |
| SQLite | ANGORA | 4846 | 26.8% | **1.8E-05** |
| NASM | ANGORA | 272 | 2.9% | 9.7E-02 |

The results of our coverage measurements are shown in Figure 4.2. As we can see, in all cases GRIMOIRE provides a significant advantage over the baseline (unmodified REDQUEEN). Surprisingly, in most cases, neither ANGORA, REDQUEEN, nor QSYM seem to have a significant edge over plain AFL. This can be explained by the fact that REDQUEEN and ANGORA mostly aim to overcome certain „magic byte" fuzzing roadblocks. Similarly, QSYM is also effective to solve these roadblocks. Since we provide a dictionary with strings from the target binary to each fuzzer, these roadblocks become much less common. Thus, the techniques introduced in ANGORA, REDQUEEN and

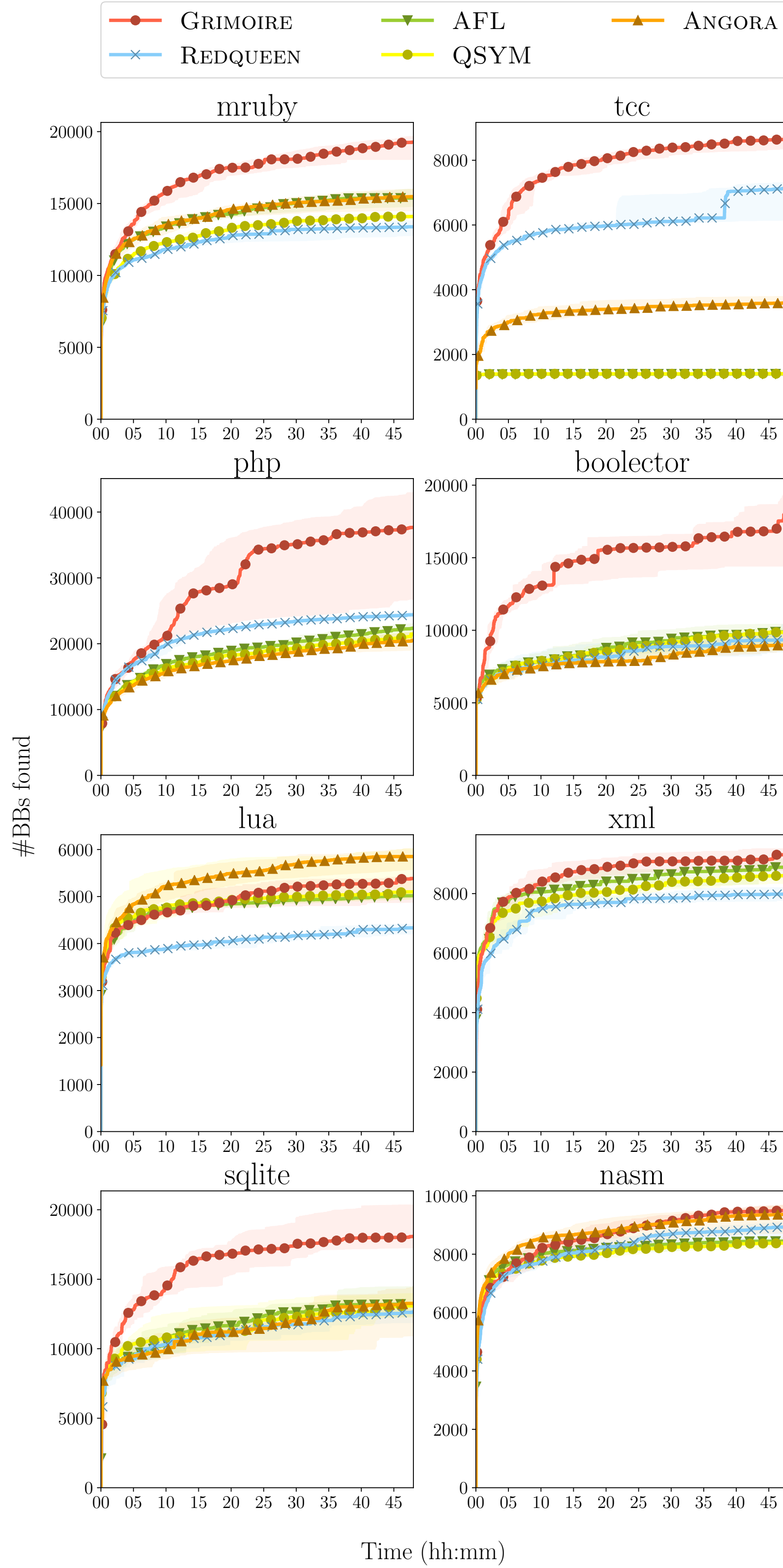


**Figure 4.2:** The coverage (in basic blocks) produced by various tools over 12 runs for 48h on various targets. Displayed are the median and the 66.7% intervals.

QSym are less relevant given the seeds provided to the fuzzers. However, in the case of TCC, we can observe that providing the strings dictionary does not help AFL. Therefore, we believe that Angora and Redqueen find strings that are not part of the dictionary and thus outperform AFL.

| Target | Best Coverage (#BBS / %) | Fuzzer | Mean (%) | Median (%) | **Median (#BBs)** | Std Deviation | Skewness | Kurtosis |
|---|---|---|---|---|---|---|---|---|
| mruby | 20258 / 70.5% | Grimoire | 66.1% | 66.6% | **19 137** | 4.55 | −0.54 | −0.76 |
| | | AFL | 53.7% | 53.4% | **15 355** | 4.28 | 0.14 | −0.27 |
| | | Angora | 53.3% | 53.8% | **15 452** | 4.87 | 0.17 | −0.96 |
| | | QSym | 49.2% | 49.0% | **14 084** | 2.20 | 0.33 | 0.95 |
| | | Redqueen | 45.9% | 46.4% | **13 339** | 4.64 | −0.98 | 0.05 |
| TCC | 9211 / 77.6% | Grimoire | 71.8% | 72.9% | **8647** | 5.71 | −1.89 | 3.68 |
| | | AFL | 11.8% | 11.8% | **1397** | 3.80 | 1.27 | 1.14 |
| | | Angora | 31.0% | 30.3% | **3600** | 6.51 | 1.01 | 0.06 |
| | | QSym | 11.9% | 11.8% | **1403** | 3.26 | 1.52 | 2.59 |
| | | Redqueen | 56.7% | 56.4% | **6695** | 8.13 | 0.03 | −1.93 |
| PHP | 46805 / 27.9% | Grimoire | 20.8% | 21.2% | **35 606** | 20.26 | 0.12 | −1.38 |
| | | AFL | 13.2% | 13.3% | **22 323** | 3.64 | −0.09 | −0.96 |
| | | Angora | 12.1% | 12.2% | **20 501** | 6.39 | −0.37 | −0.58 |
| | | QSym | 12.7% | 12.7% | **21 276** | 2.60 | 0.22 | −1.11 |
| | | Redqueen | 14.5% | 14.5% | **24 367** | 1.87 | 0.37 | −0.83 |
| Boolector | 23207 / 33.1% | Grimoire | 25.2% | 24.9% | **17 461** | 16.77 | 0.51 | −0.65 |
| | | AFL | 14.0% | 14.0% | **9790** | 7.46 | 0.30 | −0.57 |
| | | Angora | 13.2% | 12.8% | **8986** | 9.20 | 0.79 | −0.17 |
| | | QSym | 13.7% | 14.0% | **9782** | 6.94 | −0.39 | −1.24 |
| | | Redqueen | 13.3% | 13.3% | **9305** | 9.63 | 0.21 | −1.23 |
| Lua | 6205 / 64.1% | Grimoire | 54.4% | 55.2% | **5339** | 6.47 | 0.20 | −0.73 |
| | | AFL | 51.9% | 51.9% | **5016** | 1.61 | 0.84 | −0.15 |
| | | Angora | 59.9% | 60.1% | **5817** | 2.96 | 0.05 | −1.39 |
| | | QSym | 54.8% | 52.6% | **5091** | 9.52 | 1.07 | −0.65 |
| | | Redqueen | 44.5% | 44.4% | **4299** | 2.30 | −0.30 | −1.19 |
| libxml | 10437 / 13.2% | Grimoire | 11.7% | 11.6% | **9190** | 5.52 | 0.98 | 0.02 |
| | | AFL | 11.1% | 11.2% | **8881** | 3.40 | −0.39 | −0.92 |
| | | Angora | 0.0% | 0.0% | **0** | nan | 0.00 | −3.00 |
| | | QSym | 10.8% | 10.8% | **8598** | 2.36 | 0.95 | 1.45 |
| | | Redqueen | 10.1% | 10.1% | **7979** | 3.72 | 0.72 | −0.25 |
| SQLite | 22031 / 57.1% | Grimoire | 48.6% | 46.8% | **18 064** | 9.25 | 0.80 | −0.72 |
| | | AFL | 34.6% | 33.9% | **13 072** | 10.02 | 0.60 | −0.34 |
| | | Angora | 33.1% | 34.2% | **13 218** | 12.12 | −0.30 | −1.05 |
| | | QSym | 33.4% | 33.6% | **12 988** | 10.91 | −0.33 | −0.18 |
| | | Redqueen | 32.3% | 32.6% | **12 599** | 4.77 | 0.18 | −0.21 |
| NASM | 10015 / 51.1% | Grimoire | 47.7% | 48.4% | **9483** | 7.58 | −2.58 | 5.67 |
| | | AFL | 43.2% | 43.0% | **8442** | 1.68 | 1.07 | 1.09 |
| | | Angora | 46.9% | 47.0% | **9211** | 5.27 | 0.06 | −1.19 |
| | | QSym | 42.1% | 42.6% | **8357** | 4.72 | −1.49 | 2.40 |
| | | Redqueen | 44.9% | 45.5% | **8928** | 4.21 | −0.20 | −0.89 |

**Table 4.2:** Statistics on basic block coverage for tested fuzzers. In the column „Best Coverage", we provide the highest number of basic blocks a run found and the percentage relative to the number of basic blocks obtained from IDA Pro [113].

A complete statistical description of the results is given in Table 4.2. We perform a confirmatory statistical analysis on the results, as shown in Table 4.1. The results show that in all but two cases (Lua and NASM), Grimoire offers relevant and significant improvements over all state-of-the-art alternatives. On average, it finds nearly 20% more coverage than the second best alternative.

Lua accepts both source files (text) as well as byte code. Grimoire can only make effective mutations in the domain of language features and not the bytecode. However, other fuzzers can perform on both; this is why Angora outperforms Grimoire on this target. It is worth mentioning that Grimoire outperforms Redqueen, the baseline on top of which our approach is implemented.

To partially answer **RQ 1**, we showed that in terms of *code coverage*, GRIMOIRE outperforms other state-of-the-art bug finding tools (in most cases). Second, to answer **RQ 3**, we demonstrated that GRIMOIRE significantly improves the performance on targets with highly structured inputs when compared to our baseline (REDQUEEN).

### 4.5.3 Structured Gray-box Fuzzers

Generally, we expect structured and grammar-based fuzzers to have an edge over grammar inference fuzzers like GRIMOIRE since they have access to a manually crafted grammar. To quantify this advantage, we evaluate GRIMOIRE against current structured gray-box fuzzers. To this end, we choose NAUTILUS (commit „dd3554a“), a state-of-the-art coverage-guided fuzzer, since it can fuzz a wide variety of targets if provided with a hand-written grammar. We evaluate on the targets used in NAUTILUS' experiments, `mruby`, `PHP` and `Lua`, as their grammars are available. Unfortunately, GRIMOIRE is not capable of running `ChakraCore`, the fourth target NAUTILUS was evaluated on; thus, we replace it by `JavaScriptCore` and use NAUTILUS' JavaScript grammar. We observed that the original version of NAUTILUS had some timeout problems during fuzzing where the timeout detection did not work properly. We fixed this for our evaluation.

For each of the four targets, we perform an experiment with the same setup as the first experiment (again, 12 runs for 48 hours). The results are shown in Figure 4.3. As expected, our completely automated method is defeated in most cases by NAUTILUS since it uses manually fine-tuned grammars. Surprisingly, in the case of `mruby`, we find that GRIMOIRE is able to outperform even NAUTILUS.

To evaluate whether GRIMOIRE is still useful in scenarios where a grammar is available, we perform another experiment. We extract the corpus produced by NAUTILUS after half of the time (i. e., 24 hours) and continue to use GRIMOIRE for another 24 hours using this seed corpus. For these *incremental runs*, we reduce GRIMOIRE's upper bound for input generalization to $2,048$ bytes; otherwise, our fuzzer would mainly spend time in the generalization phase since NAUTILUS produces very large inputs. The results are displayed in Figure 4.3 (incremental). This experiment demonstrates that even despite manual fine-tuning, the grammar often contains blind spots, where an automated approach such as ours can infer the implicit structure which the program expects. This structure may be quite different from the specified grammar. As Figure 4.3 shows, by using the corpus created by NAUTILUS, GRIMOIRE surpasses NAUTILUS individually in all cases (**RQ 2**). A confirmatory statistical analysis of the results is presented in Table 4.3. In three cases, GRIMOIRE is able to improve upon hand written grammars by nearly 10%.

Additionally, we intended to compare GRIMOIRE against CODEALCHEMIST and JSFUNFUZZ, two other state-of-the art grammar-based fuzzers which specialize on JavaScript engines. Although these two fuzzers are not coverage-guided—making a fair evaluation challenging—we consider the comparison of specialized JavaScript grammar-based fuzzers to general-purpose grammar-based fuzzers as interesting. Unfortunately, JSFUNFUZZ was not working with `JavaScriptCore` out of the box as it is specifically tailored to `SpiderMonkey`. Since it requires significant modifications to run on `JavaScriptCore`,

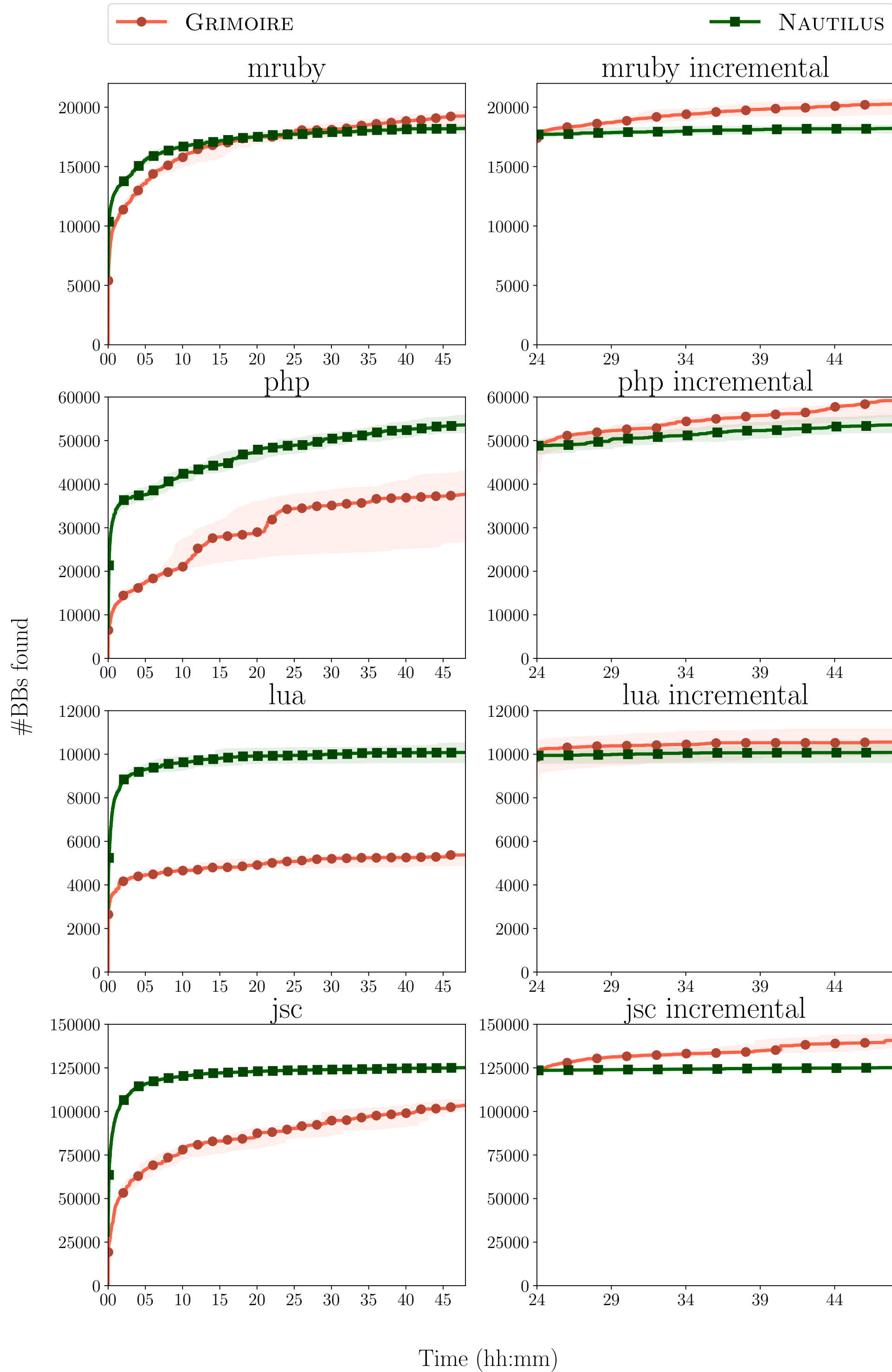


**Figure 4.3:** The coverage (in basic blocks) produced by Grimoire and Nautilus (using the hand written grammars of the authors of Nautilus) over 12 runs at 48 h on various targets. The incremental plots show how running Nautilus for 48h compares to running Nautilus for the first 24h and then continue fuzzing for 24h with Grimoire. Displayed are the median and the 66.7% confidence interval.

