PRACTICAL METHODS FOR FUZZING REAL-WORLD SYSTEMS

by

Prashast Srivastava

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL STATEMENT OF COMMITTEE APPROVAL

To my family

ACKNOWLEDGMENTS

I am greatly indebted to my major advisor, Mathias Payer who gave me a chance to pursue my PhD under his tutelage. He has been a constant source of knowledge for how to conduct research and communicate its results effectively. His guidance has not just been limited to my academic pursuits. I have learned a great deal about what it means to lead people with empathy and that is a lesson that I hope to carry forward with me into the future. It was also an absolute privilege to have Antonio Bianchi as my co-advisor. His feedback was instrumental in not only refining my research directions but also figuring out how to navigate academia as well.

This dissertation also would not have come to fruition without the support of my peers and collaborators along the way. I am greatly indebted to both past and present members of HexHive for not only providing lively research discussions but also serving as a source of much needed levity. Furthermore, I would like to thank PurSec lab members for being a constant source of encouragement as well. In addition, I would also like to thank all the people who were a part of this journey during my time at Purdue. Finally, and most importantly I would like to thank my family who served as an anchor for me throughout the course of my PhD.

TABLE OF CONTENTS

LIST OF TABLES 9

LIST OF FIGURES 11

ABBREVIATIONS 12

ABSTRACT 13

1 INTRODUCTION 15

2 GRAMATRON: EFFECTIVE GRAMMAR-AWARE FUZZING 22

2.1 Introduction 22

2.2 Background 24

2.2.1 Context-Free Grammars 25

2.2.2 Automaton Classes 25

2.2.3 Grammar-Aware Mutation Operators 26

2.3 Gramatron Overview 27

2.4 Grammar Automatons 28

2.4.1 Motivating Example 28

2.4.2 Automaton Construction 30

2.4.3 Automata-Based Mutation 34

2.5 Implementation 35

2.5.1 Fuzzing Workflow 35

2.6 Evaluation 36

2.6.1 Performance Microbenchmarks 37

2.6.2 Faster Bug Discovery 43

2.6.3 Bug Discovery Performance in the Wild 47

2.7 Discussion and Future Work 49

2.8 Threats to Validity 50

2.9 Related Work 52

2.10 Conclusion 54

3 FIRMFUZZ: AUTOMATED IOT FIRMWARE
INTROSPECTION AND ANALYSIS 55

3.1 Introduction 55

3.2 Firmware Preprocessing 57

3.2.1 Information Gathering Phase 58

3.2.2 Firmware Preparation 58

Peripheral Mapping 58

Helper Injection 59

Network Configuration 60

3.3 Firmware Fuzzing 60

3.3.1 Syntactically Legal Input Generation 61

3.3.2 Deterministic Bug Discovery 62

3.3.3 Elimination of Fuzzing Side-Effects 63

3.3.4 Payload Delivery 63

3.4 Evaluation 66

3.4.1 Firmware Images Tested 66

3.4.2 Case Study 67

TRENDnet TEW-673GRU Router 67

3.4.3 Comparison with Existing Analysis Frameworks 68

Firmadyne 68

w3af 69

Zed Attack Proxy 69

3.4.4 Runtime Performance 70

3.4.5 Vulnerability Detection Accuracy 70

3.5 Discussion and Future Work 71

3.6 Related Work 71

3.7 Conclusion 73

4 SIEVEFUZZ: TARGET-TAILORED PROGRAM STATE RESTRICTION 74

4.1 Introduction 74
4.2 Background 76
4.3 Pitfalls of Distance Minimization 77
4.4 Overcoming the Bottlenecks of Directedness 80
4.5 Preemptive Termination 81
4.5.1 Tripwiring 81
4.6 Implementation: SieveFuzz 84
4.6.1 Architectural Overview 84
4.6.2 High-level Fuzzing Workflow 84
4.6.3 Maintaining Fast On-demand Analysis 85
4.6.4 Maintaining Fast SUT Execution 86
4.6.5 Maintaining Exploration Diversity 87
4.7 Evaluation 88
4.7.1 RQ1: Tripwiring's Search Space Restriction 90
4.7.2 RQ2: Targeted Defect Discovery 92
4.7.3 RQ3: Target Location Feasibility for Tripwiring 96
4.8 Discussion and Future Work 98
4.9 Related Work 99
4.10 Conclusion 100

5 CRYSTALLIZER: DESERIALIZATION VULNERABILITY DISCOVERY FRAMEWORK 101
5.1 Introduction 101
5.2 Deserialization Attacks 103
5.2.1 Serialization and Deserialization 103
5.2.2 Payload Formalization 104
5.2.3 Payload Example 105
5.2.4 Challenges 107
5.3 Crystallizer Design 108
5.3.1 Static Analysis Module 109

5.3.2 Sink Identification 109

5.3.3 Probabilistic Concretization 111

5.4 Implementation 114

5.5 Evaluation 115

5.5.1 RQ1: Library-based evaluation 116

5.5.2 RQ2: Comparison against state-of-the-art tools 119

5.5.3 RQ3: Comparative Performance Evaluation 123

5.5.4 RQ4: Sink Identification Evaluation 124

5.5.5 RQ5: Crystallizer in-the-wild 126

5.6 Discussion and Future Work 127

5.7 Related Work 128

5.8 Conclusion 129

6 SUMMARY 130

REFERENCES 132

LIST OF TABLES

2.1 Input diversity comparison using branch cov showing absolute cov% and cov increase% relative to baseline of Gramatron (G), Nautilus$_{GH}$ (N$_{GH}$), and N$_{GH}$-G: Nautilus$_{GH}$ with GNF grammar. . . . 38

2.2 % improvement in terms of space, time, and mutation scale of Gramatron against Nautilus$_{GH}$. Effect size quantifies the improvement magnitude (> 0.8 = large, > 0.5 = medium). . . . 41

2.3 Performance breakdown of time to bug discovery across 10 fuzzing campaigns. *G*: Gramatron, *(N_P, N_{GH})*: Nautilus$_P$ and Nautilus$_{GH}$. . . . 45

2.4 Fuzz targets with types of bugs discovered. *Key*: OOB: Out-of-bounds read, (H)BO: (Heap) Buffer Overflow, AV: Assertion Violation, and UAS: Stack use-after-scope. . . . 48

2.5 Median standard deviation of the time taken to find different bug triggers. . . . 49

3.1 Firmware images tested . . . 66

3.2 Vulnerability Detection Comparison of FirmFuzz against Zed Attack Proxy(ZAP), Firmadyne and w3af . . . 68

4.1 Information about our ground-truth bug benchmark corpus. *Key*: R: real-world, S: synthetic, UAF: use-after-free, FPE: floating point exception, BoF: buffer overflow, OOB: out-of-bounds and NPD: NULL pointer dereference. . . . 88

4.2 Percentage of code regions (at function level) removed by tripwiring during fuzzing, and the analysis time spent in tripwiring's pre-fuzzing initialization. . . . 90

4.3 Bug exposure effectiveness; and mean exposure times and effect sizes for SieveFuzz versus minimization-directed AFLGo and undirected AFL++ across 10×24-hour fuzzing trials per our ten ground-truth bugs. *Bold* effect sizes reflect statistically-large (i.e., Vargha and Delaney $A_{12} > 0.71$) improvements in bug exposure times; while `[n/a]` denotes that the statistical test cannot be performed due to an insufficient number of exposing trials by SieveFuzz's competitor. . . . 92

4.4 Bug exposure effectiveness; and mean exposure times and effect sizes for SieveFuzz versus precondition-directed BEACON across 10×24-hour fuzzing trials per our eight ground-truth bugs. In this experiment, we run SieveFuzz with the same fuzz target configuration as BEACON. *Bold* effect sizes reflect statistically-large (i.e., Vargha and Delaney $A_{12} > 0.71$) improvements in bug exposure times; while `[n/a]` denotes that the statistical test cannot be performed due to an insufficient number of exposing trials by BEACON. . . . 94

4.5 Comparisons of the mean test case throughputs (executions/sec) between Sieve-Fuzz (tripwiring-directed), and BEACON (precondition-directed). Values > 1.0 represent a relative *speedup* (shown in bold), while values < 1.0 represent a relative *slowdown*. 95