**Table 4.3:** Confirmatory data analysis of our experiment. We compare the coverage produced by GRIMOIRE against NAUTILUS with hand written grammars. The effect size is the difference of the medians in basic blocks in the incremental experiment. In three experiments, the effect size is relevant and the changes are highly significant (marked bold, $p < 5.0E-02$). Note that we abbreviate `JavaScriptCore` with `JSC`.

| Target | Best Alternative | Effect Size ($\Delta = \bar{A} - \bar{B}$) | Effect Size in % of Best | p-value |
|---|---|---|---|---|
| `mruby` | NAUTILUS | 2025 | 10.0% | **1.8E-05** |
| `Lua` | NAUTILUS | 553 | 5.2% | 5.0E-02 |
| `PHP` | NAUTILUS | 5465 | 9.3% | **3.6E-03** |
| `JSC` | NAUTILUS | 15445 | 11.0% | **1.8E-05** |

we considered the required engineering effort to be out of scope for this work. On the other hand, CODEALCHEMIST requires an extensive seed corpus of up to 60,000 valid JavaScript files—which were not released together with the source files. We tried to replicate the seed corpus as described by the authors of CODEALCHEMIST. However, despite the authors' kind help, we were unable to run CODEALCHEMIST with our corpus.

Overall, these experiments confirm our assumption that structured gray-box fuzzers such as NAUTILUS have an edge over grammar inference fuzzers like GRIMOIRE. However, deploying our approach on top of a grammar-based fuzzer (incremental runs) increases code coverage. Therefore, we partially respond to **RQ 1** and provide an answer to **RQ 2** by stating that GRIMOIRE is a valuable addition to current fuzzing techniques.

### 4.5.4 Grammar Inference Techniques

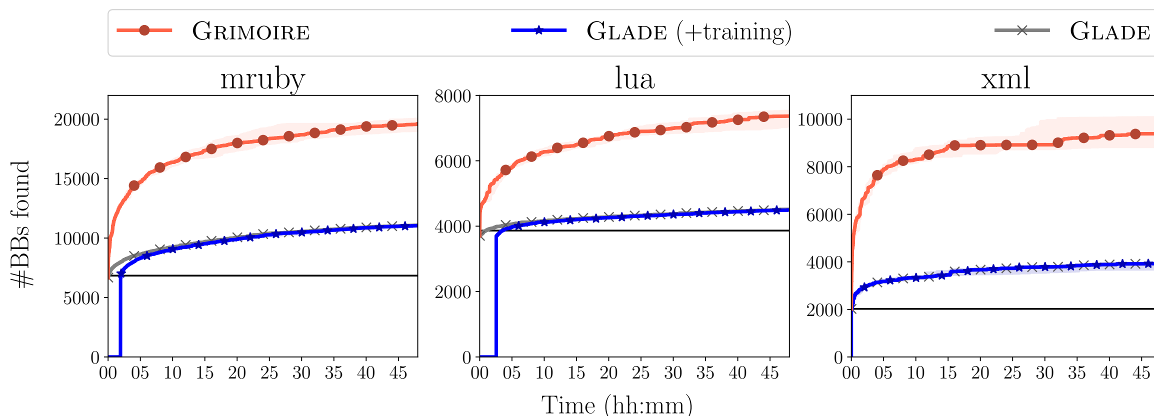


**Figure 4.4:** Comparing GRIMOIRE against GLADE (median and 66.7% interval). In the plot for GLADE +Training, we include the training time that glade used. For comparison, we also include plots where we omit the training time. The horizontal bar displays the coverage produced by the seed corpus that GLADE used during training.

To answer **RQ 4**, we compare our approach to other grammar inference techniques in the context of fuzzing. Existing work in this field includes Glade, Autogram and Pygmalion. However, since Pygmalion targets only Python and Autogram only Java programs, we cannot evaluate them as Grimoire only supports targets that can be traced with Intel-PT (since Redqueen heavily depends on it).

Therefore, for this evaluation, we use Glade (commit „b9ef32e“), a state-of-the-art grammar inference tool. It operates in two stages. Given a program as black-box oracle as well as a corpus of valid input samples, it learns a grammar in the first stage. In the second stage, Glade uses this grammar to produce inputs that can be used for fuzzing. Glade does not generate a continuous stream of inputs, hence we modified it to provide such capability. We then use these inputs to measure the coverage achieved by Glade in comparison to Grimoire. Note that due to the excessive amount of inputs produced by Glade, we use a corpus minimization tool—`afl-cmin`—to identify and remove redundant inputs before measuring the coverage [233].

Note, we have to extend Glade for each target that is not natively supported and must manually create a valid seed corpus. For this reason, we restrict ourselves to the three targets `libxml`, `mruby` and `Lua`. From these, `libxml` is the only one that was also used in Glade's evaluation. Therefore, we are able to re-use their provided corpus for this target. We choose the other two since we want to achieve comparability with regards to previous experiments.

To allow for a fair comparison, we provide the same corpus to Grimoire. Again, we repeat all experiments 12 times for 48 hours each. The results of this comparison are depicted in Figure 4.4. Note that this figure includes two different experiments of Glade. In the first experiment, we include the time Glade spent on training into the measurement while for the second measurement, Glade is provided the advantage of concluding the training stage before measurement is started for the fuzzing process. As can be seen in Figure 4.4, Grimoire significantly outperforms Glade on all targets for both experiments. Similar to earlier experiments, we perform a confirmatory statistical analysis. The results are displayed in Table 4.4; they are in all cases relevant and statistically significant. If we consider only the new coverage found (beyond what is already contained in the training set), we are able to outperform Glade by factors from two to five. We therefore conclude in response to **RQ 4** that we significantly exceed comparative grammar inference approaches in the context of fuzzing.

We designed another experiment to evaluate whether Glade's automatically inferred grammar can be used for Nautilus and how it performs compared to hand written grammars. However, Glade does not use the grammar directly but remembers how the grammar was produced from the provided test cases and uses the grammar only to apply local mutations to the input. Unfortunately, as a consequence, their grammar contains multiple unproductive rules, thus preventing their usage in Nautilus.

### 4.5.5 Mutations Statistic

During the aforementioned experiments, we also collected various statistics on how effective different mutators are. We measured how much time was spent using Gri-

**Table 4.4:** Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against GLADE. The effect size is the difference of the medians in basic blocks. In all experiments, the effect size is relevant and the changes are highly significant: it is multiple orders of magnitude smaller than the usual bound of $p < 5.0E\text{-}02$ (bold).

| Target | Best Alternative | Effect Size ($\Delta = \bar{A} - \bar{B}$) | Effect Size in % of Best | p-value |
|---|---|---|---|---|
| mruby | GLADE | 8546 | 43.6% | **9.1E-05** |
| Lua | GLADE | 2775 | 38.1% | **9.1E-05** |
| libxml | GLADE | 5213 | 57.2% | **9.1E-05** |

MOIRE's different mutation strategies as well as how many of the inputs were found by each strategy. This allows us to rank mutation strategies based on the number of new paths found per time used. The strategies include a havoc stage, REDQUEEN's Input-to-State-based mutation stage and our structural mutation stage. The times for our structural mutators include the generalization process (including the necessary minimization that also benefits the other mutators).

As Table 4.5 shows, our structural mutators are competitive with other mutators, which answers **RQ 5**. As the coverage results in Figure 4.2 show, the mutators are also able to uncover paths that would not have been found otherwise.

### 4.5.6 Real-World Bugs

We use GRIMOIRE on a set of different targets to observe whether it is able to uncover previously unknown bugs (**RQ 6**). To this end, we manually triaged bugs found during our evaluation. As illustrated in Table 4.6, GRIMOIRE found more bugs than all other tools in the evaluation combined. We responsibly disclosed all of them to the vendors. For these, 11 CVEs were assigned. Note that we found a large number of bugs that did not lead to assigned CVEs. This is partially because projects such as PHP do not consider invalid inputs as security relevant, even when custom scripts can trigger memory corruption. We conclude **RQ 6** by finding that GRIMOIRE is indeed able to uncover novel bugs in real-world applications.

## 4.6 Discussion

The methods introduced in this chapter produce significant performance gains on targets that expect highly structured inputs without requiring any expert knowledge or manual work. As we have shown, GRIMOIRE can also be used to support grammar-based fuzzers with well-tuned grammars but cannot outperform them on their own. In contrast to similar methods, our approach does not rely on complex primitives such as symbolic execution or taint tracking. Therefore, it can easily be integrated into existing fuzzers. Additionally, since GRIMOIRE is based on REDQUEEN, it can be used on a wide variety of binary-only targets, ranging from userland programs to operating system kernels.

**Table 4.5:** Statistics for each of Grimoire's mutation strategies (i. e., our structured mutations, Redqueen's Input-to-State-based mutations and havoc). For every target evaluated we list the total number of inputs found by a mutation, the time spent on this strategy and the ratio of inputs found per minute.

| Mutation | Target | #Inputs | Time Spent (min) | #Inputs/Min |
|---|---|---|---|---|
| Structured | `mruby` | 9040 | 1531.18 | 5.90 |
| | `PHP` | 27063 | 2467.17 | **10.97** |
| | `Lua` | 2849 | 2064.49 | 1.38 |
| | `SQLite` | 5933 | 1325.26 | **4.48** |
| | `TCC` | 6618 | 2271.03 | 2.91 |
| | `Boolector` | 3438 | 2399.85 | 1.43 |
| | `libxml` | 4883 | 2001.38 | 2.44 |
| | `NASM` | 12696 | 1955.42 | **6.49** |
| | `JavaScriptCore` | 38465 | 2460.95 | **15.63** |
| Input-to-State | `mruby` | 814 | 268.23 | 3.03 |
| | `PHP` | 902 | 111.46 | 8.09 |
| | `Lua` | 530 | 307.12 | 1.73 |
| | `SQLite` | 603 | 768.72 | 0.78 |
| | `TCC` | 1020 | 118.23 | **8.63** |
| | `Boolector` | 325 | 102.87 | **3.16** |
| | `libxml` | 967 | 359.03 | 2.69 |
| | `NASM` | 1329 | 213.84 | 6.22 |
| | `JavaScriptCore` | 400 | 82.76 | 4.83 |
| Havoc | `mruby` | 2010 | 339.03 | **5.93** |
| | `PHP` | 2546 | 278.21 | 9.15 |
| | `Lua` | 1684 | 492.99 | **3.42** |
| | `SQLite` | 1827 | 742.13 | 2.46 |
| | `TCC` | 2514 | 484.73 | 5.19 |
| | `Boolector` | 956 | 373.85 | 2.56 |
| | `libxml` | 2173 | 504.86 | **4.30** |
| | `NASM` | 2876 | 678.59 | 4.24 |
| | `JavaScriptCore` | 3800 | 279.62 | 13.59 |

Despite all advantages, our approach has significant difficulties with more syntactically complex constructs, such as matching the ID of opening and closing tags in XML or identifying variable constructs in scripting languages. For instance, while Grimoire is able to produce nested inputs such as „`<a><a><a>FOO</a></a></a>`“, it struggles to generalize „`<a>□</a>`“ to the more unified representation „<$\boxed{A}$>□</$\boxed{B}$>“ with the constraint $A = B$. A solution for such complex constructs could be the following generalization heuristic: (i) First, we record the new coverage for the current input. (ii) We then change only a single occurrence of a substring in our input and record its new coverage. For instance, consider that we replace a single occurrence of „`a`“ by „`b`“ in „`<a><a><a>FOO</a></a></a>`“ and obtain „`<b><a><a>FOO</a></a></a>`“. This

**Table 4.6:** Overview of submitted bugs and CVEs. Fuzzers which did not find the bug during our evaluation are denoted by ✗, while those who did are marked by ✓. We indicate targets not evaluated by a specific fuzzer with '-'. We abbreviate Use-After-Free (UAF), Out-of-Bounds (OOB) and Buffer Overflow (BO).

| Target | CVE | Type | Grimoire | Redqueen | AFL | QSym | Angora | Nautilus |
|---|---|---|---|---|---|---|---|---|
| PHP | | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| PHP | | OOB-read | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| PHP | | OOB-read | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| PHP | | OOB-read | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| TCC | 2018-20374 | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| TCC | 2018-20375 | OOB-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| TCC | 2018-20376 | OOB-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| TCC | 2019-12495 | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| TCC | 2019-9754 | OOB-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| TCC | | OOB-write | ✗ | ✓ | ✗ | ✗ | ✗ | - |
| Boolector | 2019-7559 | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| Boolector | 2019-7560 | UAF-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| NASM | 2019-8343 | UAF-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| NASM | | OOB-write | ✓ | ✗ | ✓ | ✗ | ✗ | - |
| NASM | | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| NASM | | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| NASM | | OOB-write | ✓ | ✗ | ✓ | ✗ | ✗ | - |
| NASM | | OOB-write | ✗ | ✗ | ✓ | ✗ | ✗ | - |
| gnuplot | 2018-19490 | BO | ✓ | - | - | - | - | - |
| gnuplot | 2018-19491 | BO | ✓ | - | - | - | - | - |
| gnuplot | 2018-19492 | BO | ✓ | - | - | - | - | - |

change results in an invalid XML tag which leads to different coverage compared to the one observed in (i). (iii) Finally, we change multiple instances of the same substring and compare the new coverage of the modified input with the one obtained in (i). If we achieved the same new coverage in (iii) and (i), we can assume that the modified instances of the same substring are related to each other. For example, we replace multiple occurrences of „a" with „b" and obtain „<b><a><a>FOO</a></a></b>". In this example, the coverage is the same as for the original input since the XML remains syntactically correct.

Similarly, our generalization approach might be too coarse in many places. Obtaining more precise rules would help uncovering deeper parts of the target application in cases where multiple valid statements have to be produced. Consider, for instance, a scripting language interpreter such as the ones used in our evaluation. Certain operations might

require a number of constructors to be successfully called. For example, it might be necessary to get a valid path object to obtain a file object that can finally be used to perform a read operation. A more precise representation would be highly useful in such cases. One could try to infer whether a combination is „valid“ by checking if the combination of two inputs exercises the combination of the new coverage introduced by both inputs. For instance, assume that input „a□b“ triggers the coverage bytes 7 and 10 and that input „□=□“ triggers coverage byte 20. Then, a combination of these two inputs such as „□a□=□b“ could trigger the coverage bytes 7, 10 and 20. Using this information, it might be possible to infer more precise grammar descriptions and thus generate inputs that are closer to the target's semantics than it is currently possible in Grimoire. While this approach would most likely further reduce the gap between hand-written grammars and inferred grammars, well-designed hand-written grammars will always have an edge over fuzzers with no prior knowledge: any kind of inference algorithm first needs to uncover structures before the obtained knowledge can be used. A grammar-based fuzzer has no such disadvantage. If available, human input can improve the results of grammar inference or steer its direction. An analyst can provide a partial grammar to make the grammar-fuzzer focus on a specific interesting area and avoid exploring paths that are unlikely to contain bugs. Therefore, Grimoire is useful if the grammar is unknown or under-specified but cannot be considered a full replacement for grammar-based fuzzers.

## 4.7 Related Work

A significant number of approaches to improve the performance of different fuzzing strategies has been proposed over time. Early on, fuzzers typically did not observe the inner workings of the target application, yet different approaches were proposed to improve various aspects of fuzzers: different mutation strategies were evaluated [78, 111], the process of selecting and scheduling of seed inputs was analyzed [57, 180, 217] and, in some cases, even learned language models were used to improve the effectiveness of fuzzing [95, 107]. After the publication of AFL [231], the research focus shifted towards coverage-guided fuzzing techniques. Similarly to the previous work on blind fuzzing, each individual component of AFL was put under scrutiny. For example, AFLFast [46] and AFLGo [47] proposed scheduling mechanisms that are better suited to some circumstances. Both, CollAFL [85] and InsTrim [119], enhanced the way in which coverage is generated and stored to reduce the amount of memory needed. Other publications improved the ways in which coverage feedback is collected [96, 194, 206, 222]. To advance the ability of fuzzers to overcome constraints that are hard to guess, a wide array of techniques were proposed. Commonly, different forms of symbolic execution are used to solve these challenging instances [53, 56]. In most of these cases, a restricted version of symbolic execution (concolic execution) is used [92–94, 106, 205, 215]. To further improve upon these techniques, DigFuzz [235] provides a better scheduling for inputs to the symbolic executor. Sometimes, instead of using these heavy-weight primitives, more lightweight techniques such as taint tracking [59, 86, 106, 179], patches [34, 76, 171, 215] or instrumentation [34, 142] are used to overcome the same hurdles.

While these improvements generally work very well for binary file formats, many modern target programs work with highly structured data. To target these programs, generational fuzzing is typically used. In such scenarios, the user can often provide a grammar. In most cases, fuzzers based on this technique are blind fuzzers [78, 116, 167, 186, 229].

Recent projects such as AFLSmart [173], Nautilus [33] and Zest [169] combined the ideas of generational fuzzing with coverage guidance. CodeAlchemist [108] even ventures beyond syntactical correctness. To find novel bugs in mature JavaScript interpreters, it tries to automatically craft syntactically and semantically valid inputs by recombining input fragments based on inferred types of variables. All of these approaches require a good format specification and—in some cases—good seed corpora. CodeAlchemist even needs access to a specialized interpreter for the target language to trace and infer type annotations. In contrast, our approach has no such preconditions and is thus easily integrable into most fuzzers.