5.1 Evaluation Benchmarks paired with their ground truth chains. 115

5.2 Gadget graph size of the target libraries and the time taken by Crystallizer to create it. 116

5.3 Candidate chains explored by Crystallizer along with chains that were successfully concretized, chains that were deemed to be interesting, and chains that were manually validated to be exploitable. 117

5.4 Novel gadget chains found by Crystallizer along with their average gadget frequency and a comparison of the unique classes present in the discovered ground truth chain and the novel chains. 120

5.5 Comparison of Gadget Inspector against Crystallizer in terms of gadget chains reported for libraries and the ones which were confirmed to be exploitable. . . . 121

5.6 Comparison of Crystallizer against Crystallizer-NG in terms of gadget chains reported for libraries and the ones which were confirmed to be exploitable. . . . 123

5.7 "Pre-filtering" refers to the set of *sink gadgets* flagged by *Sink Identification*'s oracle. "Post-filtering" shows the number of remaining *sink gadgets* after applying the static filters. These are the sinks that Crystallizer tries to concretize paths to. "% reduction" refers to the difference between the number of pre- and post-filtered sinks. 124

LIST OF FIGURES

2.1 Gramatron overview. 27

2.2 Parse tree for the input $<?php\ rand\ (); return;\ ?>$. 31

2.3 FSA for the grammar in Listing 2.2. 31

2.4 Visualization of a representative run of microbenchmark comparison of Gramatron against Nautilus$_{\text{GH}}$. 40

3.1 FirmFuzz Workflow 57

3.2 FirmFuzz fuzzing workflow 60

3.3 FirmFuzz Mutation Strategy 65

4.1 A visualization of *tripwiring*-directed fuzzing. 80

4.2 SieveFuzz's high-level state machine. Here, *reachable* denotes that our analysis identifies some path(s) from the program entry point to the target location. . . . 85

4.3 Amount of function state space removed during tripwiring, and SieveFuzz's and AFLGo's discovery times per each real-world bug benchmark. 97

5.1 A simplified example for the *gadget chain* executed upon the payload (Listing 3) being deserialized. 105

5.2 Architectural overview of Crystallizer. 108

5.3 Time required by Crystallizer to discover the exploitable gadget chains. 118

ABBREVIATIONS

AFW	Arbitrary File Write
AV	Assertion Violation
BO	Buffer Overflow
CI	Command Injection
CG	Call Graph
DoS	Denial Of Service
FSA	Finite State Automatons
IoT	Internet of Things
NPD	Null Pointer Dereference
OOB	Out-of-bounds
RCE	Remote Code Execution
SUT	Software Under Test
UAS	Use-After-Scope

ABSTRACT

The current software ecosystem is exceptionally complex. A key defining feature of this complexity is the vast input space that software applications must process. This feature inhibits fuzzing (an effective automated testing methodology) in uncovering deep bugs (i.e., bugs with complex preconditions). We improve the bug-finding capabilities of fuzzers by reducing the input space that they have to explore. Our techniques incorporate domain knowledge from the software under test. In this dissertation, we research how to incorporate domain knowledge in different scenarios across a variety of software domains and test objectives to perform deep bug discovery.

We start by focusing on language interpreters that form the backend of our web ecosystem. Uncovering deep bugs in these interpreters requires synthesizing inputs that perform a diverse set of semantic actions. To tackle this issue, we present Gramatron, a fuzzer that employs grammar automatons to speed up bug discovery. Then, we explore firmwares belonging to the rapidly growing IoT ecosystem which generally lack thorough testing. FirmFuzz infers the appropriate runtime state required to trigger vulnerabilities in these firmwares using the domain knowledge encoded in the user-facing network applications. Additionally, we showcase how our proposed strategy to incorporate domain knowledge is beneficial under alternative testing scenarios where a developer analyzes specific code locations, e.g., for patch testing. SieveFuzz leverages knowledge of targeted code locations to prohibit exploration of code regions and correspondingly parts of the input space that are irrelevant to reaching the target location. Finally, we move beyond the realm of memory-safety vulnerabilities and present how domain knowledge can be useful in uncovering logical bugs, specifically deserialization vulnerabilities in Java-based applications with Crystallizer. Crystallizer uses a hybrid analysis methodology to first infer an over-approximate set of possible payloads through static analysis (to constrain the search space). Then, it uses dynamic analysis to instantiate concrete payloads as a proof-of-concept of a deserialization vulnerability.

Throughout these four diverse areas we thoroughly demonstrate how incorporating domain knowledge can massively improve bug finding capabilities. Our research has developed tooling that not only outperforms the existing state-of-the-art in terms of efficient bug dis-

covery (with speeds up to 117% faster), but has also uncovered 18 previously unknown bugs, with five CVEs assigned.

1. INTRODUCTION

Real-world software systems are highly complex. A way to characterize this complexity is through the vast input state space that they can process. These systems will inevitably have bugs creep in, owing to errors made by programmers [1]. These bugs cause software to reach an invalid state, and some states may be leveraged by an adversary to perform security violations [2]. These bugs, therefore, become vulnerabilities.

To prevent exploitation of vulnerabilities and to limit the impact of bugs, developers perform software testing to find and fix bugs. Fuzzing is a highly effective dynamic software testing methodology for finding bugs [3]. Fuzzing tests a program by creating or mutating inputs followed by executing the program under test with the generated input. Observing the execution of the target can inform future input generation and any inputs that cause the target to crash are set aside for later analysis. However, current approaches can have trouble finding *deep* bugs in complex systems. We define a bug as being deep if it requires a complex set of preconditions to be met to be triggered.

In this work, we focus on building practical fuzzing solutions for finding deep bugs in widely-used software systems that are public-facing. Such systems require rigorous security vetting. First, we concentrate on language interpreters that form the backbone of the internet. Second, we look into the rapidly growing Internet of Things (IoT) ecosystem, specifically Linux-based firmware running on these IoT devices. Third, we propose a targeted fuzzing solution for testing specific parts of the software to make objectives like patch testing more viable. Finally, we examine deserialization-based attack vectors focusing on enterprise applications written in Java due to their longstanding prevalence. Designing each of these solutions comes with its own set of unique challenges:

- **Diverse semantic structure**: Language interpreters, e.g. PHP, accept input as defined by a context-free grammar. Deep bugs in these can only be uncovered by synthesizing inputs that are not only syntactically correct but also perform a diverse set of semantic actions. Existing methodologies are inefficient at generating inputs with a diverse semantic structure to trigger deep bugs.

- **Appropriate runtime state**: In IoT firmware, vendor-developed applications that perform the core device functionality are riddled with deep bugs as this software has never been thoroughly tested. These bugs can only be triggered if their preconditions have been satisfied with an appropriate runtime state. Existing methodologies have failed at providing a convincing solution for inferring this state. This is because the runtime state can be setup in a myriad of ways and finding the appropriate one is non-trivial.

- **Global search in input space**: For some testing scenarios, such as patch testing we may want to test a specific code location in the software under test. Targeted fuzzing methodologies have been proposed for such instances. However, a key problem with existing solutions is that they perform a global search through the input space with each input being treated as a potential candidate to reach the targeted location which is inefficient. This inhibits the fuzzer from uncovering deep bugs in the targeted location.

- **Complex payload creation**: Security vulnerabilities that stem from performing deserialization occur due to the lack of proper input validation. Exercising deserialization-based vulnerabilities requires synthesizing a set of nested objects which divert the control flow towards sinks that perform attacker-intended functionality. As of now, the process of discovering such vulnerabilities is predominantly manual. Incorporating more automation into the process of uncovering such vulnerabilities requires: (i) devising an effective way for identifying sinks, and (ii) efficiently traversing the large state space of possible payloads.

Thesis statement and Dissertation Layout

The root cause of all these challenges is that the input space of complex, real-world software is too vast for a fuzzer to explore efficiently to find deep bugs. To address this key issue, my thesis is:

Fuzzing can find more bugs by meaningfully reducing the input space to be explored through sampling inputs from a distribution that incorporates domain knowledge about the software under test.

As evidence to support my thesis, I will present in subsequent chapters four pieces of work. Across these works, varying sources of domain knowledge are employed as detailed below to reduce the input space based on software functionality or the test objective. All of these works have been peer-reviewed and published, except Crystallizer which is, at the time of writing, under review.

- Gramatron [4], a grammar-aware, coverage-guided fuzzer that utilizes the context-free grammar, explicit grammar accepted by the software under test, to create grammar automatons. These allow for unbiased sampling from the input space creating a diverse set of semantic constructs which in turn allows for finding deep bugs reliably and faster.
- FirmFuzz [5], a device-independent emulation and dynamic analysis framework for Linux-based IoT firmware. It performs firmware emulation and then systematically interacts with user-facing network applications to leverage its implicitly-specified grammar for bug discovery.
- SieveFuzz [6] employs tripwiring—a directed fuzzing methodology that uses targeted program state as a guide to statically remove parts of the input space that are irrelevant to reaching a target location.
- Crystallizer, a hybrid testing framework to automatically identify deserialization vulnerabilities in Java-based applications. Crystallizer uses the control-flow information of the software coupled with information about language semantics to create an implicit grammar. It then uses this implicitly-specified grammar to generate inputs that uncover deserialization vulnerabilities.

Gramatron

Language interpreters accept input as defined by a context-free grammar. Fuzzers that aware of the input grammar can explore deeper program states through *grammar-aware* mutations. For example, awareness of the PHP grammar allows the fuzzer to generate syntactically valid inputs that test the interpreter's functionality and not just the initial parsing. Grammar-aware fuzzers employ the context-free grammar in conjunction with the parse trees for input generation. However, the current methodology has a twofold problem: (i) Fuzzers when using existing grammars perform biased sampling from the input space, and (ii) Existing mutation operators that operate on parse trees perform small-scale mutations. These problems together contribute to prohibiting existing grammar-aware fuzzers from reliably generating complex bug triggers.

Gramatron uses grammar automatons in conjunction with aggressive mutation operators to synthesize complex bug triggers faster. We build grammar automatons to address the sampling bias. It restructures the grammar to allow for unbiased sampling from the input state space. We redesign grammar-aware mutation operators to be more aggressive, i.e., perform large-scale changes. Gramatron can consistently generate complex bug triggers efficiently as compared to using conventional grammars with parse trees. Inputs generated from scratch by Gramatron have higher diversity as they achieve up to 24.2% more coverage relative to existing fuzzers. Gramatron makes input generation 98% faster and the input representations are 24% smaller. Our redesigned mutation operators are 6.4× more aggressive while still being 68% faster at performing these mutations. We evaluate Gramatron across three interpreters with 10 known bugs consisting of three complex bug triggers and seven simple bug triggers against two Nautilus variants. Gramatron finds all the complex bug triggers reliably and faster. For the simple bug triggers, Gramatron outperforms Nautilus four out of seven times. To demonstrate Gramatron's effectiveness in the wild, we deployed Gramatron on three popular interpreters for a 10-day fuzzing campaign where it discovered 10 new vulnerabilities. This work was published at ISSTA'21 [4]. Furthermore, Gramatron has been incorporated into mainline AFL++ [7] and LibAFL [8], two leading coverage-guided fuzzers, as a dedicated testing mode by their respective developers.

FirmFuzz

With the number of IoT devices growing at an exhilarating pace, their security remains stagnant. Imposing secure coding standards across all vendors is infeasible. Testing individual devices allows an analyst to evaluate their security post-deployment. Any discovered vulnerabilities can then be disclosed to the vendors to assist them in securing their products. The search for vulnerabilities should ideally be automated for efficiency and device-independent for scalability.

We present *FirmFuzz*, an automated device-independent emulation and dynamic analysis framework for Linux-based firmware images. It employs a greybox-based generational fuzzing approach coupled with static analysis and system introspection to provide targeted and deterministic bug discovery within a firmware image. We evaluate FirmFuzz by emulating and dynamically analyzing 32 images (from 27 unique devices) with a network accessible from the host performing the emulation. During testing, FirmFuzz discovered seven previously undisclosed vulnerabilities across six different devices: two IP cameras and four routers. So far, four CVE's have been assigned. This work was published at IoTSP'19 [5].

SieveFuzz

Fuzzing is the de-facto default technique to discover software flaws, randomly testing programs to discover crashing test cases. Yet, a particular scenario may only care about specific code regions (for, e.g., bug reproduction, patch or regression testing)—spurring the adoption of *directed fuzzing*. Given a set of pre-determined target locations, directed fuzzers drive exploration toward them through *distance minimization* strategies that (1) isolate the closest-reaching test cases and (2) mutate them stochastically. However, these strategies are applied onto *every* explored test case—irrespective of whether they ever reach the targets—stalling progress on the paths where targets are *unreachable*. Accelerating directed fuzzing requires prioritizing *target-reachable* paths.

To overcome the bottleneck of wasteful exploration in directed fuzzing, we introduce *tripwiring*: a lightweight technique to preempt and terminate the fuzzing of paths that will *never* reach target locations. By constraining exploration to only the set of target-reachable program paths, tripwiring curtails directed fuzzers' search noise—while unshackling them from the high-overhead instrumentation and bookkeeping of distance minimization—enabling directed fuzzers to obtain up to 99× higher test case throughput. We implement tripwiring-directed fuzzing as a prototype, SieveFuzz, and evaluate it alongside the state-of-the-art directed fuzzers AFLGo, BEACON, and the leading undirected fuzzer AFL++. Overall, across nine benchmarks, SieveFuzz's tripwiring enables it to trigger bugs on an average 47% more consistently and 117% faster than AFLGo, BEACON and AFL++. This work was published at ACSAC'22 [6].

Crystallizer

Applications leverage serialization and deserialization to exchange data between instances. Serialization allows developers to exchange messages or perform remote method invocation in distributed applications. However, the application logic itself is responsible for security. Adversaries may abuse bugs in the deserialization logic to forcibly invoke attacker-controlled methods by crafting malicious bytestreams (payloads).

Crystallizer presents a novel hybrid framework to automatically identify deserialization vulnerabilities by combining static and dynamic analyses. Our intuition is to first over-approximate possible payloads through static analysis (to constrain the search space). Then, we use dynamic analysis to instantiate concrete payloads as a proof-of-concept of a vulnerability (giving the analyst concrete examples of possible attacks). Our proof-of-concept focuses on Java deserialization as the imminent domain of such attacks. We evaluate our prototype on seven popular Java libraries against state-of-the-art frameworks for uncovering gadget chains. In contrast to existing tools, we uncovered 47 previously unknown exploitable chains. Finally, we show the real-world security impact of Crystallizer by using it to synthesize gadget chains to mount RCE and DoS attacks on two popular Java applications automatically. We have responsibly disclosed all newly discovered vulnerabilities. This work is currently under review.

Permissions and Attributions

While all the work presented in this dissertation was lead by me as the primary author, it did greatly benefit from collaborations with the following people: (i) The content of Chapter 2 is the result of a collaboration with Mathias Payer, and part of a previously published paper [4], (ii) The content of Chapter 3 is the result of a collaboration with Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer, and part of a previously published paper [5], (iii) The content of Chapter 4 is the result of a collaboration with Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer, and part of a previously published paper [6], and (iv) The content of Chapter 5 is the result of a collaboration with Flavio Toffalini, Kostyantyn Vorobyov, Francois Gauthier, Antonio Bianchi, and Mathias Payer, and part of a paper that is under review as of the time of this writing.

Outlook

The subsequent chapters in the dissertation are organized as follows. Chapter 2 presents grammar automatons and Gramatron, a tool that uses these automatons to perform effective grammar-aware fuzzing. The design behind our IoT firmware analysis framework (FirmFuzz) is detailed in Chapter 3. In Chapter 4, we showcase SieveFuzz, our tool that employs tripwiring: a lightweight approach to perform directed fuzzing. Chapter 5 describes Crystallizer, our hybrid approach to automatically identify deserialization vulnerabilities in Java-based applications. In each of these chapters, we also provide the necessary background, experimental evaluation results, and discussion about relevant related and future work. Finally, Chapter 6 concludes our dissertation.

2. GRAMATRON: EFFECTIVE GRAMMAR-AWARE FUZZING

2.1 Introduction

Language interpreters like PHP, JavaScript (JS), or Ruby accept input whose structure is defined as per a grammar. These interpreters form the building blocks for complex application frameworks but are themselves highly vulnerable to exploitation with 98 reported bugs between January 2018 and January 2021 [9–11]. Hence, they are lucrative targets for adversaries. Testing these building blocks, i.e., the interpreters, is essential to ensure the safety of software running on top of them such as web applications.