Finally, to alleviate some of the disadvantages that the mentioned grammar-based strategies have, multiple approaches were developed to automatically infer grammars for given programs. Glade [38] can systematically learn an approximation to the context-free grammars parsed by a program. To learn the grammar, it needs an oracle that can answer whether a given input is valid or not as well as a small set of valid inputs. Similar techniques are used by Pygmalion [98] and Autogram [118]. However, both techniques directly learn from the target application without requiring a modified version of the target. Autogram still needs a large set of inputs to trace, while Pygmalion can infer grammars based solely on the target application. Additionally, both approaches require complex analysis passes and even symbolic execution to produce grammars. These techniques cannot easily be scaled to large binary applications. Finally, all three approaches are computationally expensive.

## 4.8 Conclusion

We developed and demonstrated the first fully automatic algorithm that integrates structure-aware mutations into the fuzzing process. In contrast to other approaches, we need no additional modifications or assumptions about the target application. We demonstrated the capabilities of our approach by evaluating our implementation called Grimoire against various state-of-the-art coverage-guided fuzzers. Our evaluation shows that we outperform other coverage-guided fuzzers both in terms of coverage and the number of bugs found. From this observation, we conclude that it is possible to significantly improve the fuzzing process in the absence of program input specifications. Furthermore, we conclude that even when a program input specification is available, our approach is still useful when it is combined with a generational fuzzer.

# Chapter 5

# Predicate Synthesis to Automate Root Cause Explanation

## 5.1 Introduction

Fuzz testing (short: *fuzzing*) is a powerful software testing technique that, especially in recent years, gained a lot of traction both in industry and academia [33, 34, 59, 171, 179, 205, 231]. In essence, fuzzing capitalizes on a high throughput of inputs that are successively modified to uncover different paths within a target program. The recent focus on new fuzzing methods has produced a myriad of crashes for software systems, sometimes overwhelming the developers who are tasked with fixing them [29, 189]. In many cases, finding a new crashing input has become the easy and fully automated part, while triaging crashes remains a manual, labor-intensive effort. This effort is mostly spent on identifying the actual origin of a crash [232]. The situation is worsened as fuzzing campaigns often result in a large number of crashing inputs, even if only one actual bug is found: a fuzzer can identify multiple paths to a crash, while the fault is always the same. Thus, an analyst has to investigate an inflated number of potential bugs. Consequently, developers lose time on known bugs that could be spent on fixing others.

To reduce the influx of crashes mapping to the same bug, analysts attempt to *bucket* such inputs. Informally speaking, bucketing groups crashing inputs according to some metric—often coverage or hashes of the call stack—into equivalence classes. Typically, it is assumed that analyzing one input from each class is sufficient. However, recent experiments have shown that common bucketing schemes produce far too many buckets and, even worse, cluster distinct bugs into the same bucket [134]. Even if there are only a few inputs to investigate, an analyst still faces another challenge: Understanding the reasons why a given input leads to a crash. Often, the real cause of a crash—referred to as *root cause*—is not located at the point the program crashes; instead, it might be far earlier in the program's execution flow. Therefore, an analyst needs to analyze the path from the crashing location backward to find the root cause, which requires significant effort.

Consider, for example, a type confusion bug: a pointer to an object of type `A` is used in a place where a pointer to `B` is expected. If a field of `B` is accessed, an invalid access on a subsection of `A` can result. If the structures are not compatible (e. g., `A` contains a string where a pointer is expected by `B`), this can cause memory corruption. In this case, the crashing location is most likely not the root cause of the fault, as the invariant „points to an instance of `B`" is violated in a different spot. The code that creates the object of type `A` is also most likely correct. Instead, the particular control flow that makes a value from type `A` end up in `B`'s place is at fault.

In a naive approach, an analyst could inspect stack and register values with a debugger. Starting from the crash, they can manually backtrace the execution to the root cause. Using state-of-the-art sanitizers such as the ASAN family [197] may detect illegal memory accesses closer to the root cause. In our example, the manual analysis would start at the crashing location, while ASAN would detect the location where the memory corruption occurred. Still, the analyst has to manually recognize the type confusion as the root cause—a complicated task since most code involved is behaving correctly.

More involved approaches such as POMP [221], RETRACER [68], REPT [69] and DEEPVSA [105] use automated reverse execution and backward taint analysis. These are particularly useful if the crash is not reproducible. For example, REPT and RETRACER can analyze crashes that occurred on end-devices by combining core dumps and Intel PT traces. However, these approaches generally do not allow to automatically identify the root cause unless there is a direct data dependency connecting root cause and crashing instruction. Furthermore, REPT and RETRACER focus on providing an interactive debugging session for an analyst to inspect manually what happened before the crash.

In cases such as the type confusion above, or when debugging JIT-based software such as JavaScript engines, a single crashing input may not allow identifying the root cause without extensive manual reasoning. Therefore, one can use a fuzzer to perform *crash exploration*. In this mode, the fuzzer is seeded with crashing inputs which it mutates as long as they keep crashing the target application. This process generates new inputs that are related to the original crashing input, yet slightly different (e. g., they could trigger the crash via a different path). A diverse set of crashing inputs that mostly trigger the same bug can aid analysis. Observing multiple ranges of values and different control-flow edges taken can help narrow down potential root causes. However, none of the aforementioned methods takes advantage of this information. Consequently, identifying the root cause remains a challenging task, especially if there is no direct data dependency between root cause and crashing instruction. Although techniques such as ASAN, POMP, REPT and RETRACER provide more context, they often fail to identify the root cause and provide no explanation of the fault.

In this chapter, we address this problem by developing an automated approach capable of finding the root cause given a crashing input. This significantly reduces human effort: unlike the approaches discussed previously, we do not only identify a code location, but also an explanation of the problem. This also reduces the number of locations an analyst has to inspect, as AURORA only considers instructions with a plausible explanation.

To enable precise identification of the root cause, we first pick one crashing input and produce a diverse set of similar inputs, some of which cause a crash while others do not. We then execute these newly-generated inputs while tracking the binary program's internal state. This includes control-flow information and relevant register values for each instruction. Given such detailed traces for many different inputs, we create a set of simple Boolean expressions (around 1,000 per instruction) to predict whether the input causes a crash. Intuitively, these predicates capture interesting runtime behavior such as whether a specific branch is taken or whether a register contains a suspiciously small value.

Consider our previous type confusion example and assume that a pointer to the constructor is called at some location in the program. Using the tracked information obtained from the diversified set of inputs, we can observe that (nearly) all calls in crashing inputs invoke the constructor of type `A`, while calls to the constructor of `B` imply that the input is not going to cause a crash. Thus, we can pinpoint the problem at an earlier point of the execution, even when no data taint connection exists between crashing location and root cause. This example also demonstrates that our approach needs to evaluate a large set of predicates, since many factors have to be captured, including different program contexts and vulnerability types. Using the predicates as a metric for each instruction, we can automatically pinpoint the possible root cause of crashes. Additionally, the predicates provide a concrete explanation of *why* the software fault occurs.

We built a prototype implementation of our approach in a tool called Aurora. To evaluate Aurora, we analyze 25 targets that cover a diverse set of vulnerability classes, including five use-after-free vulnerabilities, ten heap buffer overflows and two type confusion vulnerabilities that previous work fails to account for. We show that Aurora reliably allows identifying the root cause even for complex binaries. For example, we analyzed a type confusion bug in `mruby` where an exception handler fails to raise a proper exception type. It took an expert multiple days to identify the actual fault. Using our technique, the root cause was pinpointed automatically.

In summary, our key contributions are threefold:

- We present the design of Aurora, a generic approach to automatically pinpoint the location of the root cause and provide a semantic explanation of the crash.

- We propose a method to synthesize domain-specific predicates for binary programs, tailored to the observed behavior of the program. These predicates allow accurate predictions on whether a given input will crash or not.

- We implement a prototype of Aurora and demonstrate that it can automatically and precisely identify the root cause for a diverse set of 25 software faults.

To foster research on this topic, we release the implementation of Aurora at `https://github.com/RUB-SysSec/aurora`.

## 5.2 Challenges in Root Cause Analysis

Despite various proposed techniques, root cause identification and explanation are still complex problems. Thus, we now explore different techniques and discuss their limitations.

### 5.2.1 Running Example

The following code snippet shows a minimized example of Ruby code that leads to a type confusion bug in the `mruby` interpreter [8] found by a fuzzer:

```
1 NotImplementedError = String
2 Module.constants
```

In the first line, the exception type `NotImplementedError` is modified to be an alias of type `String`. As a consequence, each instance of `NotImplementedError` created in the future will be a `String` rather than the expected exception. In the second line, we call the `constants` function of `Module`. This function does not exist, provoking `mruby` to raise a `NotImplementedError`. Raising the exception causes a crash in the `mruby` interpreter.

To understand why the crash occurs, we need to dive into the C code base of the `mruby` interpreter. Note that `mruby` types are implemented as structs on the interpreter level. When we re-assign the exception type `NotImplementedError` to `String`, this is realized on C level by modifying the pointer such that it points to a struct representing the `mruby` `String` type. The method `Module.constants` is only a stub that creates and raises an exception. When the exception is raised in the second line, a new instance of `NotImplementedError` is constructed (which now actually results in a `String` object) and passed to `mruby`'s custom exception handling function. This function assumes that the passed object has an exception type without checking this further. It proceeds to successfully attach some error message—here „Module.constants not implemented“ (length $0x20$)—to the presumed exception object. Then, the function continues to fill the presumable exception with debug information available. During this process, it attempts to dereference a pointer to a table that is contained within all exception objects. However, as we have replaced the exception type by the string type, the layout of the underlying struct is different: At the accessed offset, the `String` struct stores the length of the contained string instead of a pointer as it would be the case for the exception struct. As a result, we do not dereference the pointer but interpret the length field as an address, resulting in an attempt to dereference $0x20$. Since this leads to an illegal memory access, the program crashes.

To sum up, redefining an exception type with a string leads to a type confusion vulnerability, resulting in a crash when this exception is raised. The developer fix introduces a type check, thus preventing this bug from provoking a crash.

### 5.2.2 Crash Triaging

Assume our goal is to triage the previously explained bug, given only the crashing input (obtained from a fuzzing run) as a starting point. In the following, we discuss different approaches to solve this task and explain their challenges.

**Debugger.** Starting at the crashing location, we can manually inspect the last few instructions executed, the registers at crashing point and the call stack leading to this situation. Therefore, we can see that $0x20$ is first loaded to some register and then dereferenced, resulting in the crash. Our goal then is to identify *why* the code attempts to dereference this value and *how* this value ended up there. We might turn towards the call stack, which indicates that the problem arises during some helper function that is called while raising an exception. From this point on, we can start investigating by manually following the flow of execution backward from the crashing cause up to the root cause. Given that the code of the `mruby` interpreter is non-trivial and the bug is somewhat complex, this takes a lot of time. Thus, we may take another angle and use some tool dedicated to detecting memory errors, for example, sanitizers.

**Sanitizer.** Sanitizers are a class of tools that often use compile-time instrumentation to detect a wide range of software faults. There are various kinds of sanitizers, such as MSAN [204] to detect usage of uninitialized memory or ASAN [197] to detect heap- and stack-based buffer overflows, use-after-free (UAF) errors and other faults. Sanitizers usually rely on the usage of shadow memory to track whether specific memory can be accessed or not. ASAN guards allocated memory (e. g., stack and heap) by marking neighboring memory as non-accessible. As a consequence, it detects out-of-bounds accesses. By further marking freed memory as non-accessible (as long as other free memory is available for allocation), temporal bugs can be detected. MSAN uses shadow memory to track for each bit, whether it is initialized or not, thereby preventing unintended use of uninitialized memory.

Using such tools, we can identify invalid memory accesses even if they are not causing the program to crash immediately. This situation may occur when other operations do not access the overwritten memory. Additionally, sanitizers provide more detailed information on crashing cause and location. As a consequence, sanitizers are more precise and pinpoint issues closer to the root cause of a bug.

Unfortunately, this is not the case for our example: re-compiling the binary with ASAN provides no new insights because the type confusion does not provoke any memory errors that can be detected by sanitizers. Consequently, we are stuck at the same crashing location as before.

**Backward Taint Analysis.** To deepen our understanding of the bug, we could use automated root cause analysis tools [68, 69, 221] that are based on reverse execution and backward taint tracking to increase the precision further. However, in our example, there is no direct data flow between the crash site and the actual root cause. The data flow ends in the constructor of a new `String` that is unrelated to the actual root cause. As taint tracking does not provide interesting information, we try to obtain related inputs that trigger the same bug in different crashing locations. Finding such inputs would give us a different perspective on the bug's behavior.

**Crash Exploration.** To achieve this goal, we can use the so-called *crash exploration* mode [232] that fuzzers such as AFL [231] provide. This mode takes a crashing input as a seed and mutates it to generate new inputs. From the newly generated inputs, the fuzzer only keeps those in the fuzzing queue that still result in a crash. Consequently, the fuzzer creates a diverse set of inputs that mostly lead to the same crash but exhibited new code coverage by exercising new paths. These inputs are likely to trigger the same bug via different code paths.

To gain new insights into the root cause of our bug, we need the crash exploration mode to trigger new behavior related to the type confusion. In theory, to achieve this, the fuzzer could assign another type than `String` to `NotImplementedError`. However, fuzzers such as AFL are more likely to modify the input to something like „`Stringgg`" or „`Strr`" than assigning different, valid types. This is due to the way its mutations work [44]. Still, AFL manages to find various crashing inputs by adding new `mruby` code unrelated to the bug.

To further strengthen the analysis, a fuzzer with access to domain knowledge, such as *grammar-based fuzzers* [33, 78, 173], can be used. Such a fuzzer recognizes that `String` is a grammatically valid element for Ruby which can be replaced by other grammar elements. For example, `String` can be replaced by `Hash`, `Array` or `Float`. Assume that the fuzzer chooses `Hash`; the newly derived input crashes the binary at a later point of execution than our original input. This result benefits the analyst as comparing the two inputs indicates that the crash could be related to `NotImplementedError`'s type. As a consequence, the analyst might start focusing on code parts related to the object type, reducing the scope of analysis. Still, this leaves the human analyst with an additional input to analyze, which means more time spent on debugging.

Overall, this process of investigating the root cause of a given bug is not easy and—depending on the bug type and its complexity—may take a significant amount of time and domain knowledge. Even though various methods and tools exist, the demanding tasks still have to be accomplished by a human. In the following, we present our approach to automate the process of identifying and explaining the root cause for a given crashing input.

## 5.3 Design

Given a crashing input and a binary program, our goal is to find an explanation of the underlying software fault's root cause. We do so by locating behavioral differences between crashing and non-crashing inputs. In its core, our method conducts a statistical analysis of differences between a set of crashing and non-crashing inputs. Thus, we first create a dataset of diverse program behaviors related to the crash, then monitor relevant input behavior and, finally, comparatively analyze them. This is motivated by the insight that crashing inputs must—at some point—semantically deviate from non-crashing inputs. Intuitively, the first relevant behavior during program execution that causes the deviation is the root cause.

In a first step, we create two sets of related but diverse inputs, one with crashing and one with non-crashing inputs. Ideally, we only include crashing inputs caused by the same root cause. The set of non-crashing inputs has no such restrictions, as they are effectively used as counterexamples in our method. To obtain these sets, we perform crash exploration fuzzing on one initial crashing input (a so-called *seed*).

Given the two sets of inputs, we observe and monitor (i. e., trace) the program behavior for each input. These traces allow us to correlate differences in the observations with the outcome of the execution. Using this statistical reasoning, we can identify differences that predict whether a program execution will crash or not. To formalize these differences, we synthesize predicates that state whether a bug was triggered. Intuitively, the first predicate that can successfully predict the outcome of all (or most) executions also explains the root cause. As the final result, we provide the analyst with a list of relevant explanations and addresses, ordered by the quality of their prediction and time of execution. That is, we prefer explanations that predict the outcome well. Amongst good explanations, we prefer these that are able to predict the crash as early as possible.

On a high-level view, our design consist of three individual components: (1) *input diversification* to derive two diverse sets of inputs (crashing and non-crashing), (2) *monitoring input behavior* to track how inputs behave and (3) *explanation synthesis* to synthesize descriptive predicates that distinguish crashing from non-crashing inputs. In the following, we present each of these components.

### 5.3.1 Input Diversification

As stated before, we need to create a diverse but similar set of inputs for the single crashing seed given as input to our approach. On the one hand, the inputs should be diverse such that statistical analysis reveals measurable differences. On the other hand, the inputs should share a similar basic structure such that they explore states similar to the root cause. This allows for a comparative analysis of how crashes and non-crashes behave on the buggy path.

To efficiently generate such inputs, we can use the *crash exploration mode* bundled with fuzzers such as AFL. As described previously, this mode applies mutations to inputs as long as they keep crashing. Inputs not crashing the binary are discarded from the queue and saved to the non-crashing set; all inputs remaining within the fuzzing queue constitute the crashing set. In general, the more diversified inputs crash exploration produces, the more precise the statistical analysis becomes. Fewer inputs are produced in less time but cause more false positives within the subsequent analysis. Once the input sets have been created, they are passed to the analysis component.

### 5.3.2 Monitoring Input Behavior

Given the two sets of inputs—crashing and non-crashing—we are interested in collecting data allowing semantic insights into an input's behavior. To accommodate our binary-only approach, we monitor the runtime execution of each input, collecting the values

of various expressions. For each instruction executed, we record the minimum and maximum value of all modified registers (this includes general-purpose registers and the flag register). Similarly, we record the maximum and minimum value stored for each memory write access. Notably and perhaps surprisingly, we did not observe any benefit in tracing the memory addresses used; therefore, we do not aggregate information on the target addresses. It seems that the resulting information is too noisy and all relevant information is already found in observed registers. We only trace the minimum and maximum of each value to limit the amount of data produced by loops. This loss of information is justified by the insight that values causing a crash usually surface as either a minimum or maximum value. Our evaluation empirically supports this thesis. This optimization greatly increases the performance, as the amount of information stored per instruction is constant. At the same time, it is precise enough to allow statistical identification of differences. Besides register and memory values, we store information on control-flow edges. This allows us to reconstruct a coarse control-flow graph for a specific input's execution. Control flow is interesting behavior, as it may reveal code that is only executed for crashing inputs. Furthermore, we collect the address ranges of stack and heap to test whether certain pointers are valid heap or stack pointers.