Fuzzing is an effective software security testing methodology. However, current fuzzing approaches are ineffective at performing *deep* testing of interpreters. A majority of the test inputs generated by grammar-unaware fuzzers [12, 13] are rejected by the interpreter during parsing. Interpreters reject all inputs that violate the grammar, and when fuzzers are unaware of the accepted grammar, they will mostly create syntactically incorrect input. For example, a common mutation operator is flipping random input bits. A fuzzer unaware of the grammar may flip bits in input keywords, creating invalid mutants that are rejected by the parser. The interpreter components past the parsing stage corresponding to semantic analysis remain untested if fuzzers are grammar-unaware.

Fuzzing the semantic analysis components requires generating syntactically valid inputs. Existing grammar-aware fuzzers [14–17] use: (i) a context-free grammar (CFG) to generate test inputs, and (ii) parse trees to represent the syntactic structure of the input. The fuzzer mutates the parse trees using the grammar to generate syntactically valid inputs for testing—grammar-aware mutations. We observe a twofold problem with the current methodology for grammar-aware fuzzing:

- **Biased sampling:** Fuzzers when using existing grammars perform biased sampling from the input state space. This bias occurs due to how the production rules in the CFG are laid out for generating inputs. This bias can make it harder for the fuzzer to generate complex bug triggers which require chaining multiple parts of the input state space.

- **Small-scale mutations:** Grammar-aware fuzzers employ parse trees for input generation and mutation. Existing mutation operators for parse trees perform localized small-scale changes. This slows down the fuzzer while trying to discover bugs with complex bug triggers if it wastes its time fuzzing grammar parts not relevant towards triggering the bug.

We propose a two-fold solution to the above problem: (i) Restructuring the production rules in the grammar to eliminate the sampling bias, and (ii) Redesigning mutation operators to perform larger-scale changes. However, implementing these solutions on top of inputs represented as parse trees imposes a performance overhead. This overhead arises from a fuzzer having to maintain metadata in the form of the input derivation structure for each input as its being generated or mutated. To remove this overhead and implement our solution in a performance-optimal way, we convert the input grammar to a finite state automaton (FSA), which we refer to as *grammar automatons*.

Grammar automatons restructure the grammar to eliminate the sampling bias. Furthermore, grammar automatons allow performing *aggressive* mutations (more terminals may be changed) efficiently. Aggressive mutations ensure that the fuzzer does not get stuck performing localized search of grammar parts that are not relevant towards triggering bugs. We present Gramatron, a proof-of-concept for our claim that grammar automatons are an effective solution for performing grammar-aware fuzzing. It represents inputs as automaton walks and uses grammar-aware mutation operators that have been redesigned for fast and aggressive mutations.

We evaluated using performance microbenchmarks if: (i) Gramatron resolves the sampling bias in a performant manner, and (ii) redesigned mutation operators perform aggressive changes efficiently. Gramatron generates higher diversity inputs as they achieve up to 24.2% more coverage. This indicates that our generated inputs cover richer semantics of the input grammar. Additionally, on average, inputs represented as automaton walks are 24% smaller and input generation is 98% faster. Our redesigned mutation operators are on an average 68% faster and 6.4× more aggressive in their mutations.

To showcase that using conventional grammars coupled with small-scale mutations can be a problem while trying to discover complex bug triggers, we compared Gramatron against two variants (detailed in § 2.6) of the current state-of-the-art grammar-aware fuzzer Nautilus [18]. We evaluate all systems against a set of three interpreters with 10 known bugs consisting of three complex bug triggers and seven simple bug triggers. For the three complex bug triggers, Gramatron outperforms one variant on all of them and the other variant on two of them. Gramatron finds four out of the seven simple bug trigger faster than one variant and three out of the seven than the other variant. Furthermore, to prove its effectiveness in the wild, we deployed Gramatron on three popular interpreters for a 10-day fuzzing campaign. It discovered 10 new vulnerabilities. Additionally, we discuss a bug found as a case-study to show how Gramatron is effective at generating complex bug triggers.

The main contributions of Gramatron are:

- We leverage grammar automatons which restructures grammars to allow generating highly diverse inputs reliably as an effective methodology to synthesize complex bug triggers
- We redesign and optimize mutation operators for grammar automatons to enable them to do aggressive mutations for faster discovery of bugs with complex triggers.
- As a proof-of-concept, we build and evaluate Gramatron, an efficient grammar-aware fuzzer. Source code of our framework is available at https://github.com/HexHive/Gramatron.
- Gramatron discovered 10 new vulnerabilities over a 10-day fuzzing campaign, so far one CVE has been assigned.

2.2 Background

Gramatron transforms grammars to an automaton to resolve the sampling bias and enable performing aggressive mutations efficiently. This section introduces the necessary background for automaton generation. We also introduced the mutation operators which are customized for this new representation.

2.2.1 Context-Free Grammars

Language interpreters define the input format accepted by them using a Context-Free Grammar (CFG). Formally, a Context-free Grammar (CFG) [19] is defined as: $CFG = (T, N, R, S)$. T is a finite set of *terminals* (characters in the generated string). N is a finite set of nonterminal symbols (placeholders for patterns of T that can be generated by N). R is a finite set of rules for substituting N with T. The rules are of the form $A \rightarrow a$ where A is *always* a nonterminal symbol. However, a can be a permutation of symbols from both N and T. S is the starting nonterminal symbol from which all strings belonging to the grammar are derived.

In order to eliminate sampling bias, Gramatron restructures the grammar using the normal form by enforcing certain rules. We focus on two normal forms, used together by Gramatron to create grammar automatons from a CFG: *Chomsky Normal Form* (CNF) [20] and Greibach Normal Form (GNF) [21].

Both CNF and GNF are similar to CFG except they have constraints placed on their rules (R). For CNF, each nonterminal can either generate a single terminal or two nonterminals. For GNF, each nonterminal can either generate a terminal or a terminal followed by any number of nonterminals.

2.2.2 Automaton Classes

We give formal definitions of different automaton classes that Gramatron employs to create grammar automatons. Additionally, we detail the necessary background for the theoretical challenges involved in creating grammar automatons.

Finite State Automatons

Gramatron creates grammar automatons for performing fuzzing using finite state automatons (FSA). This allows Gramatron to create lightweight input representations that enable fast and aggressive mutations. Formally, a FSA is defined as $M = (Q, \Sigma, \delta, q_0, F)$ [22]. Here, Q is a finite set of states. Σ is the finite set of terminals. δ defines the set of all transitions over the automaton states. q_0 is the start state in the automaton corresponding to the start symbol of the grammar. F is the accepting state in the automaton. A set of transitions from the start state to the final state describe an input string that belongs to the grammar.

Pushdown Automatons

Gramatron leverages pushdown automatons (PDA) to create grammar automatons. A PDA is a language recognizer for a CFG. In the context of Gramatron, we will consider the 1-state PDA that accepts inputs by an empty stack. Formally, a PDA is defined as $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ [23]. Here, Q is a finite set of states. Σ is the finite set of terminals. Γ is a set of symbols that can be pushed and popped from the stack (referred to as *stack symbols*). Z is the start symbol pushed to the stack. q_0 is the start state and F is the accepting automaton state. δ is the transition function that governs the automaton behavior. It takes as argument $\delta(q, a, X)$ where, $q \in Q$, $a \in \Sigma$, and $X \in \Gamma$. The output of δ is a finite set of pairs (p, γ) where p is a new state and γ is the string of symbols that replace X at the top of the stack by being pushed in reverse. If γ is empty then only X is popped from the stack.

2.2.3 Grammar-Aware Mutation Operators

Grammar-aware mutation operators mutate inputs while maintaining syntactic validity. They take in as input a grammar along with a syntactically valid test case to create mutants. There are three mutation operators which have proven successful in finding deep bugs [18]: *random mutation* (pick a random non-leaf nonterminal node and create a new subtree), *random recursive* (find recursive production rules and unroll up to n times), and *splice* (combine two inputs while preserving syntax). We implemented them in Gramatron and tailored them for fast and aggressive mutant generation.