We do not trace any code outside of the main executable, i. e., shared libraries. This decreases overhead significantly while removing tracing of code that—empirically—is not interesting for finding bugs within a given binary program. For each input, we store this information within a trace file that is passed on to the statistical analysis.

### 5.3.3 Explanation Synthesis

Based on the monitoring, explanation synthesis is provided with two sets of trace files that describe intrinsic behaviors of crashing and non-crashing inputs. Our goal is to isolate behavior in the form of predicates that correlate to differences between crashing and non-crashing runs. Any such predicate pointing to an instruction indicates that this particular instruction is related to a bug. Our predicates are Boolean expressions describing concrete program behavior, e. g., „the maximum value of `rax` at this position is less than 2“. A predicate is a triple consisting of a semantic description (i. e., the Boolean expression), the instruction's address at which it is evaluated and a score indicating the ability to differentiate crashes from non-crashes. In other words, the score expresses the probability that an input crashes for which the predicate evaluates to true. Consequently, predicates with high scores identify code locations somewhere on the path between root cause and crashing location. In the last step, we sort these predicates first by score, then by the order in which they were executed. Given this sorted list of predicates, a human analyst can then manually analyze the bug. Since these predicates and the calculation of the score are the core of our approach, we present more details in the following section.

## 5.4 Predicate-based Root Cause Analysis

Given the trace information for all inputs in both sets, we can reason about potential root cause locations and determine predicates that explain the root cause. To this end, we construct predicates capable of discriminating crashing and non-crashing runs, effectively pinpointing conditions within the program that are met only when encountering the crash. Through the means of various heuristics described in Section 5.4.4, we filter the conditions and deduce a set of locations close to the root cause of a bug, aiding a developer in the tedious task of finding and fixing the root cause. This step potentially outputs a large number of predicates, each of which partitions the two sets. In order to determine the predicate explaining the root cause, we set conditional breakpoints that represent the predicate semantics. We then proceed to execute the binary for each input in the crashing set, recording the order in which predicates are triggered. As a result, we obtain for each input the order in which the predicates were encountered during execution. Given this information and the predicates' scores, we can define a ranking over all predicates. In the following, we present this approach in detail.

The first step is to read the results obtained by tracing the inputs' behavior. Given these traces, we collect all control-flow transitions observed in crashing and non-crashing inputs and construct a joined control-flow graph that is later used to synthesize control-flow predicates. Afterward, we compute the set of instructions identified by their addresses that are relevant for our predicate-based analysis. Since we are interested in behavioral differences between crashes and non-crashes, we only consider addresses that have been visited by at least one crashing and one non-crashing input. Note that—as a consequence—some addresses are discarded if they are visited in crashes but not in non-crashes. However, in such a situation, we would observe control-flow transitions to these discarded addresses from addresses that are visited by inputs from both sets. Consequently, we do not lose any precision by removing these addresses.

Based on the trace information, we generate many predicates for each address (i. e., each instruction). Then, we test all generated predicates and store only the predicate with the highest score. In the following, we describe the types of predicates we use, how these predicates can be evaluated and present our ranking algorithm. Note that by assumption a predicate forecasts a non-crash, if it is based on an instruction that was never executed. This is motivated by the fact that not-executed code cannot be the cause of a crash.

### 5.4.1 Predicate Types

To capture a wide array of possible explanations of a software fault's root cause, we generate three different categories of predicates, namely (1) control-flow predicates, (2) register and memory predicates, as well as (3) flag predicates. In detail, we use the following types of predicates:

**Control-flow Predicates.** Based on the reconstructed control-flow graph, we synthesize edge predicates that evaluate whether crashes and non-crashes differ in execution flow. Given a control-flow edge from `x` to `y`, the predicate `has_edge_to` indicates that

we observed at least one transition from `x` to `y`. Contrary, `always_taken_to` expresses that *every* outgoing edge from `x` has been taken to `y`. Finally, we evaluate predicates that check if the number of successors is greater than or equal to $n \in \{0, 1, 2\}$.

**Register and Memory Predicates.** For each instruction, we generate predicates based on various expressions: the minimum and the maximum of all values written to a register or memory, respectively. For each such expression (e. g., $r = \texttt{max}(\texttt{rax})$) we introduce a predicate $r < c$. We synthesize constants for $c$ such that the predicate is a good predictor for crashing and non-crashing inputs. The synthesis is described in Section 5.4.3. Additionally, we have two fixed predicates testing whether expressions are valid heap or stack pointers, respectively: `is_heap_ptr`$(r)$ and `is_stack_ptr`$(r)$.

**Flag Predicates.** On the x86 and x86-64 architecture, the flag register tracks how binary comparisons are evaluated and whether an overflow occurred, making it an interesting target for analysis. We use flag predicates that each check one of the flag bits, including the carry, zero and overflow flag.

### 5.4.2 Predicate Evaluation

For each address, we generate and test predicates of all types and store the predicate with the highest score. In the following, we detail how to evaluate and score an individual predicate. Generally speaking, we are interested in measuring the quality of a predicate, i. e., how well it *predicts* the actual behavior of the target application. Thus, it is a simple binary classification. If the target application crashes on a given input—also referred to as *test case* in the following—the predicate should evaluate to true. Otherwise, it should evaluate to false. We call a predicate *perfect* if it correctly predicts the outcome of all test cases. In other words, such a predicate perfectly separates crashing and non-crashing inputs.

Unfortunately, there are many situations in which we cannot find a perfect predicate; consequently, we assign each predicate a probability on how well it predicts the program's behavior given the test cases. For example, if there are multiple distinct bugs within the input set, no predicate will explain all crashes. This can occur if the crash exploration happens to modify a crashing input in such a way that it triggers one or multiple other bug(s). Alternatively, the actual best predicate might be more complex than predicates that could be synthesized automatically; consequently, it cannot predict all cases perfectly.

To handle such instances, we model the program behavior as a noisy evaluation of the given predicate. In this model, the final outcome of the test case is the result of the predicate XORed with some random variable. More precisely, we define a predicate $p$ as a mapping from an input trace to a Boolean variable ($p : \textit{trace} \mapsto \{0, 1\}$) that predicts whether the execution crashes. Using this predicate, we build a statistical model $O(\text{input}) = p(\text{input}) \oplus R$ to approximate the observed behavior. The random variable $R$ is drawn from a Bernoulli distribution ($R \sim \text{Bernoulli}(\theta)$) and denotes the noise introduced by insufficiently precise predicates. Whenever $R = 0$, the predicate $p(\text{input})$ correctly predicts the outcome. When $R = 1$, the predicate mispredicts the outcome. Our stochastic model has a single parameter $\theta$ that represents the probability

that the predicate mispredicts the actual outcome of the test case. We cannot know the real value of $\theta$ without simulating every possible behavior of a program. Instead, we perform maximum likelihood estimation using the sample of actual test inputs to approximate a $\hat{\theta}$. This value encodes the uncertainty of the predictions made by the predicate. We later employ this uncertainty to rank the different predicates:

$$\hat{\theta} = \frac{C_f + N_f}{C_f + C_t + N_f + N_t}$$

We count the number of both mispredicted crashes ($C_f$) and mispredicted non-crashes ($N_f$) divided by the number of all predictions, i. e., the number of all mispredicted inputs as well as the number of all correctly predicted crashed ($C_t$) and non-crashes ($N_t$).

As we demonstrate in Section 5.6.3, using crash exploration to obtain samples can cause a significant class imbalance, i. e., we may find considerably more non-crashing than crashing inputs. To avoid biasing our scoring scheme towards the bigger class, we normalize each class by its size:

$$\hat{\theta} = \frac{1}{2} * \left(\frac{C_f}{C_f + C_t} + \frac{N_f}{N_f + N_t}\right)$$

If $\hat{\theta} = 0$, the predicate is perfect. If $\hat{\theta} = 1$, the negation of the predicate is perfect. The closer $\hat{\theta}$ is to 0.5, the worse our predicate performs in predicting the actual outcome.

Finally, we calculate a score using $\hat{\theta}$. To obtain a score in the range of [0, 1], where 0 is the worst and 1 the best possible score, we calculate $2 * \texttt{abs}(\hat{\theta} - 0.5)$. We use this score to pick the best predicate for each instruction that has been visited by at least one crashing and one non-crashing input. While the score is used to rank predicates, $\hat{\theta}$ indicates whether $p$ or its negation $\neg p$ is the better predictor. Intuitively, if $\hat{\theta} > 0.5$, $p$ is a good predictor for non-crashing inputs. As our goal is to predict crashes, we use the negated predicate in these cases.

**Example 5.1.** *Assume that we have* $1,013$ *crashing and* $2,412$ *non-crashing inputs. Furthermore, consider a predicate* $p_1$*, with* $p_1 := \texttt{min}(\texttt{rax}) < \mathit{0xff}$*. Then, we count* $C_f := 1013$*,* $C_t = 0$*,* $N_f = 2000$ *and* $N_t = 412$*. Therefore, we estimate* $\hat{\theta}_1 = \frac{1}{2} \cdot \left(\frac{1013}{1013} + \frac{2000}{2000+412}\right) \approx 0.9146)$*. The predicate score is* $s_1 = 2 \cdot \text{abs}(0.9146 - 0.5) \approx 0.8292$*, indicating that the input is quite likely to crash the program. Even though* $\hat{\theta}$ *is large and the majority of the outcomes is mispredicted, this high score is explained by the fact that—as* $\hat{\theta}_1 > 0.5$*—we invert the predicate* $p_1$*. Thus, true and false positives/negatives are switched, resulting in a large amount of true positives (*$C_t = 1013$*) and true negatives (*$N_t = 2000$*) for the inverted predicate:* $\neg p_1 := \texttt{min}(\texttt{rax}) \geq \mathit{0xff}$

*Testing another predicate* $p_2$ *for the same instruction with* $\hat{\theta}_2 = 0.01$*, we calculate the score* $s_2 = 2 \cdot \text{abs}(0.01 - 0.5) = 0.98$*. Since* $s_2 > s_1$*, consequently we only store* $p_2$ *as best predicate for this instruction.*

### 5.4.3 Synthesis of Constant Values

When computing our register and memory predicates of type $r < c$, we want to derive a constant $c$ that splits the test inputs into crashing and non-crashing inputs based on all values observed for $r$ during testing. These predicates can only be evaluated once a value for $c$ is fixed. Since $c$ can be any 64-bit value, it is prohibitively expensive to try all possible values. However, $c$ splits the inputs into exactly two sets: Those where $r$ is observed to be smaller than $c$ and the rest. The only way to change the quality of the predicate is to choose a value of $c$ that flips the prediction of at least one value of $r$. All constants $c$ between two different observations of $r$ perform the exact same split of the test inputs. Consequently, the only values that change the behavior of the predicate are exactly the observed values of $r$. We exploit this fact to find the best value(s) for $c$ using only $\mathcal{O}(n * \log(n))$ steps where $n$ is the number of test cases.

To implement this, we proceed as follows: In a preprocessing step, we collect all values for an expression $r$ at the given instruction and sort them. Then, we test each value observed for $r$ as a candidate for $c$. We then want to evaluate our candidate for $c$ on all inputs reaching the address. Naively, we would recompute the score for each value of $c$; however, this would yield a quadratic runtime. To increase the performance, we exploit the fact that we only need $C_t$, $C_f$, $N_t$, $N_f$ to calculate the score. This property of our scoring scheme allows us to update the score in constant time when checking the next candidate value of $c$.

To calculate the score for any candidate value $c_i$, we start at the smallest candidate $c_0$ and calculate the predicate's score by evaluating the predicate on all inputs and counting the number of correctly predicted outcomes. After calculating the score of the $i^{\text{th}}$ possible candidate $c_i$, we can update the score for the candidate $c_{i+1}$ by tracking the number of correctly predicted crashes and non-crashes. Since using $c_{i+1}$ instead of $c_i$ only flips a single prediction, we can efficiently update $C_t$, $C_f$, $N_t$, $N_f$ in constant time. When using $c_i$ resulted in a correctly predicted crash for the $i^{\text{th}}$ observation, we decrement $C_t$. Likewise, if the old prediction was an incorrectly predicted non-crash, we decrement $N_f$. The other cases are handled accordingly. Afterward, we increment the number of observed outcomes based on the results of the new predicate in the same fashion. This allows us to track $C_t$, $C_f$, $N_t$, $N_f$ while trying all values of $c$ to determine the value which maximizes the score. Finally, we might have dropped some inputs that did not reach the given instruction; thus, we then perform one re-evaluation of the score on the whole dataset to determine the final score for this predicate.

Note that the predicate is constructed involving all addresses reaching that instruction. Consequently, it is perfect with respect to the whole dataset: all data not yet evaluated does not reach this address and thus cannot affect the synthesized value. Another consequence of this fact is that our synthesis works both for ranges and single values.

**Example 5.2.** *Consider that we want to synthesize a value $c$ that maximizes the score of the predicate $p(r) = r < c$. Assume that we have four inputs reaching the address where the predicate is evaluated and we observed the following data:*

| ***outcome*** | *crash* | *crash* | *non-crash* | *non-crash* |
|---|---|---|---|---|
| ***values of*** $r$ | *0x08* | *0x0f* | *0x400254* | *0x400274* |

*In this example, the values are already sorted. Remember that we are interested in locating the cutoff value, i. e., the value of c that separates crashing and non-crashing inputs best. Hence, we proceed to calculate the score for each candidate, starting with the smallest* $c = 0x8$*. Since* $r < 0x8$ *is never true for our four inputs, they are all predicted to be non-crashing. Therefore, we obtain* $C_f = 2$*,* $C_t = 0$*,* $N_f = 0$*,*$N_t = 2$*. This results in* $\hat{\theta} = \frac{1}{2}\left(\frac{2}{2+0} + \frac{0}{0+2}\right) = 0.5$ *and, consequently, in a score* $= 2 * abs(\hat{\theta} - 0.5) = 0$*, indicating that this is not a good candidate for c. Using the next candidate* $c = 0x0\text{f}$*, we now predict that the first input is crashing. Since the first input triggered a crash, we update* $C_f$ *and* $C_t$ *by incrementing* $C_t$ *and decrementing* $C_f$*. Consequently, we obtain* $C_f = 1$*,* $C_t = 1$*,* $N_f = 0$ *and* $N_t = 2$*, resulting in* $\hat{\theta} = 0.75$ *and a final score of* $0.5$*. Repeating this for the next step, we obtain a perfect score for the next value* $0x400254$ *as both crashing values are smaller. This yields the final predicate* $p(r) = x < 0x400254$ *that will be re-evaluated on the whole dataset.*

We observed that if *all* recorded constants are either valid stack or heap addresses (i. e., pointers), we receive a high number of false positives since these addresses are too noisy for statistical analysis. Accordingly, we do not synthesize predicates other than `is_heap_ptr` and `is_stack_ptr` for these cases.

### 5.4.4 Ranking

Once all steps of our statistical analysis are completed, we obtain the best predicate for each instruction. A predicate's score indicates how well a predicate separates crashing and non-crashing inputs. Since we synthesize one predicate for each instruction, we obtain a large number of predicates. Note that most of them are independent of the bug; thus, we discard predicates with a score lower than the empirically determined threshold of 0.9. Consequently, the remaining predicates identify locations that are related to the bug.

Still, we do not know in which order relevant predicates are executed; therefore, we cannot distinguish whether a predicate is related to the root cause or occurs later on the path to the crash site. As predicates early in the program trace are more likely to correspond to the root cause, we introduce a new metric called the *execution rank*. To calculate the execution rank, we determine the temporal order in which predicates are executed. To do so, we add a conditional breakpoint for each relevant predicate $p$. This breakpoint triggers if the predicate evaluates to true. For each crashing input, we can execute the program, recording the order in which breakpoints are triggered. If some predicate $p$ is at program position $i$ and we observed $n$ predicates in total, $p$'s execution rank is $\frac{i}{n}$. If some predicate is not observed for a specific run, we set its execution rank to 2 as a penalty. Since a predicate's execution rank may differ for each crashing input due to different program paths taken, we average over all executions.

However, the primary metric is still its prediction score. Thus, we sort predicates by their prediction score and resolve ties by sorting according to the execution rank.

**Example 5.3.** *Consider three predicates $p_1$, $p_2$ and $p_3$ with their respective scores* 1*,* 0.99 *and* 0.99*. Furthermore, assume that we have the crashing inputs $i_1$ and $i_2$. Let the observed predicate order be $(p_1, p_3)$ for $i_1$ and $(p_1, p_3, p_2)$ for $i_2$. Then, we obtain the execution ranks:*

$p_1$: $\frac{1}{2} \cdot \left(\frac{1}{2} + \frac{1}{3}\right) \approx 0.41$
$p_2$: $\frac{1}{2} \cdot \left(2 + \frac{3}{3}\right) = 1.5$
$p_3$: $\frac{1}{2} \cdot \left(\frac{2}{2} + \frac{2}{3}\right) \approx 0.83$

*Since we sort first by score and then by execution rank, we obtain the final predicate order $(p_1, p_3, p_2)$.*

## 5.5 Implementation

To demonstrate the practical feasibility of the proposed approach, we implemented a prototype of Aurora. We briefly explain important implementation aspects in the following, the full source code is available at `https://github.com/RUB-SysSec/aurora`.

**Input Diversification.** For the purpose of exploring inputs close to the original crash, we use AFL's *crash exploration mode* [232]. Given a crashing input, it finds similar inputs that still crash the binary. Inputs not crashing the program are not fuzzed any further. We modified AFL (version 2.52b) to save these inputs to the *non-crashing set* before discarding them from the queue.

**Monitoring Input Behavior.** To monitor the input behavior, we implemented a pintool for Intel PIN [122] (version 3.7). Relying on Intel's generic and architecture-specific inspection APIs, we can reliably extract relevant information.

**Explanation Synthesis.** The explanation synthesis is written in Rust. It takes two folders containing traces of crashes and non-crashes as input. Then, it reconstructs the joined control-flow graph and then synthesizes and evaluates all predicates. Finally, it monitors and ranks the predicates as described before. To monitor the execution of the predicates, we set conditional breakpoints using the `ptrace` syscall. In a final step, we use `binutils`' `addr2line` [83] to infer the source file, function name and line for each predicate. If possible, all subsequent analysis parts are performed in parallel. Overall, about $5,000$ lines of code were written for this component.

## 5.6 Experimental Evaluation

Based on the prototype implementation of Aurora, we now answer the following research questions:

**RQ 1:** Is Aurora able to identify and explain the root cause of complex and highly exploitable bug classes such as type confusions, use-after-free vulnerabilities and heap buffer overflows?

**RQ 2:** How close is the automatically identified explanation to the patch implemented by the developers?

**RQ 3:** How many predicates are related to the fault?

To answer these research questions, we devise a set of experiments where we analyze various types of software faults. For each fault, we have manually analyzed and identified the root cause; furthermore, we considered the patches provided by the developers.

### 5.6.1 Setup

All of our experiments are conducted within a cloud VM with 32 cores (based on Intel Xeon Silver 4114, 2.20 GHz) and 224 GiB RAM. We use the Ubuntu 18.04 operating system. To facilitate deterministic analysis, we disable address space layout randomization (ASLR).

We selected 25 software faults in different well-known applications, covering a wide range of fault types. In particular, we picked the following bugs:

- ten heap buffer overflows, caused by an integer overflow (#1 `mruby` [7]), a logic flaw (#2 `Lua` [4], #3 `Perl` [16] and #4 `screen` [22]) or a missing check (#5 `readelf` [21], #6 `mruby` [6], #7 `objdump` [14], #8 `patch` [15]), #9 `Python` 2.7/3.6 [18] and #10 `tcpdump` [24])
- one null pointer dereference caused by a logic flaw (#11 `NASM` [11])
- three segmentation faults due to integer overflows (#12 `Bash` [1] and #13 `Bash` [2]) or a race condition (#14 `Python` 2.7 [19])
- one stack-based buffer overflow (#15 `nm` [13])
- two type confusions caused by missing checks (#16 `mruby` [8] and #17 `Python` 3.6 [20])
- three uninitialized variables caused by a logic flaw (#18 `Xpdf` [25]) or missing checks (#19 `mruby` [10] and #20 `PHP` [17])
- five use-after-frees, caused by a double free (#21 `libzip` [3]), logic flaws (#22 `mruby` [9], #23 `NASM` [12] and #24 `Sleuthkit` [23]) or a missing check (#25 `Lua` [5])

These bugs have been uncovered during recent fuzzing runs or found in the bug tracking systems of well-known applications. Our general selection criteria are (i) the presence of a proof-of-concept file crashing the application and (ii) a developer-provided fix. The former is required as a starting point for our analysis, while the latter serves as ground truth for the evaluation.

For each target, we compile two binaries: One instrumented with AFL that is used for crash exploration and one non-instrumented binary for tracing purposes. Note that some of the selected targets (e. g., #1, #5 or #19) are compiled with sanitizers, ASAN or MSAN, because the bug only manifests when using a sanitizer. The targets compiled without any sanitizer are used to demonstrate that we are not relying on any sanitizers or requiring source code access. The binary used for tracing is always built with debug symbols and without sanitizers. For the sake of the evaluation, we need to measure the quality of our explanations, as stated in the **RQ 1** and **RQ 2**. Therefore, we use debug symbols and the application's source code to compare the identified root cause with the developer fix. To further simplify this process, we derive source line, function name and source file for each predicate via `addr2line`. This does not imply that our approach by

any means requires source code: all our analysis steps run on the binary level regardless of available source code. Experiments using a binary-only fuzzer would work the exact same way. However, establishing the ground truth would be more complex and hence we use source code strictly for evaluation purposes.

For our evaluation, we resort to the well-known AFL fuzzer and run its crash exploration mode for two hours with the proof-of-concept file as seed input. We found that this is enough time to produce a sufficiently large set of diverse inputs for most targets. However, due to the highly structured nature of the input languages for `mruby`, `Lua`, `nm`, `libzip`, `Python` (only #17) and `PHP`, AFL found less than 100 inputs within two hours. Thus, we repeat the crash exploration with 12 hours instead of 2 hours. Each input found during exploration is subsequently traced. Since some inputs do not properly terminate, we set a timeout of five minutes after which tracing is aborted. Consequently, we do lose a few inputs, see Table 5.4 for details. Similarly, our predicate ranking component may encounter timeouts. As monitoring inputs with conditional breakpoints is faster than tracing an input, we empirically set the default timeout to 60 seconds.

**Table 5.1:** Results of our root cause explanations. For 25 different bugs, we note the target, root and crashing cause as well as whether the target has been compiled using a sanitizer. Furthermore, we provide the number of predicates and source lines (SLOC) a human analyst has to examine until the location is reached where the developers applied the bug fix (denoted as *Steps to Dev. Fix*). Finally, the number of true and false positives (denoted as TP and FP) of the top 50 predicates are shown. * describes targets where no top 50 predicates with a score above or equal to 0.9 exist.

| | Target | Root Cause | Crash Cause | Sanitizer | Best Score | Steps to Dev. Fix | | Top 50 | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | #Predicates | #SLOC | TP | FP |
| #1 | `mruby` | int overflow | heap buffer overflow | ASAN | 0.998 | 1 | 1 | 50 | 0 |
| #2 | `Lua` | logic flaw | heap buffer overflow | ASAN | 1.000 | 1 | 1 | 50 | 0 |
| #3 | `Perl` | logic flaw | heap buffer overflow | - | 1.000 | 13 | 10 | 43 | 7 |
| #4 | `screen` * | logic flaw | heap buffer overflow | - | 0.999 | 26 | 16 | 30 | 0 |
| #5 | `readelf` | missing check | heap buffer overflow | ASAN | 1.000 | 7 | 5 | 50 | 0 |
| #6 | `mruby` | missing check | heap buffer overflow | ASAN | 1.000 | 1 | 1 | 12 | 38 |
| #7 | `objdump` | missing check | heap buffer overflow | ASAN | 0.981 | 3 | 3 | 48 | 2 |
| #8 | `patch` | missing check | heap buffer overflow | ASAN | 0.997 | 1 | 1 | 50 | 0 |
| #9 | `Python` | missing check | heap buffer overflow | - | 1.000 | 46 | 28 | 44 | 6 |
| #10 | `tcpdump` | missing check | heap buffer overflow | - | 0.994 | 1 | 1 | 50 | 0 |
| #11 | `NASM` | logic flaw | nullptr dereference | - | 1.000 | - | - | 50 | 0 |
| #12 | `Bash` | int overflow | segmentation fault | - | 0.992 | 10 | 6 | 28 | 22 |
| #13 | `Bash` | int overflow | segmentation fault | - | 0.999 | 9 | 6 | 35 | 15 |
| #14 | `Python` | race condition | segmentation fault | - | 1.000 | 13 | 13 | 27 | 23 |
| #15 | `nm` * | missing check | stack buffer overflow | ASAN | 0.980 | 1 | 1 | 35 | 0 |
| #16 | `mruby` | missing check | type confusion | - | 1.000 | 33 | 15 | 50 | 0 |
| #17 | `Python` | missing check | type confusion | - | 1.000 | 215 | 141 | 7 | 43 |
| #18 | `Xpdf` | logic flaw | uninitialized variable | ASAN | 0.997 | 16 | 11 | 50 | 0 |
| #19 | `mruby` | missing check | uninitialized variable | MSAN | 1.000 | 16 | 5 | 50 | 0 |
| #20 | `PHP` | missing check | uninitialized variable | MSAN | 1.000 | 42 | 19 | 29 | 21 |
| #21 | `libzip` * | double free | use-after-free | ASAN | 1.000 | 1 | 1 | 39 | 0 |
| #22 | `mruby` | logic flaw | use-after-free | ASAN | 1.000 | 9 | 6 | 42 | 8 |
| #23 | `NASM` * | logic flaw | use-after-free | - | 0.957 | 1 | 1 | 14 | 9 |
| #24 | `Sleuthkit` | logic flaw | use-after-free | - | 1.000 | 2 | 2 | 48 | 2 |
| #25 | `Lua` | missing check | use-after-free | ASAN | 1.000 | 3 | 3 | 50 | 0 |

**Table 5.2:** Maximum and average distance between developer fix and crashing location in both all and unique executed assembly instructions. For reference, the average amount of instructions executed between program start and crash is also provided.

| | Target | Maximum #Instructions all | unique | Average #Instructions all | unique | Average Total #Instructions all | unique |
|---|---|---|---|---|---|---|---|
| #3 | `Perl` | 845,689 | 7,321 | 435,873 | 5,697 | 1,355,013 | 32,259 |
| #4 | `screen` | 28,289,736 | 3,441 | 127,459 | 1,932 | 397,595 | 9,456 |
| #9 | `Python` | 3,759,699 | 9,330 | 743,216 | 5,445 | 34,914,300 | 60,508 |
| #10 | `tcpdump` | 6,727 | 1,567 | 2,263 | 546 | 103,655 | 3,622 |
| #11 | `NASM` | 22,678,105 | 8,256 | 1,940,592 | 4,383 | 2,546,740 | 9,729 |
| #12 | `Bash` | 450,428 | 3,549 | 11,965 | 116 | 1,053,498 | 19,221 |
| #13 | `Bash` | 2,584,606 | 1,094 | 178,873 | 612 | 1,100,495 | 16,817 |
| #14 | `Python` | 3,923,167 | 13,028 | 58,990 | 835 | 29,226,209 | 60,917 |
| #16 | `mruby` | 253,173 | 840 | 2,154 | 533 | 14,926,707 | 26,982 |
| #17 | `Python` | 800 | 428 | 498 | 407 | 46,112,224 | 74,590 |
| #23 | `NASM` | 7,401,732 | 4,842 | 184,036 | 2,919 | 2,885,104 | 8,244 |
| #24 | `Sleuthkit` | 199 | 156 | 197 | 155 | 25,780 | 5,960 |

### 5.6.2 Experiment Design

An overview of all analysis results can be found in Table 5.1. Recall that in practice the crashing cause and root cause of a bug differ. Thus, for each bug, we first denote its root cause as identified by Aurora and verified by the developers' patches. Subsequently, we present the crashing cause, i. e., the reason reported by ASAN or identified manually. For each target, we record the best predicate score observed. Furthermore, we investigate each developer fix, comparing it to the root cause identified by our automated analysis. We report the number of predicates an analyst has to investigate before finding the location of the developers' fix as *Steps to Dev. Fix.* We additionally provide the number of source code lines (column *SLOC*) a human analyst needs to inspect before arriving at the location of the developer fix since these fixes are applied on the source code level. Note that this number may be smaller than the number of predicates as one line of source code usually translates to multiple assembly instructions. Up to this day, no developer fix was provided for bug #23 (`NASM`). Hence, we manually inspected the root cause, identifying a reasonable location for a fix. Bug #11 has no unique root cause; the bug was fixed during a major rewrite of program logic (20 files and 500 lines changed). Thus, we excluded it from our analysis.

To obtain insights into whether our approach is actually capable of identifying the root cause even when it is separated from the crashing location by the order of thousands of instructions, we design an experiment to measure the distance between developer fix and crashing location in terms of executed assembly instructions. More specifically, for each target, we determine the maximum distance, the average distance over all crashing inputs and—to put this number in relation—the average of total instructions executed during a program run. Each metric is given in the number of assembly instructions executed and unique assembly instructions executed, where each instruction is counted at most once. Note that some bugs only crash in the presence of a sanitizer (as indicated by ASAN or MSAN in Table 5.1) and that our tracing binaries are never instrumented to avoid sanitizer artifacts disturbing our analysis. As a consequence, our distance measurement would run until normal program termination rather than program crash

for such targets. Since this would distort the experiment, we exclude such bugs from the comparison.

Finally, to provide an intuition of how well our approach performs, we analyze the top 50 predicates (if available) produced for each target, stating whether they are related to the bug or unrelated false positives. We consider predicates as related to the bug when they pinpoint a location on the path from root cause to crashing location and separate crashing and non-crashing inputs. For false positives, we evaluate various heuristics that allow to identify them and thereby reduce the amount of manual effort required.

### 5.6.3 Results

Following Aurora's results, the developer fix will be covered by the first few explanations. Typically, an analyst would have to inspect less than ten source code lines to identify the root cause. Exceptions are larger targets, such as `Python` (13 MiB) and `PHP` (31 MiB), or particularly challenging bugs such as type confusions (#16 and #17). Still, the number of source code lines to inspect is below 28 for all but the `Python` type confusion (#17), which contains a large amount of false positives. Despite the increased number of source code lines to investigate, the information provided by Aurora is still useful: for instance, for bug #16—where 15 lines are needed—most of the lines are within the same function and only six functions are identified as candidates for containing the root cause. We explain the increased number of false positives found for these targets at the end of this section in detail.

Another aspect of a bug's complexity is the distance between the root cause and crashing location. As Table 5.2 indicates, Aurora is capable of both identifying root causes when the distance is small (a few hundred instructions, e. g., 197 for `Sleuthkit`) and significant (millions of instructions, e. g., roughly 28 million for `screen`). Overall, we conclude **RQ 1** and **RQ 2** by finding that Aurora is generally able to provide automated root cause explanations close to the root cause—less than 30 source code lines and less than 50 predicates—for diverse bugs of varying complexity.

The high quality of the explanations generated by Aurora is also reflected by its high precision (i. e., the ratio of true positives to all positives). Among the top 50 predicates, there are significantly more true positives than negatives. More precisely, for 18 out of 25 bugs, we have a precision $\geq 0.84$, including 12 bugs with a precision of 1.0 (no false positives). Only for two bugs, the precision is less than 0.5—0.14 for #17 and 0.24 for #6. Note that for #6, the predicate pinpointing the developer fix is at the top of the list, rendering all these false positives irrelevant to triaging the root cause.

Despite the high precision, some false positives are generated. During our evaluation, we observed that they are mostly related to (1) (de-)allocation operations as well as garbage collectors, (2) control-flow, i. e., predicates which indicate that non-crashes executed the pinpointed code in diverse program contexts (e. g., different function calls or more loop iterations), (3) operations on (complex) data structures such as hash maps, arrays or locks, (4) environment, e. g., parsing command-line arguments or environment variables (5) error handling, e. g., signals or stack unwinding. Such superficial features may differentiate crashes and non-crashes but are generally not related to the actual

bug (excluding potential edge cases like bugs in the garbage collector). Many of these false positives occur due to insufficient code coverage achieved during crash exploration, causing the sets of crashing and non-crashing inputs to be not diverse enough.

**Table 5.3:** Analysis results of false positives within the top 50 predicates. For each target, we classify its false positives into the categories they are related to: allocation or garbage collector (Alloc), control flow (CF), data structure (DS), environment (Env) or error handling (Error). Additionally, we track the number of predicates an analyst has to inspect in more detail (In-depth Analysis) as well as propagations of false positives that can be discarded easily.

| | **Target** | **False Positive Categories** | | | | | **Propagations** | **In-depth Analysis** |
|---|---|---|---|---|---|---|---|---|
| | | **Alloc** | **CF** | **DS** | **Env** | **Error** | | |
| #3 | `Perl` | - | - | 7 | - | - | - | - |
| #6 | `mruby` | - | - | 38 | - | - | - | - |
| #7 | `objdump` | - | 2 | - | - | - | - | - |
| #9 | `Python` | - | 1 | - | 2 | 3 | - | - |
| #12 | `Bash` | 1 | 1 | - | 1 | 4 | 8 | 7 |
| #13 | `Bash` | 1 | 1 | - | - | 4 | 5 | 4 |
| #14 | `Python` | - | - | - | 3 | - | 15 | 5 |
| #17 | `Python` | 40 | - | 2 | - | - | - | 1 |
| #20 | `PHP` | - | - | - | 21 | - | - | - |
| #22 | `mruby` | - | 1 | - | - | - | 4 | 3 |
| #23 | `NASM` | 3 | - | - | - | 2 | 2 | 2 |
| #24 | `Sleuthkit` | - | 2 | - | - | - | - | - |

To detect such false positives during our evaluation, we employed various heuristics: First, we use the predicate's annotations to identify functions related to one of the five categories of false positives and discard them. Then, for each predicate, we inspect concrete values that crashes and non-crashes exhibit during execution. This allows us to compare actual values to the predicate's explanation and—together with the source code line—recognize semantic features such as loop counters or constants based on timers. Once a false positive is identified, we discard any propagation of the predicate's explanation and thereby subsequent related predicates. In our personal experience, these heuristics allow us to reliably detect many false positives without considering data-flow dependencies or other program context. This is supported by our results detailed in Table 5.3. Based on the five categories, we evaluate how many false positives within the top 50 predicates can be identified heuristically. Additionally, we denote the number of propagations as well as the number of false positives that must be analyzed in-depth. Note that an analyst had to conduct such an analysis for only half of the targets with false positives. We note that this may differ for other bugs or other target applications, especially edge cases such as bugs in the allocator or garbage collector.