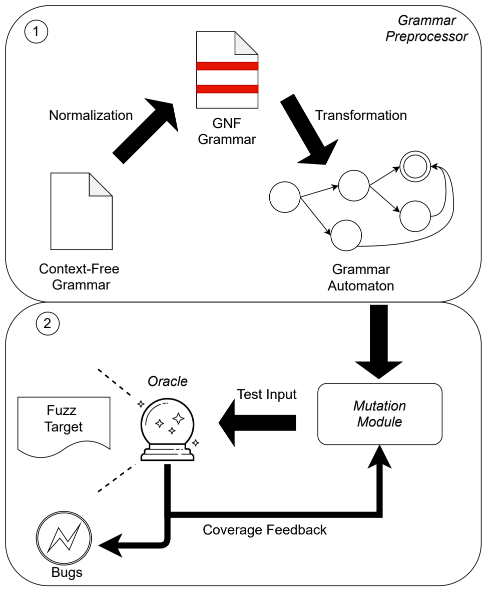


Figure 2.1. Gramatron overview.

2.3 Gramatron Overview

Gramatron is a coverage-guided, grammar-aware generational fuzzer. As input, Gramatron takes a CFG accepted by the target. Gramatron outputs crashing test cases . Figure 2.1 highlights Gramatron's two distinct phases: ① preprocessing phase, followed by a ② fuzzing phase. In the preprocessing phase, Gramatron transforms a grammar into its corresponding grammar automaton. It first converts the grammar into a form for unbiased input sampling from which it creates the grammar automaton. It is an FSA that encodes the input space represented by the grammar.

In the fuzzing phase, Gramatron uses the automaton and mutation operators redesigned for fast and aggressive mutations. This enables Gramatron to rapidly find bugs with complex triggers. Furthermore, to guide the fuzzing towards interesting parts of the grammar, it uses coverage feedback to guide its mutation.

2.4 Grammar Automatons

A fuzzer must generate syntactically valid inputs to fuzz *beyond* the application parser. Previous fuzzers employ parse trees with CFG for input generation and mutation. A parse tree encodes the replacement rules applied to generate the input from the grammar. The fuzzer uses this information to perform mutations. Existing grammar-aware fuzzers are ineffective at synthesizing complex bug triggers due to biased sampling and localized small-scale mutations.

Conventional grammar structure introduces bias during input generation when a fuzzer is choosing replacement rules at random to expand. Grammar rules are laid out in a way that makes it probabilistically harder to generate certain parts of the input state space. This bias in turn causes the fuzzer to generate certain patterns less effectively, slowing down bug discovery. Existing fuzzers use mutation operators designed for parse trees which perform localized small-scale changes. These small-scale mutations are detrimental if the fuzzer gets stuck fuzzing parts of the grammar that are not relevant towards triggering a bug.

Gramatron performs an automatic two-step transformation on the CFG to create a grammar automaton. First, it transforms the CFG into its GNF which performs the grammar restructuring. Second, it converts the GNF of the grammar into an automaton. Gramatron can encode any input that abides by the grammar as an automaton walk. Grammar automatons restructure the grammar enabling the fuzzer to perform unbiased sampling from the input state space. This enable the fuzzer to generate inputs with higher diversity more frequently. We also redesign the mutation operators to operate on grammar automatons and perform aggressive changes efficiently to discover bugs with complex triggers.

2.4.1 Motivating Example

The following concise example gives an intuition as to how grammar automatons perform unbiased sampling from the input distribution and allow for aggressive mutations to be performed efficiently. Listing 2.1 presents a subset of the PHP grammar [24].

```
program    -> '<?php' phpBlock '?>'
phpBlock   -> stmt                          |
              stmt phpBlock
stmt       -> callStmt                      |
              retStmt
callStmt   -> func '();'
retStmt    -> 'return ;'
func       -> 'rand'                        |
              'mt_rand'
```

Listing 2.1 Subset of PHP grammar.

```
program    -> '<?php' phpBlock C
phpBlock   -> 'rand' A                      |
              'mt_rand' A                   |
              'rand' A phpBlock             |
              'mt_rand' A phpBlock          |
              'return;' phpBlock            |
              'return;'
A          -> '();'
C          -> '?>'
```

Listing 2.2 The same PHP grammar in GNF.

Biased sampling

The CFG in Listing 2.1 shows how a conventional structure forces the fuzzer to perform biased sampling from the input distribution. A fuzzer generating an input from this CFG will start from the symbol (`program`) and iteratively keep applying the rules until the resulting string has no nonterminals. `callStmt` that can generate twice as many distinct subtrees as the `retStmt` has the same 50% probability of being picked for generation. Therefore, the fuzzer undersamples the subtrees corresponding to `callStmt`. This leads to low diversity in the inputs generated which in turn leads to lesser test coverage during fuzzing.

The grammar automaton as shown in Figure 2.3 describes the GNF of the CFG (Listing 2.2). GNF allows Gramatron to perform unbiased sampling from the input state space. It restructures the grammar to explicitly enumerate all distinct subtrees that can be generated by each nonterminal. Thus, we see that subtrees corresponding to the function invocation become twice as likely to be picked for generation because they can generate twice as many subtrees.

Aggressive mutations

Gramatron uses aggressive mutation operators. Specifically, given an input string and a point of mutation, Gramatron mutates until the end of the string relative to the mutation point. Automatons have a design that is more conducive for performing such changes as compared to parse trees.

To understand why automatons are more optimal, lets assume we have a sample program `<?php rand(); return; ?>`. Figure 2.2 shows the parse tree for this program. From Figure 2.3, Gramatron would represent this input as a sequence of automaton transitions. This would be $[0\_1, 1\_4, 4\_1, 1\_3, 3\_1]$. Lets assume we chose the `return;` statement as the mutation point.

If we wanted to make an aggressive change relative to the nonterminal node `stmt`, we would need to maintain a parse stack while performing the mutation. The parse stack keeps track of the unexpanded nonterminal nodes as the input is being generated. The overhead of maintaining this stack has a worst case complexity of $O(n)$ where n is the number of parse tree nodes.

This overhead of maintaining a parse stack can be completely avoided by performing this mutation on an automaton-based representation. This is because the parse stacks are implicitly encoded in the automaton states. Therefore, just by diverging the walk from a state, we can perform an aggressive mutation.

2.4.2 Automaton Construction

We first describe the automaton construction algorithm used by Gramatron. Then we will go over the challenge faced while applying this algorithm and the insight we used to solve that challenge.

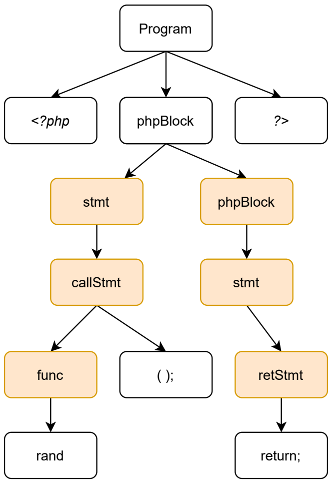


Figure 2.2. Parse tree for the input $<?php\ rand\ (); return;\ ?>$.

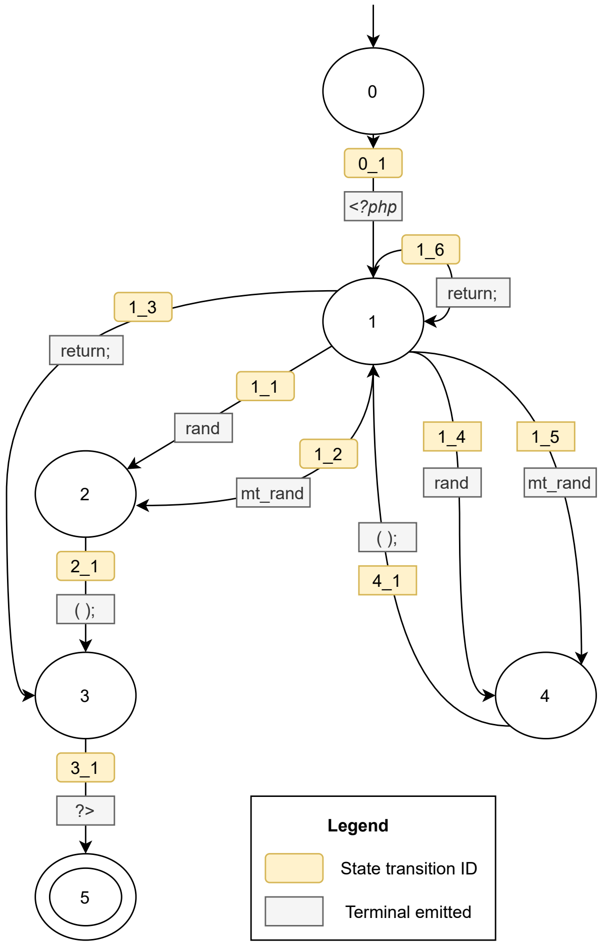


Figure 2.3. FSA for the grammar in Listing 2.2.

Construction Algorithm

Gramatron performs a two-step procedure to transform a grammar into its corresponding automaton is: (i) transforming the grammar to its GNF and (ii) converting the transformed grammar into an automaton. First, Gramatron converts a grammar G to its CNF [20] and then performs fixed point iterations over its CNF to convert it into its GNF. Gramatron performs its grammar construction by first specifying the transition function of the PDA for each CFG production rule. For a grammar in GNF, the transition function is:

$$\delta(q, t, A) = \{(q, W) \mid A \to \mathtt{t}W \in R\}. \tag{2.1}$$

Here, t is a terminal, A is a nonterminal and W corresponds to a nonterminal set. Gramatron uses this transition function to build the grammar automaton. It does so by enumerating (if possible) all valid PDA stack states belonging to the CFG. The final state in the grammar automaton corresponds to an empty stack. For each stack state, there exists a state in the grammar automaton.

Gramatron uses a worklist-based algorithm to build the automaton. It initializes the worklist with a tuple consisting of the initial automaton state and its parse stack with the start symbol of G. It iterates over the worklist until it is empty. For each iteration, it: (i) pops an element from the worklist, (ii) from the parse stack (P) of the element, pops the topmost stack symbol (S) to create a new P'. For the stack symbol, finds all possible transitions as per the transition function, (iii) for each transition, computes the new stack P'' from P' by pushing the stack symbols (if any) in reverse, (iv-a) if P'' is equivalent to a parse stack for a previously generated automaton state, then creates a transition from current state using the terminal t to that state, and (iv-b) if P'' is a new stack state, then creates a transition from current state using terminal t to a new automaton state with stack P'' and add the new automaton state along with its parse stack to the worklist. Performing an automaton walk on the FSA creates a new seed input.

Construction Challenge/Insight

It is theoretically impossible to create an algorithm that converts any arbitrary CFG into a FSA. The impossibility arises out of a specific grammar type called self-embedding grammars with infinite automaton states. A CFG is self-embedding [25] if it contains a production rule of the form:

$$\omega \overset{*}{\Rightarrow} u\omega v \mid \{u, v\} \in T^{+} \wedge \omega \in N \tag{2.2}$$

Here, T, and N follow the standard notation as specified in § 2.2.1.

However, a key insight we had is that grammar-aware fuzzers impose an upper-bound on the generated input size. This ensures that they do not generate arbitrarily large inputs. Hence, they instantiate a subset (under-approximation) of the *language* specified by the CFG. Here, language refers to the (possibly) infinite state space of inputs that can be generated from a CFG. Gramatron leverages this insight to address the theoretical impossibility while creating grammar automatons. It approximates the CFG with a regular language [26–28]. This regular approximation is then transformed into grammar automata.

In order to perform this regular approximation, Gramatron limits the size of the parse stack (denoted by P in the algorithm) to an upper bound while generating the automaton [29, 30]. Hence, the construction algorithm terminates and an automaton is constructed. The trade-off incurred is that the generated automaton can express only a subset of the language specified by the self-embedding grammar. In the context of programming language grammars, it implies that constructs from self-embedding rules can only be nested up to a static depth. This depth is directly proportional to the allowed stack size as specified by the user and can be tuned accordingly.

However, this tradeoff is not detrimental in the context of fuzzing. Grammar-aware fuzzers already limit the input size to prevent generating arbitrarily large inputs. Hence, grammar automatons allow Gramatron to be as expressive as existing grammar-aware fuzzers while being more performant.

An exception to the above discussion of generating an under-approximation of the language are non-self-embedding grammars. They do not contain any self-embedding rules. Such grammars have a finite number of possible states and transitions [31]. Therefore Gramatron can generate a grammar automaton that can generate the *exact* language as specified by the non-self-embedding CFG.

2.4.3 Automata-Based Mutation

In Gramatron, the mutation operators (splicing, random mutation, and random recursive) operate on grammar automaton walks. To address the risk of getting stuck in a local subtree, we enable the splice and random mutation operator to perform more aggressive changes. Given an input string and a target mutation point in it, Gramatron mutates it until the end of the string. For each mutation operator, let an input I be mutated. Its corresponding representation in the form of a walk is $W = [T_1, ..T_N]$, consisting of N transitions to go from the start state of the FSA to its final accepting state. Let the visited automaton states be $S = [S_1, ..S_{N+1}]$.

Splice

Let there be two inputs represented as automaton walks, W_1 and W_2. A random transition from W_1 is chosen as the point to splice it with W_2 — T_C where $1 \leq C \leq N$. The subwalk originating from this point is replaced with a *fitting* subwalk from W_2, one that originates from the same state as T_C, which is S_C. Automatons outperform parse trees for splicing, as parse trees require heavyweight restructuring of parse tree nodes. The operator is not only changing the subtree rooted under the chosen splice point but also everything to the right of the subtree as well. For automatons, the same mutation simply requires concatenating two lists.

Random Mutation

Gramatron undertakes a three-step procedure to perform this mutation. First, it chooses a random transition in the walk W, T_C to diverge the walk. Second, it generates the unmutated part of the input verbatim using the $C-1$ transitions. Third, from the divergent state, Gramatron performs a random walk over the automaton until it reaches the final state to generate the mutant. This operator becomes faster at generating inputs represented as automaton walks. This is because it requires generating a new substring for the mutant from the provided grammar. Since grammar automatons make input generation faster, substring generation for the new mutant is also faster.

Random Recursion

Without preprocessing, finding recursions in a parse tree has the runtime complexity of $O(n \log n)$ where n is the number of tree nodes. This is because for each node you have to traverse its parents recursively to find *all* recursive features. Gramatron limits the runtime complexity for finding recursive features to $O(m)$ where m is the number of terminals in the input and $m << n$. Grammar automatons enable Gramatron to traverse W *only* once to log all recursive features. It then replicates the subwalk corresponding to a randomly picked recursive feature upto n times. In the current implementation $n = 5$.

2.5 Implementation

Gramatron is implemented in C and Python: the (ahead of time) grammar preprocessor is implemented in Python and the (performance critical) input generator and mutator are implemented in C. It takes as input a grammar accepted by the fuzz target. Gramatron modifies AFL++ [32] to leverage grammar automatons to perform input generation and mutation while fuzzing. Furthermore, it uses code coverage feedback from the fuzz target to guide its mutation.

2.5.1 Fuzzing Workflow

Gramatron is a coverage-guided grammar-aware fuzzer. There are three main stages to performing coverage-guided fuzzing [33, 34]: (i) *seed scheduling*, picking a seed from a set of seeds for generating mutants, (ii) *seed mutation*, generating mutants from the seed, and (iii) *seed selection*, selecting seeds as interesting candidates for further fuzzing based on feedback. Gramatron extends the seed mutation of AFL++ making it grammar-aware to generate syntactically valid inputs. To prevent mutants from growing arbitrarily large, Gramatron deprioritizes (but does not prohibit) inputs that have reached a size greater than 2048 bytes.

Gramatron proceeds in two stages: corpus generation and the fuzzing stage. During *corpus generation*, Gramatron generates a predefined number of syntactically valid seed inputs by performing random walks over the grammar automatons. In the current implementation, Gramatron generates 100 seed inputs. Using this seed corpus, Gramatron transitions to the fuzzing stage.

A fuzz iteration consists of four steps: (i) choose a seed from the queue, (ii) pass the seed through each of the mutation operators, and (iii) test generated mutants on the fuzz target, (iv) select candidates for further testing based on the coverage feedback. Furthermore, to prohibit the fuzzer from getting stuck in a local coverage minimum, it also generates a candidate through random walk over the grammar automaton for each fuzz iteration.

2.6 Evaluation

Our evaluation answers the following research questions:

- **RQ1**: Do grammar automatons perform unbiased sampling from the input state space?
- **RQ2**: Does Gramatron perform aggressive mutations efficiently?
- **RQ3**: Can Gramatron discover complex bug triggers faster?
- **RQ4**: Can grammar automatons find new vulnerabilities?