Since we use a statistical model, false positives are a natural side effect, yet, precisely this noisy model is indispensable. For 15 of the analyzed bugs, we could find a perfect predicate (with a score of 1.0), i. e., predicates that perfectly distinguish crashes and non-crashes. In the remaining ten cases, some noise has been introduced by crash exploration. However, as our results indicate, small amounts of noise do not impair our

analysis. Therefore, we answer **RQ 3**, concluding that nearly all predicates found by AURORA are strongly related to the actual root cause.

Since the statistical model is only as good as the data it operates on, we also investigate the crash exploration and tracing phases. The results are presented in Table 5.4. Most traces produced by crash exploration could be traced successfully. The only exception being `Bash`, which caused many non-terminating runs that we excluded from subsequent analysis. Note that we were still able to identify the root cause.

We also investigate the time required for tracing, predicate analysis and ranking. We present the results in Table 5.5. On average, AURORA takes about 50 minutes for tracing, while the predicate analysis takes roughly 18 minutes and ranking four minutes. While these numbers might seem high, remember that the analysis is fully automated. In comparison, an analyst triaging these bugs might spend multiple days debugging specific bugs and identifying why the program crashes.

### 5.6.4 Case Studies

In the following, we conduct in-depth case studies of various software faults to illustrate different aspects of our approach.

#### 5.6.4.1 Case Study: Type Confusion in `mruby`

First, we analyze the results of our automated analysis for the example given in Section 5.2.1 (Bug #16). As described, the `NotImplementedError` type is aliased to the `String` type, leading to a type confusion that is hard to spot manually. Consequently, it is particularly interesting to see whether our automated analysis can spot this elusive bug. As exploring the behavior of `mruby` was challenging for AFL, we ran the initial crash exploration step for 12 hours in order to get more than 100 diversified crashes and non-crashes. Running our subsequent analysis on the best 50 predicates reported by AURORA, we manually found that all of the 50 predicates are related to the bug and provide insight into some aspects of the root cause.

The line with the predicate describing the location of the developers' fix is ranked 15th. This means that an analyst has to inspect 14 lines of code that are related to the bug but do not point to the developer fix. In terms of predicates, the 33rd predicate explains the root cause. This discrepancy results from the fact that one source code line may translate to multiple assembly instructions. Thus, multiple predicates may refer to values used in the same source code line.

The root cause predicate itself conditions on the fact that the minimal value in register `rax` is smaller than 17. Remember that the root cause is the missing type check. Types in `mruby` are implemented as `enum`, as visible in the following snippet of mruby's source code (`mruby/value.h`):

```
112    MRB_TT_STRING,      /* 16 */
113    MRB_TT_RANGE,       /* 17 */
114    MRB_TT_EXCEPTION,   /* 18 */
```

**Table 5.4:** Number of crashing (#c) and non-crashing (#nc) inputs found by crash exploration (Exploration) as well as the percentage of how many could be successfully traced (Tracing).

| | Target | Exploration | | Tracing | |
|---|---|---|---|---|---|
| | | #c | #nc | #c | #nc |
| #1 | `mruby` | 120 | 2708 | 100% | 99.9% |
| #2 | `Lua` | 398 | 1482 | 100% | 100% |
| #3 | `Perl` | 1591 | 6037 | 100% | 99.9% |
| #4 | `screen` | 858 | 2164 | 100% | 100% |
| #5 | `readelf` | 687 | 1803 | 100% | 100% |
| #6 | `mruby` | 809 | 3914 | 100% | 99.9% |
| #7 | `objdump` | 27 | 122 | 100% | 100% |
| #8 | `patch` | 266 | 886 | 74.8% | 89.7% |
| #9 | `Python` | 211 | 1546 | 100% | 100% |
| #10 | `tcpdump` | 161 | 619 | 100% | 100% |
| #11 | `NASM` | 2476 | 2138 | 100% | 100% |
| #12 | `Bash` | 842 | 5483 | 7.1% | 15.9% |
| #13 | `Bash` | 213 | 2102 | 50.7% | 55.5% |
| #14 | `Python` | 253 | 1695 | 98.0% | 98.2% |
| #15 | `nm` | 111 | 468 | 100% | 100% |
| #16 | `mruby` | 1928 | 4063 | 100% | 100% |
| #17 | `Python` | 705 | 2536 | 99.7% | 99.8% |
| #18 | `Xpdf` | 779 | 545 | 100% | 100% |
| #19 | `mruby` | 1128 | 2327 | 99.7% | 99.9% |
| #20 | `PHP` | 800 | 2081 | 100% | 100% |
| #21 | `libzip` | 36 | 286 | 100% | 100% |
| #22 | `mruby` | 1629 | 3557 | 100% | 99.9% |
| #23 | `NASM` | 590 | 1787 | 99.8% | 100% |
| #24 | `Sleuthkit` | 108 | 175 | 100% | 100% |
| #25 | `Lua` | 579 | 1948 | 100% | 100% |

Our identified root cause pinpoints the location where the developers insert their fix and semantically states that the type of the presumed exception object is smaller than 17. In other words, the predicate distinguishes crashes and non-crashes according to their type. As can be seen, the `String` type has a value of 16; thus, it is identified as crashing input, while the exception type is assigned 18. This explains the type confusion's underlying fault.

The other predicates allow tracing the path from the root cause to the crashing location. For example, the predicates rated best describe the freeing of an object within the garbage collector. This is because the garbage collector spots that `NotImplemenetedError` is changed to point to `String` instead of the original class. As a consequence, the garbage collector decides to free the struct containing the original class `NotImplementedError`, a very uncommon event. Subsequent predicates point to

**Table 5.5:** Time spent on tracing, predicate analysis (PA) and ranking of each target (in hours:minutes).

| | Target | Tracing | PA | Ranking |
|---|---|---|---|---|
| #1 | `mruby` | 01:08 | 00:19 | 00:04 |
| #2 | `Lua` | 00:09 | 00:03 | < 1 min |
| #3 | `Perl` | 00:53 | 01:52 | 00:17 |
| #4 | `screen` | 00:11 | 00:04 | < 1 min |
| #5 | `readelf` | 00:05 | 00:02 | < 1 min |
| #6 | `mruby` | 01:44 | 00:42 | 00:16 |
| #7 | `objdump` | < 1 min | < 1 min | < 1 min |
| #8 | `patch` | 00:36 | < 1 min | < 1 min |
| #9 | `Python` | 01:20 | 00:15 | 00:05 |
| #10 | `tcpdump` | 00:01 | < 1 min | < 1 min |
| #11 | `NASM` | 00:20 | 00:12 | 00:07 |
| #12 | `Bash` | 00:49 | 00:01 | 00:03 |
| #13 | `Bash` | 00:26 | 00:02 | 00:01 |
| #14 | `Python` | 01:23 | 00:14 | 00:08 |
| #15 | `nm` | 00:01 | < 1 min | < 1 min |
| #16 | `mruby` | 01:47 | 00:49 | 00:02 |
| #17 | `Python` | 04:03 | 00:55 | 00:03 |
| #18 | `Xpdf` | 00:19 | 00:01 | 00:03 |
| #19 | `mruby` | 01:58 | 00:21 | 00:22 |
| #20 | `PHP` | 01:16 | 00:47 | 00:03 |
| #21 | `libzip` | < 1 min | < 1 min | < 1 min |
| #22 | `mruby` | 01:57 | 00:49 | 00:16 |
| #23 | `NASM` | 00:10 | 00:03 | 00:02 |
| #24 | `Sleuthkit` | < 1 min | < 1 min | < 1 min |
| #25 | `Lua` | 00:11 | 00:07 | < 1 min |

locations where the string is attached to the presumed exception object during the raising of the exception. Additionally, predicates pinpoint the crashing location by stating that a crash will occur if the dereferenced value is smaller than a byte.

#### 5.6.4.2 Case Study: Heap Buffer Overflow in `readelf`

GNU Binutils' `readelf` application may crash as a result of a heap buffer overflow when parsing a corrupted MIPS option section [21]. This bug (Bug #5) was assigned CVE-2019-9077. Note that this bug only crashes when ASAN is used. Consequently, we use a binary compiled with ASAN for crash exploration but run subsequent tracing on a non-ASAN binary. The bug is triggered when parsing a binary input where a field indicates that the size is set to 1 despite the actual size being larger. This value is then processed further, amongst others, by an integer division where it is divided by $0x10$, resulting in a value of 0. The 0 is then used as size for allocating memory for some struct. More specifically, it is passed to the `cmalloc` function that delegates the

call to `xmalloc`. In this function, the size of 0 is treated as a special case where one byte should be allocated and returned. Subsequently, writing any data larger than one byte—which is the case for the struct the memory is intended for—is an out-of-bounds write. As no crucial data is overwritten, the program flow continues as normal unless it was compiled with ASAN, which spots the out-of-bounds write.

To prevent this bug, the developers introduced a fix where they check whether the allocated memory's size is sufficient to hold the struct. Analyzing the top 50 predicates, we observe that each of these predicates is assigned a score larger than or equal 0.99. Our seventh predicate pinpoints the fix by making the case that an input crashes if the value in `rcx` is smaller than 7. The other predicates allow us to follow the propagation until the crashing location. For instance, two predicates exist that point to the integer division by $0x10$, which causes the 0. The first predicate states that crashes have a value smaller than $0x7$ after the division. The second predicate indicates that the zero flag is set, demonstrating a use case for our flag predicates. We further see an edge predicate, which indicates that only crashes enter the special case, which is triggered when `xmalloc` is called with a size of 0.

#### 5.6.4.3 Case Study: Use-after-free in `Lua`

In version 5.3.5, a use-after-free bug (#25, CVE-2019-6706) was found in the `Lua` interpreter [5]. `Lua` uses so-called *upvalues* to implement closures. More precisely, upvalues are used to store a function's local variables that have to be accessed after returning from the function [121]. Two upvalues can be joined by calling `lua_upvaluejoin`. The function first decreases the first upvalue's reference count and, critically, frees it if it is not referenced anymore, before then setting the reference to the second upvalue. The function does not check whether the two passed parameters are equal, which semantically has no meaning. However, in practice, the upvalue will be freed before setting the reference, thus provoking a use-after-free. ASAN detects the crash immediately while regular builds crash with a segmentation fault a few lines later.

Our approach manages to create three predicates with a score of 1. All of these three predicates are edge predicates, i. e., detecting that for crashes, another path was taken. More precisely, for the very first predicate, we see the return from the function where the second upvalue's index was retrieved. Note that this is before the developers' fix, but the first point in the program where things go wrong. The second predicate describes the function call where the upvalue references are fetched, which are then compared for equality in the developer fix, i. e., it is located closely before the fix. The third predicate is located right after the developer fix; thus, we have to inspect three predicates or three source lines until we locate the developer fix. It describes the return from the function decreasing the reference count. All other predicates follow the path from the root cause to the crashing location.

#### 5.6.4.4 Case Study: Uninitialized Variable in `mruby`

The `mruby` interpreter contains a bug where uninitialized memory is accessed (Bug #19). This happens in the `unpack_m` function when unpacking a base64 encoded value from a

packed string. A local `char` array of size four is declared without initialization. Then, a state machine implemented as a while loop iterates over the packed string, processing it. The local `char` array is initialized in two states during this processing step. However, crafting a specific packed string allows to avoid entering these two states. Thereby, the local array is never properly initialized and MSAN aborts program execution upon the use of the uninitialized memory.

When analyzing the top 50 predicates, we find that they are are all related to the bug. The 16th predicate pinpoints the location where the developer fix is inserted. It describes that crashes fail to pass the condition of the while loop and—as a consequence—leave the loop with the local variable being uninitialized. Another predicate we identify pinpoints if the condition allows skipping the initialization steps, stating that this is a characteristic inherent to crashing inputs. All other predicates highlight locations during or after the state machine. Note that the crash only occurs within MSAN; thus, the binary we trace does not crash. However, this does not pose a problem for our analysis, which efficiently pinpoints root cause and propagation until the crashing and non-crashing runs no longer differ. In this particular case, the uninitialized memory is used to calculate a value that is then returned. For instance, we see that the minimal memory value written is less than $0x1c$ at some address. Consequently, our analysis pinpoints locations between the root cause and the usage of the uninitialized value.

#### 5.6.4.5 Case Study: Null Pointer Dereference in `NASM`

For `NASM` (#11, CVE-2018-16517), we analyze a logic flaw which results in a null pointer dereference that crashes the program. This happens because a pointer to a `label` is not properly initialized but set to `NULL`. The program logic assumes a later initialization within a state machine. However, this does not happen because of a non-trivial logic flaw. The developers fix this problem by a significant rewrite, changing most of the implementation handling labels (in total, 500 lines of code were changed). Therefore, we conclude that no particular line can be determined as the root cause; nevertheless, we investigate how our approach performs in such a scenario. This is a good example to demonstrate that sometimes defining the root cause can be a hard challenge even for a human.

Analyzing the top 50 predicates reported, we find that AURORA generates predicates pointing to various hotspots, which show that the label is not initialized correctly. More precisely, we identify a perfect edge predicate stating that the pointer is initially set to `NULL` for crashes. Subsequent predicates inform us that some function is called, which takes a pointer to the label as a parameter. They identify that for crashes the minimal value for `rdi` (the first function parameter in the calling convention) is smaller than $0x\text{ff}$. Immediately before the function attempts to dereference the pointer, we see that the minimal value of `rax` is smaller than $0x\text{ff}$, which indicates that the value was propagated. Afterward, a segmentation fault occurs as accessing the address 0 is illegal. In summary, we conclude that AURORA is useful to narrow down the scope even if no definite root cause exists.

## 5.7 Discussion

As our evaluation shows, our approach is capable of identifying and explaining even complex root causes where no direct correlation between crashing cause and root cause exists. Nevertheless, our approach is no silver bullet: It still reports some predicates that are not related to the root cause. Typically, this is caused by the crash exploration producing an insufficiently diverse set of test cases. This applies particularly to any input that was originally found by a grammar-based fuzzer since AFL's type of mutations may fail to produce sufficiently diverse inputs for such targets [44]. We expect that better fuzzing techniques will improve the ability to generate more suitable corpora. Yet, no matter how good the fuzzer is, in the end, pinpointing a single root cause will remain an elusive target for automated approaches: even a human expert often fails to identify a single location responsible for a bug.

Relying on a fuzzer illustrates another pitfall: We require that bugs can be reproduced within a fuzzing setup. Therefore, bugs in distributed or heavily concurrent systems currently cannot be analyzed properly by our approach. However, this is a limitation of the underlying fuzzer rather than Aurora: Our analysis would scale to complex systems spanning multiple processes and interacting components; our statistical model can easily deal with systems where critical data is passed and serialized by various means, including networks or databases, where traditional analysis techniques like taint tracking fail.

In some cases, the predicates that we generate might not be precise enough. While this situation did not happen during our evaluation, hypothetically, there may exist bugs that can only be explained by predicates spanning multiple locations. For example, one could imagine a bug caused by using an uninitialized value, which is only triggered if two particular conditions are met: The program avoids taking a path initializing the value and later takes a path where the value is accessed. Our single-location predicates fail to capture that the bug behavior is reliant on two locations. We leave extending our approach to more complex and compound predicates as an interesting question for future work.

Last, our system requires a certain computation time to identify and explain root causes. In some cases, Aurora ran for up to 17 hours (including 12 hours for crash exploration). We argue that this is not a problem, as our system is used in combination with normal fuzzing. Thus, an additional 17 hours of fuzzing will hardly incur a significant cost for typical testing setups. Since it took us multiple person-days to pinpoint the root cause for some of the bugs we analyzed, making the integration of our fully automated approach into the fuzzing pipeline seems feasible.

An integration to fuzzing could benefit the fuzzer: Successful fuzzing runs often produce a large number of crashing inputs, many of which trigger the same crash. To save an analyst from being overwhelmed, various heuristics are deployed to identify equivalent inputs. Most rely on the input's *coverage profile* or *stack hashing* where the last $n$ entries of the call stack are hashed [134]. Unfortunately, both techniques have been shown to be imprecise, i. e., to report far too many distinct bugs, while sometimes even joining distinct bugs into one equivalence class [134]. Given an automated approach

capable of identifying the root cause such as ours, it is possible to bucket crashing inputs according to their root cause. To this end, one could pick some random crashing input, identify its root cause and then check for all remaining crashing inputs whether the predicate holds true. Each crashing input for which the predicate is evaluated to true is then collected in one bucket. For the remaining inputs, the process could be repeated until all crashing inputs have been sorted into their respective equivalence classes.

In some cases, such as closed-source binaries or a limited amount of developer time, manually implementing fixes may be impossible. An automated approach to providing (temporary) patches may be desirable. Our approach could be extended to patch the root cause predicate into the binary such that—at the point of the root cause—any input crashing the binary leads to a graceful exit rather than a potentially exploitable crash.

## 5.8 Related Work

In the following, we focus on works related closest to ours, primarily statistical and automated approaches.

**Spectrum-based Fault Localization.** Closest related to our work are so-called spectrum-based, i. e., code coverage-based, fault localization techniques [75]. In other words, these approaches attempt to pinpoint program elements (on different levels, e. g., single statements, basic blocks or functions) that cause bugs. To this end, they require multiple inputs for the program, some of which must crash while others may not. Often, they use test suites provided by developers and depend on the source code being available. For instance, Zhang et al. [234] describe an approach to root cause identification targeting the Java Virtual Machine: first, they locate the non-crashing input from provided test suite whose control flow paths beginning overlaps the most with the one of the crashing input under investigation. Then, they determine the location of the first deviation, which they report as the root cause. Overall, most approaches either use some metric [26, 27, 126, 218, 219] to identify and rank possible root causes or rely on statistical techniques [145, 148, 162].

As a sub-category of spectrum-based fault localization, techniques based on statistical approaches use predicates to reason about provided inputs. Predicate-based techniques are used to isolate bugs[145] or to pinpoint the root cause of bugs [148, 162, 234]. These approaches typically require source code and mostly rely on inputs provided by test suites.

While our work is similar to such approaches with respect to sampling predicates and statistically isolating the root cause, our approach does not require access to source code since it solely works on the binary level. Furthermore, our analysis synthesizes domain-specific predicates tailored to the observed behavior of a program. Also, we do not require any test suite but rely on a fuzzer to generate test cases. This provides our approach with a more diversified set of inputs, allowing for more fine-grained analysis.

**Reverse Execution.** A large number of works [68, 69, 105, 161, 221] investigate the problem of analyzing a crash, typically starting from a core dump. To this end, they reverse-execute the program, reconstructing the data flow leading to the crash. To achieve this, CREDAL [220] uses a program's source code to automatically enrich the core dump analysis with information aiding an analyst in finding memory corruption vulnerabilities. Further reducing the manual effort needed, POMP requires a control-flow trace and crash dump, then uses backward taint analysis [221] to reverse the data flow, identifying program statements contributing to a crash. In a similar vein but for a different application scenario—core dumps sampled on an OS level—RETracer [68] uses backward taint analysis without a trace to reconstruct functions on the stack contributing to a crash. Improving upon RETracer, Cui et al. [69] developed REPT, an *reverse debugger* that introduces an error correction mechanism to reconstruct execution history, thereby recovering data flow leading to a crash. To overcome inaccuracies, Guo et al. [105] propose a deep-learning-based approach based on value-set analysis to address the problem of memory aliasing.

While sharing the goal of identifying instructions causing a crash, Aurora differs from these approaches by design. Reverse execution starts from a crash dump, reversing the data-flow, thereby providing an analyst with concrete assembly instructions contributing to a bug. While these approaches are useful in scenarios where a crash is not reproducible, we argue that most of them are limited to correctly identify bugs that exhibit a direct data dependency between root cause and crashing location. While REPT does not rely on such a dependency, it integrates into an interactive debugging session rather than providing a list of potential root cause predicates; thus, it is orthogonal to our approach. Moreover, Aurora uses a statistical analysis to generate predicates that not only pinpoint the root cause but also add an explanation describing how crashing inputs behave at these code locations. Furthermore, since we do not perform a concrete analysis of the underlying code, Aurora can spot vulnerabilities with no direct data dependencies.

## 5.9 Conclusion

In this chapter, we introduced and evaluated a novel binary-only approach to automated root cause explanation. In contrast to other approaches that identify program instructions related to a program crash, we additionally provide semantic explanations of how these locations differ in crashing runs from non-crashing runs. Our evaluation shows that we are able to spot root causes for complex bugs such as type confusions where previous approaches failed. Given debug information, our approach is capable of enriching the analysis' results with additional information. We conclude that Aurora is a helpful addition to identify and understand the root cause of diverse bugs.

# Chapter 6

# Conclusion

Due to the undecidability of reasoning about software security in general, analysis techniques are often goal-oriented, effectively limiting the analysis scope to specific aspects. Some reasoning techniques are based on abstraction in which we transform parts of a program into an abstract domain that is explicitly constructed to facilitate reasoning about specific characteristics. In this scenario, a behavioral substitute represents the program's behavioral semantics in the abstract domain.

In this thesis, we developed problem-specific analysis techniques based on synthesized behavioral substitutes related to code deobfuscation, fuzzing and root cause analysis. In each case, we first designed a domain-specific representation that allowed us to represent a wide range of problem instances in its associated area. Then, we applied stochastic program synthesis techniques to learn behavioral substitutes automatically. Using the target's program behavior as feedback, we were able to craft target-specific representations. By combining target-specific representations in a problem-specific domain, we were able to reason about generic instances in the problem space while performing an analysis tailored to a specific target. As a consequence, we advanced research on different topics in software security by introducing methods based on stochastic processes or statistical analysis, partially outperforming state-of-the-art techniques or introducing orthogonal approaches to existing solutions.

In the following, we discuss the results of this thesis in further detail and proceed to describe the limitations of our approaches. Finally, we outline directions for future research that are based on the techniques proposed in this thesis.

**Expression Synthesis to Simplify Obfuscated Code.** Chapter 3 introduced SYNTIA, a code deobfuscation approach based on program synthesis. Since existing code deobfuscation approaches rely on a precise analysis of the underlying code, they can be thwarted by specific code transformations. Our semantics-based approach can deobfuscate code of arbitrary syntactic complexity because it uses the obfuscated code as a black-box to observe its input-output behavior. Based on this I/O behavior, SYNTIA learns syntactically simpler expressions with the same semantics using Monte Carlo Tree Search. We demonstrated the feasibility of our approach by synthesizing instruction semantics of industrial-grade, state-of-the-art VM-based obfuscators. Furthermore, we

simplified syntactically complex arithmetic expressions and ROP gadgets, demonstrating the general nature of our approach.

**Input Structure Synthesis to Guide Feedback-driven Fuzzing.** In Chapter 4, we developed Grimoire, a fuzzer that learns syntactical patterns of structured input languages and produces new structured inputs of the target language using structure-aware mutations. To learn these patterns, it uses the inputs' code coverage as feedback and dissects the inputs by removing parts that are irrelevant to the observed coverage. Combined with structure-aware mutations, Grimoire outperformed other state-of-the-art fuzzers on generic targets that expect structured inputs (e. g., language interpreters and parsers). Furthermore, we increased the test coverage of target-specific structured fuzzers by using their corpus as a seed.

**Predicate Synthesis to Automate Root Cause Explanation.** In Chapter 5, we presented Aurora, a generic approach that employs statistical analysis to pinpoint the root cause for various types of bugs. Based on a set of trace information representing program behavior for crashing and non-crashing program runs, Aurora synthesizes Boolean predicates that distinguish crashes from non-crashes, effectively predicting if a provided input will crash or not. Ranking these predicates according to their accuracy and average execution order allows a human analyst to precisely pinpoint the root cause. In our evaluation, we demonstrated this on a wide variety of real-world targets and complex bugs, including type confusions, use-after-frees, heap buffer overflows and uninitialized variables.

## 6.1 Limitations

The results presented in this thesis emphasize the benefits of combining problem-specific analysis techniques with target-specific behavioral substitutes. Despite all advantages, these approaches have some significant drawbacks that we discuss in the following. First, we cover constraints all techniques are subject to. Afterward, we discuss shortcomings of our solutions presented for code deobfuscation, fuzzing and root cause analysis.

In general, reasoning about security aspects of software is undecidable [182]. Therefore, common approaches limit the scope to specific aspects only. Although our analysis techniques were modeled to be problem-specific, they only covered a subset of open problems in their associated areas. As a consequence, problem-specific techniques have to be precisely designed to operate in a defined scope; this is often a labour-intensive manual task. Furthermore, we apply methods based on program synthesis to obtain target-specific behavioral substitutes. For this, we require some kind of feedback from the target application. This feedback might not be easy to obtain depending on the context the analyzed program runs in, such as some (deeply) embedded systems or devices running on undocumented architectures. Finally, as a general limitation of program synthesis, our techniques are limited by semantic complexity, meaning we can only synthesize semantically „simple“ constructs.

**Code Deobfuscation.** In the field of code deobfuscation, our approach is directly limited by the constraints of stochastic program synthesis. In general, synthesis results

may not be precise, since they are approximated by randomly sampled input-output pairs. Random sampling may not cover the full semantics, as it can miss edge cases. For instance, in the case of point functions—in which only a single input triggers different behavior—it is unlikely to guess the correct input. Another limitation is non-deterministic behavior: if an oracle returns different outputs for the same input, I/O samples cannot be representative. Finally, it is not guaranteed that a synthesizer will find any solution at all. If expressions cannot be expressed by the underlying grammar or if expressions are semantically too complex, the synthesizer can only learn *partial expressions*. Nonetheless, in practice, these techniques still provide useful insights for human analysts.

**Fuzzing.** In the area of fuzzing, our presented approach is also restricted in terms of complexity. While it works well on deeply nested and syntactically complex constructs, it is still too coarse-grained to identify constructs such as opening and closing tags (e. g., as seen in XML) and variable uses. Furthermore, it cannot find semantically complex as those that require nested memory allocations and deallocations before anything „interesting“ happens. Finally, the general limitations of fuzzing also apply to our methods, such as an application's native execution time, fuzzing distributed/concurrent systems and the ability to actually execute code.

**Root Cause Analysis.** As for root cause analysis, our techniques are also limited by semantic complexity. In some cases, our generated predicates might not be precise enough to explain a bug. While we pinpoint single locations, there may exist bugs that depend on many different conditions that must be true at the same time. Furthermore, our approach still reports some predicates that are not related to the root cause. Typically, this is caused by crash exploration producing an insufficiently diverse set of test cases. In addition, our approach requires the bug to be reproducible in a fuzzing setup, which is, for instance, not the case in scenarios such as distributed and heavily concurrent systems.

## 6.2 Future Work

While we presented problem-specific techniques based on behavioral substitutes in the fields of code deobfuscation, fuzzing and root cause analysis, there are other domains where such methods might be beneficial as well. Furthermore, although our techniques were designed to be generic in their respective domain, they only covered some of the problems in their associated field. After having discussed the limitations of our approaches in the previous section, we now present general as well as project-specific ideas for future research.

Since program synthesis is actively researched, any progress in that area has a direct impact on the synthesis of behavioral substitutes. The more powerful synthesis techniques are, the better semantically complex constructs can be synthesized. For instance, *stratified synthesis* [112] might improve program synthesis significantly. The main idea of this approach is to incrementally synthesize larger parts of the instruction trace based on previous results and, thus, successively approximate high-level semantics of larger expressions. Related to software security, research has started to employ program syn-

thesis for automated exploit generation, such as recent work by Heelan et al. [109, 110]: they demonstrated automated techniques to chain exploitation primitives for heap exploitation. Another research area that benefits from program synthesis is *automated reverse engineering*: we could synthesize binary decoders [176] for unknown instruction set architectures as well as nested data structures for type inference [190].

**Code Deobfuscation.** For code deobfuscation, our program synthesis approach based on Monte Carlo Tree Search produced excellent results. However, comparing it to other stochastic algorithms such as Simulated Annealing, *beam search* [181] and *late acceptance hill climbing* [52] might be insightful. Recent research by David et al. [73] performed expression simplification based on enumerative program synthesis. Using a dynamic trace and concolic execution, they incrementally simplified subexpressions of the derived abstract syntax tree. While their approach worked well in a dynamic setting, it would be interesting to apply this approach in a static scenario.

**Fuzzing.** In order to improve our fuzzing approach, future work might explore more strategies for learning syntactically complex constructs: for instance, one could apply different learning strategies that fixate and mutate multiple occurrences of a substring at the same time to detect patterns such as XML tags and variables. This could be combined with generalization techniques more expressive than learning gaps, such as semantic placeholders for variables, size fields and other constructs. In our case, we used all triggered coverage bytes as synthesis feedback. In a more fine-grained approach, one could consider using only single coverage bytes from the input and combining them incrementally to get more exact results.

**Root Cause Analysis.** For root cause analysis, the most exciting future research direction might be exploring techniques for synthesizing compound predicates that combine different single-location predicates. This way, one could isolate the faulty behavior of bugs that require more than one condition to be `true` at the same time. Further research could also focus on more fine-grained crash exploration techniques that improve the input diversity for a given crash. More applied research could integrate root cause analysis into a fuzzing framework. Fuzzing often produces a large number of crashing inputs even though many of these inputs trigger the same bug. Given an approach for automated root cause identification, it would be interesting to bucket crashing inputs according to their root cause. To this end, one could pick some random crashing input, identify its root cause and then check for all remaining crashing inputs whether the predicate holds true. Finally, we could use synthesized predicates for automated patching. In this case, we would insert a predicate at the root cause. If it evaluates to `true`, forecasting a crash, we perform a graceful exit instead of potentially triggering an exploitable crash.

In summary, we believe that the analysis techniques developed in this thesis provide a solid baseline and advance research in their associated areas. We hope that they offer a fruitful ground for further exploration and inspire other researchers to simplify problem-specific reasoning about software security.

# List of Figures

2.1 Local search with an acceptance function approximates a global optimum (the darkest area in the map). . . . . . . . . . . . . . . . . . . . . . . . 16
2.2 Illustration of a single MCTS round (taken from Browne et al. [49]). . . 18

3.1 The Fetch–Decode–Execute cycle of a Virtual Machine. Native code calls into the VM, upon which startup code is executed (VM entry). It performs the context switch from native to VM context. Then, the next instruction is fetched from the bytecode stream, mapped to the corresponding handler using the handler table (decoding) and, finally, the handler is executed. The process repeats for subsequent VM instructions in the bytecode until the exit handler is executed, which returns back to native code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.2 Dissecting a given trace (a) into several trace windows (b). The trace windows can be used to reconstruct a (possibly disconnected) control-flow graph (c). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.3 An MCTS tree for program synthesis that grows towards the most-promising node $b\ a\ +$, the right-most leaf in layer 3. . . . . . . . . . . . 32
3.4 The left-most $U$ in $U_3\ U_2\ U_1\ *\ +$ is the top-most-right-most non-terminal in the abstract syntax tree. (The indices are provided for illustrative purposes only.) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.5 Subsequent synthesis runs increase the number of synthesized MBA expressions. Each point represents the average cumulative number of synthesized expressions from 15 separate experiments. . . . . . . . . . . 38

4.1 A high-level overview of our mutations. Given an input, we apply various mutations on its generalized and original form. Each mutation then feeds mutated variants of the input to the fuzzer's execution engine. . . . . . 55
4.2 The coverage (in basic blocks) produced by various tools over 12 runs for 48h on various targets. Displayed are the median and the 66.7% intervals. 62
4.3 The coverage (in basic blocks) produced by GRIMOIRE and NAUTILUS (using the hand written grammars of the authors of NAUTILUS) over 12 runs at 48 h on various targets. The incremental plots show how running NAUTILUS for 48h compares to running NAUTILUS for the first 24h and then continue fuzzing for 24h with GRIMOIRE. Displayed are the median and the 66.7% confidence interval. . . . . . . . . . . . . . . . . . . . . 65

4.4 Comparing GRIMOIRE against GLADE (median and 66.7% interval). In the plot for GLADE +Training, we include the training time that glade used. For comparison, we also include plots where we omit the training time. The horizontal bar displays the coverage produced by the seed corpus that GLADE used during training. . . . . . . . . . . . . . . . . . . 66

# List of Tables

3.1 Initial Simulated Annealing configuration and the parameter's lower/upper bounds. . . . . . . . . . 35
3.2 Parameter choices for different complexity classes that depend on the expression layer and the number of variables. The parameters are the SA-UCT parameter (SA), the maximum number of MCTS iterations (# iter), the number of I/O samples (# I/O) and the playout depth (PD). 36
3.3 Trace window statistics and synthesis performance for Tigress (MBA), VMProtect (VMP), Themida (flavor Tiger White, TM), and ROP gadgets. 37

4.1 Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against the best alternative. The effect size is the difference of the medians in basic blocks. In most experiments, the effect size is relevant and the changes are highly significant: it is typically multiple orders of magnitude smaller than the usual bound of $p < 5.0E\text{-}02$ (bold). . . . . . . . . . 61
4.2 Statistics on basic block coverage for tested fuzzers. In the column „Best Coverage“, we provide the highest number of basic blocks a run found and the percentage relative to the number of basic blocks obtained from `IDA Pro [113]`. . . . . . . . . . 63
4.3 Confirmatory data analysis of our experiment. We compare the coverage produced by GRIMOIRE against NAUTILUS with hand written grammars. The effect size is the difference of the medians in basic blocks in the incremental experiment. In three experiments, the effect size is relevant and the changes are highly significant (marked bold, $p < 5.0E\text{-}02$). Note that we abbreviate `JavaScriptCore` with `JSC`. . . . . . . . . . 66
4.4 Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against GLADE. The effect size is the difference of the medians in basic blocks. In all experiments, the effect size is relevant and the changes are highly significant: it is multiple orders of magnitude smaller than the usual bound of $p < 5.0E\text{-}02$ (bold). . . . . 68
4.5 Statistics for each of GRIMOIRE's mutation strategies (i. e., our structured mutations, REDQUEEN's Input-to-State-based mutations and havoc). For every target evaluated we list the total number of inputs found by a mutation, the time spent on this strategy and the ratio of inputs found per minute. . . . . . . . . . 69

4.6 Overview of submitted bugs and CVEs. Fuzzers which did not find the bug during our evaluation are denoted by ✗, while those who did are marked by ✓. We indicate targets not evaluated by a specific fuzzer with '-'. We abbreviate Use-After-Free (UAF), Out-of-Bounds (OOB) and Buffer Overflow (BO). . . . . . . . . . . . . . . . . . . . . . . . . . 70

5.1 Results of our root cause explanations. For 25 different bugs, we note the target, root and crashing cause as well as whether the target has been compiled using a sanitizer. Furthermore, we provide the number of predicates and source lines (SLOC) a human analyst has to examine until the location is reached where the developers applied the bug fix (denoted as *Steps to Dev. Fix*). Finally, the number of true and false positives (denoted as TP and FP) of the top 50 predicates are shown. * describes targets where no top 50 predicates with a score above or equal to 0.9 exist. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

5.2 Maximum and average distance between developer fix and crashing location in both all and unique executed assembly instructions. For reference, the average amount of instructions executed between program start and crash is also provided. . . . . . . . . . . . . . . . . . . . . . . . 89

5.3 Analysis results of false positives within the top 50 predicates. For each target, we classify its false positives into the categories they are related to: allocation or garbage collector (Alloc), control flow (CF), data structure (DS), environment (Env) or error handling (Error). Additionally, we track the number of predicates an analyst has to inspect in more detail (In-depth Analysis) as well as propagations of false positives that can be discarded easily. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

5.4 Number of crashing (#c) and non-crashing (#nc) inputs found by crash exploration (Exploration) as well as the percentage of how many could be successfully traced (Tracing). . . . . . . . . . . . . . . . . . . . . . . 93

5.5 Time spent on tracing, predicate analysis (PA) and ranking of each target (in hours:minutes). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

# Bibliography


[1] Bash segmentation fault. https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00044.html, .