We compared Gramatron against two Nautilus variants, the state-of-the-art grammar aware fuzzer that uses parse trees: $Nautilus_P$: the prototype presented in the paper [18] with AFL-style mutation operators that may generate syntactically invalid inputs [35], and $Nautilus_{GH}$: the performance-optimized version released recently that removed AFL-style mutation operators [36].

For **RQ1**–**RQ2**, we performed the following experiments by comparing Gramatron against Nautilus$_{\text{GH}}$, since it is performance-optimized: (i) measure complexity and diversity of inputs generated from scratch using grammar automatons against conventional grammars (**RQ1**), (ii) performance evaluation of input generation, mutation, and mutation aggressiveness (**RQ2**). To answer **RQ3**, we evaluated the elapsed time to discover known bugs of varying complexity in three fuzz targets against both Nautilus$_{\text{P}}$ and Nautilus$_{\text{GH}}$. Finally, for **RQ4**, we deployed Gramatron for 10 days on three popular and well-fuzzed interpreters to find new vulnerabilities.

We performed the evaluation on an Intel Xeon Bronze 3106 1.7GHz processor with 45GB RAM running Debian 9.3. For a fair comparison, we ran all tools in single-threaded mode on a single core. Gramatron was compiled with Clang-8.0 at *Ofast* optimization. Nautilus$_{\text{P}}$ and Nautilus$_{\text{GH}}$ are implemented in Rust and use its nightly branch. We used the optimized build (i.e., the *release* build [37]) for our evaluation.

2.6.1 Performance Microbenchmarks

We compare inputs generated using grammar automatons against conventional grammars using parse trees along three axes: (i) complexity and diversity of the inputs generated (**RQ1**), (ii) length of mutated substrings (**RQ2**), and (iii) time taken to generate and mutate these representations (**RQ2**).

However, fuzzing consists of auxiliary stages apart from input generation and mutation. While testing Nautilus$_{\text{GH}}$, we observed these stages intertwined with each other. This introduced noise in the performance measurements. To avoid this noise during performance measurement, we created seven microbenchmarks each taking in as input a CFG to answer **RQ1**–**RQ2**. For a fair comparison, we crafted our evaluation grammars from the same grammars used by the Nautilus authors in their evaluation. Table 2.1 lists the number of rules in each grammar. The average time taken to build grammar automatons per grammar was 2.09s, i.e., the automaton construction itself is lightweight.

We also performed statistical tests for each run to: (i) quantify the performance gain and (ii) check whether the improvement is statistically significant. For magnitude quantification, we performed *Cohen's-D Effect Size* [38] and for significance testing we used the *Mann-Whitney U-Test* [33]. As per the Mann-Whitney U-test , a result is significant if p-value $<$ 0.05. All results reported for the microbenchmarks are statistically significant.

Table 2.1. Input diversity comparison using branch cov showing absolute cov% and cov increase% relative to baseline of Gramatron (G), Nautilus$_{\mathrm{GH}}$ (N$_{\mathrm{GH}}$), and N$_{\mathrm{GH}}$-G: Nautilus$_{\mathrm{GH}}$ with GNF grammar.

Target	Rules	Branch Coverage % (Absolute/Relative Increase)			
		Baseline	N$_{\mathrm{GH}}$	N$_{\mathrm{GH}}$-G	G
mruby	1185	17/0	17.3/+1.8	17.4/+2.4	18.5/+8.8
PHP	8685	2.9/0	3.0/+3.4	3.2/+10.3	3.7/+27.6
JS	171	8.4/0	8.5/+1.2	8.6/+2.4	8.8/+4.8

RQ1: Unbiased Sampling.

In this experiment, we validated whether generating inputs using grammar automatons enable unbiased sampling from the input state space. Unbiased sampling creates inputs with higher diversity. Hence, we use *input diversity* as a proxy to validate if Gramatron performs unbiased sampling. We quantify input diversity using branch coverage. The intuition is that input diversity is directly proportional to the branch coverage obtained. To account for the inherent randomness incorporated by fuzzers during input generation, we generated a large (10^5) number of inputs each over five different trials.

We performed the input diversity comparison against Nautilus$_{\mathrm{GH}}$ and a baseline input generator. Both the baseline and Nautilus$_{\mathrm{GH}}$ employ parse trees along with conventional grammars. However, a key difference between them is the strategy employed while picking which grammar rules to exercise during input generation. The baseline generator picks rules at random while Nautilus$_{\mathrm{GH}}$ tries to bias its generation towards larger inputs through probabilistic weighting of the grammar production rules. Additionally, since Gramatron restructures the grammar by leveraging its GNF to perform unbiased sampling from the input state space, we wanted to see if Nautilus$_{\mathrm{GH}}$ could become as performant as Gramatron if we just supplied the GNF of the grammar to it (Nautilus$_{\mathrm{GH}}$-G).

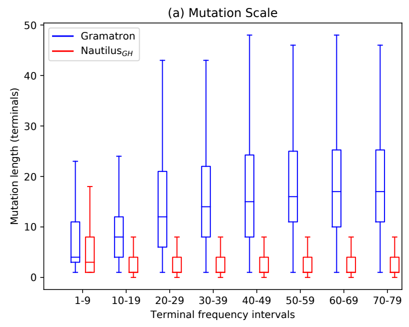
As evident from the branch coverage results in Table 2.1, Gramatron outperformed all other approaches across all grammars with respect to generating higher diversity inputs. It obtained up to 24.2% more coverage compared to the other approaches. This is because grammar automatons enable unbiased sampling from the input state space, increasing test coverage.

The baseline coverage and observed improvement are low for two reasons. First, Nautilus authors manually designed the grammars to focus on specific target functionality which is common while performing grammar-aware fuzzing. Second, we generated inputs from scratch using the grammar without leveraging coverage feedback or mutation operators. We did this to perform an evaluation of the performance improvement solely from the grammar restructuring as performed by Gramatron.

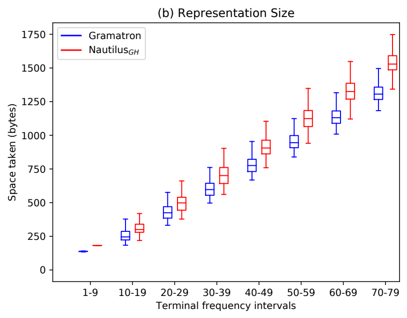
Another interesting observation is that Nautilus$_{\mathrm{GH}}$-G variant outperforms Nautilus$_{\mathrm{GH}}$ (which uses conventional grammars) but does not perform on par with Gramatron. This happens because when Nautilus$_{\mathrm{GH}}$ biases its generation it ends up biasing towards specific kind of syntactical constructs (e.g., single function invocations with a large number of arguments). These inputs, while still interesting are not helpful towards expanding the test coverage of the fuzzer. Hence, this experiment showed that Gramatron, through grammar automatons, removes sampling bias most optimally.

RQ2: Aggressive mutations

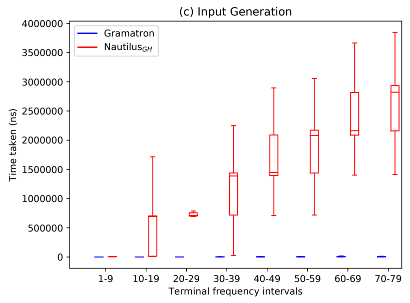
In this experiment, we validated if the mutation operators adopted by Gramatron are indeed aggressive (i.e., perform large-scale changes). We did so by comparing the mutation operators of Gramatron against those of Nautilus$_{\mathrm{GH}}$ (that perform small-scale changes). We generated inputs over 8 length buckets at intervals of size 10. Here, length corresponds to the number of terminals in the input. Each microbenchmark generated 1000 inputs for each bucket creating a sum total of 8000 inputs. We used this length threshold because, for the evaluation grammars, Nautilus$_{\mathrm{GH}}$ did not generate inputs with more than 80 terminals frequently with random walks. This occurred because generating large inputs requires triggering recursive grammar features consecutively which became probabilistically harder with each trigger.