[2] Bash segmentation fault. https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00042.html, .

[3] libzip use-after-free (CVE-2017-12858). https://blogs.gentoo.org/ago/2017/09/01/libzip-use-after-free-in-_zip_buffer_free-zip_buffer-c/, .

[4] Lua heap buffer overflow. https://www.lua.org/bugs.html#5.0-2, .

[5] Lua use-after-free (CVE-2019-6706). https://security-tracker.debian.org/tracker/CVE-2019-6706, .

[6] mruby heap buffer overflow (CVE-2018-12248). https://github.com/mruby/mruby/issues/4038, .

[7] mruby heap buffer overflow (CVE-2018-10191). https://github.com/mruby/mruby/issues/3995, .

[8] mruby type confusion. https://hackerone.com/reports/185041, .

[9] mruby use-after-free (CVE-2018-10199). https://github.com/mruby/mruby/issues/4001, .

[10] mruby uninitialized variable. https://github.com/mruby/mruby/issues/3947, .

[11] NASM nullpointer dereference (CVE-2018-16517). https://nafiez.github.io/security/2018/09/18/nasm-null.html, .

[12] NASM use-after-free. https://bugzilla.nasm.us/show_bug.cgi?id=3392556, .

[13] nm stack buffer overflow. https://sourceware.org/bugzilla/show_bug.cgi?id=21670, .

[14] objdump heap over flow (CVE-2017-9746). https://sourceware.org/bugzilla/show_bug.cgi?id=21580, .

[15] patch heap buffer overflow. https://savannah.gnu.org/bugs/?func=detailitem&item_id=54558, .

[16] Perl heap buffer overflow. https://github.com/Perl/perl5/issues/17384, .

[17] PHP uninitialized variable (CVE-2019-11038). https://bugs.php.net/bug.php?id=77973, .

[18] Python heap buffer overflow (CVE-2016-5636). https://bugs.python.org/issue26171, .

[19] Python segmentation fault. https://bugs.python.org/issue31530, .

[20] Python type confusion. https://hackerone.com/reports/116286, .

[21] readelf heap buffer overflow (CVE-2019-9077). https://sourceware.org/bugzilla/show_bug.cgi?id=24243, .

[22] screen heap buffer overflow. https://seclists.org/oss-sec/2020/q1/65, .

[23] Sleuthkit use-after-free. https://github.com/sleuthkit/sleuthkit/issues/905, .

[24] tcpdump heap buffer overflow (CVE-2017-16808). https://github.com/the-tcpdump-group/tcpdump/issues/645, .

[25] Xpdf uninitialized variable. https://forum.xpdfreader.com/viewtopic.php?f=3&t=41890, .

[26] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[27] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Localizing software faults simultaneously. In *International Conference on Quality Software*, 2009.

[28] P. W. Adriaans and M. M. van Zaanen. Computational grammatical inference. In *Innovations in Machine Learning*. Springer, 2006.

[29] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and W. Meredith. Announcing oss-fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[30] D. Andriesse, H. Bos, and A. Slowinska. Parallax: Implicit Code Integrity Verification using Return-Oriented Programming. In *Conference on Dependable Systems and Networks (DSN)*, 2015.

[31] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 1987.

[32] Apple Inc. JavaScriptCore. https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore.

[33] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[34] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[35] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[36] S. Bansal and A. Aiken. Automatic Generation of Peephole Superoptimizers. In *ACM Sigplan Notices*, 2006.

[37] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[38] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[39] J. R. Bell. Threaded Code. *Communications of the ACM*, 1973.

[40] F. Bellard. TCC: Tiny C compiler. `https://bellard.org/tcc/`.

[41] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 1972.

[42] T. Blazytko. Static data flow analysis and constraint solving to craft inputs for binary programs. Master's thesis, Ruhr-Universität Bochum, 2015.

[43] T. Blazytko, M. Contag, C. Aschermann, and T. Holz. Syntia: Synthesizing the semantics of obfuscated code. In *USENIX Security Symposium*, 2017.

[44] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz. Grimoire: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[45] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz. Aurora: : Statistical crash analysis for automated root cause explanation. In *USENIX Security Symposium*, 2020.

[46] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[47] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[48] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT–A Formal System for Testing and Debugging Programs by Symbolic Execution. *ACM SigPlan Notices*, 1975.

[49] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte

Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[50] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[51] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[52] E. K. Burke and Y. Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017.

[53] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[54] L. Cavallaro, P. Saxena, and R. Sekar. Anti-Taint-Analysis: Practical Evasion Techniques against Information Flow based Malware Defense. *Secure Systems Lab at Stony Brook University, Tech. Rep*, 2007.

[55] T. Cazenave. Monte carlo beam search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[56] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012.

[57] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[58] G. Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Universiteit Maastricht, 2010.

[59] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[60] K. Claessen, N. Smallbone, and J. Hughes. Quickspec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*, pages 6–21. Springer, 2010.

[61] C. Collberg, S. Martin, J. Myers, and B. Zimmerman. Documentation for Data Encodings in Tigress. `http://tigress.cs.arizona.edu/transformPage/docs/encodeData`, .

[62] C. Collberg, S. Martin, J. Myers, and B. Zimmerman. Documentation for Arithmetic Encodings in Tigress. `http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic`, .

[63] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997.

[64] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[65] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed Application Tamper Detection via Continuous Software Updates. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[66] K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[67] Y. Crama, A. W. Kolen, and E. Pesch. Local search in combinatorial optimization. In *Artificial Neural Networks*, pages 157–174. Springer, 1995.

[68] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering (ICSE)*, 2016.

[69] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. REPT: Reverse debugging of failures in deployed software. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2018.

[70] CVE-2014-0160. The heartbleed bug. `https://heartbleed.com/`, 2014.

[71] CVE-2014-6271. The shellshock bug. `https://seclists.org/oss-sec/2014/q3/649`, 2014.

[72] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. In *Algebraic methodology and software technology*, pages 81–95. Springer, 2006.

[73] R. David, L. Coniglio, and M. Ceccato. Qsynth: A program synthesis based approach for binary code deobfuscation. 2020.

[74] L. De Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer, 2009.

[75] H. A. de Souza, M. L. Chaim, and F. Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.

[76] W. Drewry and T. Ormandy. Flayer: Exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*. USENIX Association, 2007.

[77] T. Dullien and S. Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis, 2009.

[78] M. Eddington. Peach fuzzer: Discover unknown vulnerabilities. `https://www.peach.tech/`.

[79] S. Embleton, S. Sparks, and R. Cunningham. Sidewinder: An Evolutionary Guidance System For Malicious Input Crafting. `https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Embleton.pdf`, 2006.

[80] N. Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2017.

[81] N. Eyrolles, L. Goubin, and M. Videau. Defeating MBA-based Obfuscation. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

[82] H. Finnsson. Generalized Monte-Carlo Tree Search Extensions for General Game Playing. In *AAAI Conference on Artificial Intelligence*, 2012.

[83] F. S. Foundation. GNU Binutils. `https://www.gnu.org/software/binutils/`.

[84] Free Software Foundation. GNU Bison. `https://www.gnu.org/software/bison/`.

[85] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[86] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[87] S. Gao, S. Kong, and E. M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In *International conference on automated deduction*, pages 208–214. Springer, 2013.

[88] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz. Towards automated generation of exploitation primitives for web browsers. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[89] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*, 2012.

[90] GNU Project. GCC, the GNU compiler collection. `https://gcc.gnu.org/`.

[91] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *ACM SIGPLAN Notices*, 2012.

[92] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[93] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[94] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[95] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59, 2017.

[96] P. Goodman. Shin GRR: Make fuzzing fast again. `https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/`.

[97] Google LLC. V8. https://v8.dev/.

[98] R. Gopinath, B. Mathis, M. Höschele, A. Kampmann, and A. Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.

[99] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[100] M. Graziano, D. Balzarotti, and A. Zidouemba. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[101] A. Guinet, N. Eyrolles, and M. Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack Conference*, 2016.

[102] S. Gulwani. Dimensions in Program Synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 2010.

[103] S. Gulwani and N. Jojic. Program verification as probabilistic inference. *ACM SIGPLAN Notices*, 2007.

[104] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. *ACM SIGPLAN Notices*, 2011.

[105] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *USENIX Security Symposium*, 2019.

[106] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[107] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[108] H. Han, D. Oh, and S. K. Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[109] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *USENIX Security Symposium*, 2018.

[110] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[111] A. Helin. A general-purpose fuzzer. `https://github.com/aoh/radamsa`.

[112] S. Heule, E. Schkufza, R. Sharma, and A. Aiken. Stratified synthesis: Automatically Learning the x86-64 Instruction Set. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[113] Hex-Rays. IDA pro. `https://www.hex-rays.com/products/ida/`.

[114] D. R. Hipp. SQLite. `https://www.sqlite.org/index.html`.

[115] S. Hocevar. zzuf. `https://github.com/samhocevar/zzuf`.

[116] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, 2012.

[117] H. Hoos, T. Stützle, and E. I. Inc. *Stochastic Local Search: Foundations and Applications*. The Morgan Kaufmann Artificial. Elsevier Science, 2005. ISBN 9781558608726.

[118] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[119] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[120] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo. Lua. `https://www.lua.org/`.

[121] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. The implementation of Lua 5.0. *J. UCS*, 11(7):1159–1176, 2005.

[122] Intel Corporation. Pin – a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[123] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided Component-based Program Synthesis. In *ACM/IEEE 32nd International Conference on Software Engineering*, 2010.

[124] S. Johnson. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/yacc/`.

[125] E. Jones, T. Oliphant, and P. Peterson. Scipy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–.

[126] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[127] G. Katz and D. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *International Symposium on Automated Technology for Verification and Analysis*, 2008.

[128] D.-W. Kim, K.-H. Kim, W. Jang, and F. F. Chen. Unrelated Parallel Machine Scheduling with Setup Times using Simulated Annealing. *Robotics and Computer-Integrated Manufacturing*, 2002.

[129] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364. IEEE, 2017.

[130] J. Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *IEEE Working Conference on Reverse Engineering (WCRE)*, 2012.

[131] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[132] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 214–228. Springer, 2009.

[133] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 1983.

[134] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[135] P. Klint. Interpretation Techniques. *Software, Practice and Experience*, 1981.

[136] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2019.

[137] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning*, 2006.

[138] B. Kollenda, E. Göktaş, T. Blazytko, P. Koppe, R. Gawlik, R. K. Konoth, C. Giuffrida, H. Bos, and T. Holz. Towards automated discovery of crash-resistant primitives in binary executables. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

[139] S. Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique, 2005.

[140] D. Kroening, R. Bryant, and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2008. ISBN 9783540741046.

[141] T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd international conference on Knowledge capture*, pages 36–43, 2003.

[142] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: Program-state based binary fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[143] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A dpll (t) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification*, pages 646–662. Springer, 2014.

[144] P. Liberatore. The Complexity of Checking Redundancy of CNF Propositional Formulae. In *International Conference on Agents and Artificial Intelligence*, 2002.

[145] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[146] J. Lim and S. Yoo. Field Report: Applying Monte Carlo Tree Search for Program Synthesis. In *International Symposium on Search Based Software Engineering*, 2016.

[147] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[148] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[149] LLVM Project. Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`.

[150] K. Lu, S. Xiong, and D. Gao. RopSteg: Program Steganography with Return Oriented Programming. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[151] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao. Software Watermarking using Return-Oriented Programming. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[152] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1980.

[153] Marc, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud. Nested Monte-Carlo Search. *Communications of the ACM*, 2012.

[154] H. Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 1987.

[155] Y. Matsumoto. mruby. `http://mruby.org/`.

[156] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[157] Microsoft. ChakraCore. `https://github.com/Microsoft/ChakraCore`.

[158] Microsoft Research. The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`, 2020.

[159] J. Ming, D. Xu, L. Wang, and D. Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768, 2015.

[160] Mozilla Foundation / Mozilla Corporation. SpiderMonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[161] D. Mu, Y. Du, J. Xu, J. Xu, X. Xing, B. Mao, and P. Liu. POMP++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering*, 2019.

[162] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[163] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan Notices*, 2007.

[164] F. Nielson and N. Jones. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of logic in computer science*, 4:527–636, 1994.

[165] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2015.

[166] R. P. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1985.

[167] OpenRCE. Sulley: A pure-python fully automated and unattended fuzzing framework. `https://github.com/OpenRCE/sulley`.

[168] Oreans Technologies. Themida – Advanced Windows Software Protection System. `http://oreans.com/themida.php`.

[169] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv preprint arXiv:1812.00078*, 2018.

[170] pakt. ROPC: A Turing complete ROP compiler. `https://github.com/pakt/ropc`.

[171] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[172] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture Bug Search in Binary Executables. In *IEEE Symposium on Security and Privacy*, 2015.

[173] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury. Smart greybox fuzzing, 2018.

[174] Plaid CTF. ROP Challenge "quite quixotic chest". `https://ctftime.org/task/2305`.

[175] Python Software Foundation. Python. `https://www.python.org/`.

[176] W. Qin and S. Malik. Automated synthesis of efficient binary decoders for retargetable software toolkits. In *Design Automation Conference*, 2003.

[177] N. A. Quynh and D. H. Vu. Unicorn – The Ultimate CPU Emulator. `http://www.unicorn-engine.org`.

[178] N. A. Quynh, T. S. Di, B. Nagy, and D. H. Vu. Capstone Engine. `http://www.capstone-engine.org`.

[179] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2017. URL `https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf`.

[180] A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[181] D. R. Reddy et al. Speech understanding systems: A summary of results of the five-year research effort. *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA*, 17, 1977.

[182] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[183] R. Rolles. Unpacking Virtualization Obfuscators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[184] R. Rolles. Program Synthesis in Reverse Engineering. `http://www.msreverseengineering.com/blog/2014/12/12/program-synthesis-in-reverse-engineering`, 2014.

[185] R. Rolles. Synesthesia: A Modern Approach to Shellcode Generation. `http://www.msreverseengineering.com/blog/2016/11/8/synesthesia-modern-shellcode-synthesis-ekoparty-2016-talk`, 2016.

[186] J. Ruderman. Introducing jsfunfuzz. *`http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz`*, 2007.

[187] B. Ruijl, J. A. M. Vermaseren, A. Plaat, and H. J. van den Herik. Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. In *International Conference on Agents and Artificial Intelligence*, 2014.

[188] P. Rümmer and T. Wahl. An smt-lib theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, page 151, 2010.

[189] J. Ruohonen and K. Rindell. Empirical notes on the interaction between continuous kernel fuzzing and development. *arXiv preprint arXiv:1909.02441*, 2019.

[190] T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos. Dsibin: Identifying dynamic data structures in c/c++ binaries. In *International Conference on Automated Software Engineering (ASE)*, 2017.

[191] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. *Nicta*, 2013.

[192] M. P. Schadd, M. H. Winands, M. J. Tak, and J. W. Uiterwijk. Single-player Monte-Carlo Tree Search for SameGame. *Knowledge-Based Systems*, 2012.

[193] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. *ACM SIGPLAN Notices*, 2013.

[194] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium*, 2017.

[195] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[196] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, 2011.

[197] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[198] H. 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 (CCS)*, 2007.

[199] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy*, 2009.

[200] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[201] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 2016.

[202] Sony DADC. SecuROM Software Protection. https://www2.securom.com/Digital-Rights-Management.68.0.html.

[203] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2010.

[204] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[205] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[206] R. Swiecki. Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz.

[207] Szita, István and Chaslot, Guillaume and Spronck, Pieter. Monte-Carlo Tree Search in Settlers of Catan. In *Advances in Computer Games*, 2009.

[208] Tages SAS. SolidShield Software Protection. https://www.solidshield.com/software-protection-and-licensing.

[209] The NASM development team. NASM. https://www.nasm.us/.

[210] The PHP Group. PHP. http://php.net/.

[211] S. Veggalam, S. Rawat, I. Haller, and H. Bos. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security (ESORICS)*, pages 581–601, 2016.

[212] Veillard, Daniel. The XML C parser and toolkit of Gnome. http://xmlsoft.org/.

[213] VMProtect Software. VMProtect Software Protection. http://vmpsoft.com.

[214] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent Data-only Malware: Function Hooks without Code. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[215] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[216] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, 2009.

[217] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[218] X. Xie, T. Y. Chen, and B. Xu. Isolating suspiciousness from spectrum-based fault localization techniques. In *International Conference on Quality Software*, 2010.

[219] J. Xu, Z. Zhang, W. K. Chan, T. H. Tse, and S. Li. A general noise-reduction framework for fault localization of Java programs. *Information & Software Technology*, 55(5), 2013.

[220] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. CREDAL: towards locating a memory corruption vulnerability with your core dump. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[221] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security Symposium*, 2017.

[222] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[223] B. Yadegari and S. Debray. Bit-level Taint Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014.

[224] B. Yadegari and S. Debray. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[225] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.

[226] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[227] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[228] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2015.

[229] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 6 2011.

[230] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, pages 745–761, 2018.

[231] M. Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, .

[232] M. Zalewski. afl-fuzz: crash exploration mode. `https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html`, .

[233] M. Zalewski. Technical “whitepaper” for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt, .

[234] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[235] L. Zhao, Y. Duan, H. Yin, and J. Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[236] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications (WISA)*, 2007.