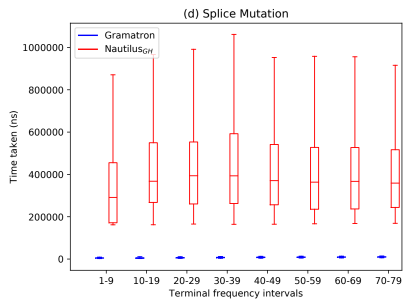
(a) Mutation Scale



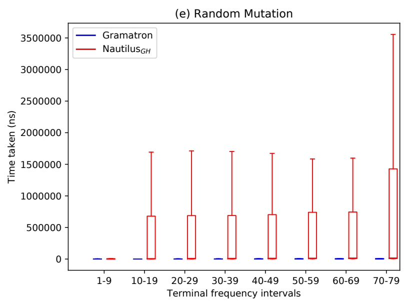
(b) Representation Size



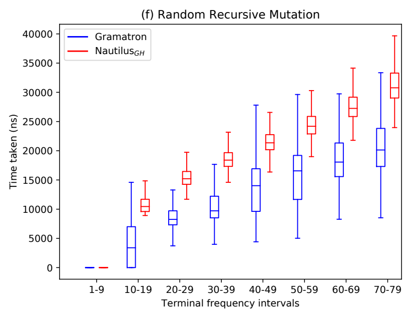
(c) Input Generation



(d) Splice Mutation



(e) Random Mutation



(f) Random Recursive Mutation

Figure 2.4. Visualization of a representative run of microbenchmark comparison of Gramatron against Nautilus$_{\mathrm{GH}}$.

To validate that Gramatron performs aggressive mutations, we compared the number of terminals in the substring generated by the mutation operator from both tools. This metric accurately quantifies the mutation scale. We compared the random mutation operator for this microbenchmark. The results are representative of the splice mutation operator as well since both are functionally equivalent. The only difference between them is how the substring is generated: for random mutation, its generated from the grammar and for the splice operator its generated from another input.

Figure 2.4a shows a representative microbenchmark run for the PHP grammar. It is evident that Gramatron performs more aggressive mutations than Nautilus$_{\mathrm{GH}}$ since the mutation scale is much larger. This is expected because for random mutation and splice, given an input string and a mutation point, Gramatron mutates till the end of the string while Nautilus$_{\mathrm{GH}}$ mutates a substring relative to the mutation point. These aggressive mutations allow Gramatron to synthesize complex bug triggers faster by performing a broader, more global search through the input state space without being tunnel-visioned on specific parts of the input state space.

Table 2.2. % improvement in terms of space, time, and mutation scale of Gramatron against Nautilus$_{\mathrm{GH}}$. Effect size quantifies the improvement magnitude (> 0.8 = large, > 0.5 = medium).

Microbenchmark	% Improvement (Effect Size)		
	mruby	PHP	JS
Representation Size	23.66 (0.44)	15.26 (0.29)	32.27 (0.66)
Input Generation	98.48 (1.88)	99.78 (1.87)	96.66 (2.15)
Splice Mutation	99.84 (3.03)	97.94 (2.45)	98.26 (1.91)
Random Mutation	26.82 (0.40)	77.75 (0.80)	73.16 (0.87)
Random Recursive	46.77 (1.19)	38.94 (0.78)	52.25 (1.11)
Mutation Scale	134.17 (-0.09)	930 (1.08)	877 (1.12)

We also calculated the percentage improvement of Gramatron by calculating the median time taken across input buckets and then comparing the mean of those medians. As evident from the "Mutation Scale" row of Table 2.2, Gramatron performed significantly more aggressive mutations. The average effect size across the different grammars is reported in Table 2.2. We see a *large* effect size for mutator aggressiveness in Gramatron.

However, for the mruby grammar, the observed effect size is minimal. This occurred due to the effect size not being robust to outliers. For this grammar, we observed the anomalous behavior that Nautilus$_{\mathrm{GH}}$ generated larger-scale mutations than Gramatron *only* for the smallest inputs with < 10 terminals. This occurred because for the smallest inputs built from this mruby grammar, the probability of Nautilus$_{\mathrm{GH}}$ extending the input further was significantly higher due to lack of nonterminal nodes for mutation. However, for inputs of all other sizes in the mruby grammar, Gramatron outperformed Nautilus$_{\mathrm{GH}}$. This is evident from the fact that Gramatron still generated on average 134.17% larger mutations as compared to Nautilus$_{\mathrm{GH}}$. Hence, we can conclude that the negligible effect size for mruby grammar is due to the outlier behavior of Nautilus$_{\mathrm{GH}}$ for small inputs.

RQ2: Generation/Mutation Efficiency

Finally, we validated whether our adopted methodology to remove the sampling bias from input distribution and perform aggressive mutations was efficient in terms of space and time. To do so, we evaluated two aspects of Gramatron: (i) space and time efficiency of automaton-based input representations, and (ii) time efficiency of the mutation operators. The evaluation methodology for these set of microbenchmarks was the same as that for evaluating aggressive mutations. From Table 2.2, we observe that Gramatron improved over all measured aspects. We see a *large* effect size along all evaluated components except space for which we see a close to *medium* effect.

To evaluate space efficiency, we compared the disk space size of their JSON-serialized representation size across inputs belonging to different length intervals, showing that our representation is more lightweight. Then, Figure 2.4c visualizes the time taken to perform input generation using grammar automatons and parse trees. Grammar automatons provide a significant runtime improvement. This is due to the fact that when a fuzzer is performing an automaton walk to generate an input, it does not need to keep track of the parse stack since its implicitly encoded in the automaton itself.

For performance evaluation of mutation operators, we designed the microbenchmarks to perform a three-step task: (i) create any metadata necessary for mutation, (ii) mutate the input and (iii) unparse the mutated input, i.e., from the input representation create a concrete test input. These three tasks together correspond to creating a mutant. The total time taken for these tasks was recorded and then bucketed based on the number of terminals in the generated mutant. We will now go over each of the mutation operators.

For random mutation, our intuition is that if the input generation is fast, this mutation will be fast too since it requires generating a new input part. Figure 2.4e validates our intuition empirically where we can see the time taken to generate mutants using Gramatron is much lesser than Nautilus$_{\text{GH}}$.

When evaluating the random recursive operator, we removed randomness introduced from the number of times chosen by both tools to multiply a recursive feature. To do so, we fixed the recursive feature multiplication, to be performed 5 times by both tools. Furthermore, inputs with less than 10 terminals did not have any recursive features. Hence, for this bucket the random recursive mutation operator performed no operations and we marked the time as zero for both tools. From Figure 2.4f, we observe a performance gain when this mutation is done on grammar automaton walks. The primary reason Gramatron is faster because finding all recursive features in an input ends up being more expensive when done on parse trees as opposed to grammar automaton walks.

The splice mutation operator requires two inputs. Hence, for each iteration we first generate two random inputs: base input and the candidate. For the base input, we generate the metadata structure corresponding to each tool. Then, we perform splicing with the candidate. Gramatron drastically outperforms $\text{Nautilus}_{\text{GH}}$ as shown in Figure 2.4d. The overhead of $\text{Nautilus}_{\text{GH}}$ comes primarily from creating metadata for the base input for performing the splice operation. Their metadata creation is more heavyweight because their mutation operator is designed to splice the candidate by searching through a corpus of base inputs rather than a single base input.

2.6.2 Faster Bug Discovery

Gramatron can find bugs with complex triggers faster in fuzz targets by using grammar automatons with aggressive mutation operators. In order to validate this claim, we created a corpus of three fuzz targets with 10 known bugs in them acting as ground truth. We created this corpus using the bugs discovered by Nautilus [18], Grammarinator [39] and from bug trackers of the fuzz targets [40]. We evaluated $\text{Nautilus}_{\text{P}}$, $\text{Nautilus}_{\text{GH}}$, and Gramatron on this corpus and compared the time taken to find each bug across ten fuzzing trials with ASan enabled and a time budget of 24 hours each.

The bug corpus used is detailed in Table 2.3; we will be referring to the bugs by their listed Bug-ID. For PHP, we used two different versions `7.2.6` [41] and `7.4.0` [42]. We sourced each of the bugs in mruby and JerryScript from six different git commits. All these bugs have syntactically valid proof-of-concept (PoC). Therefore, they all lied beyond the parser of the fuzz target.

The input state space covered by the grammar highly influences the time taken to discover a bug. Therefore, for this experiment, the grammar should challenge the fuzzer while also ensuring that the input state space is not so large that it makes it infeasible for the fuzzer to find the bug in a reasonable amount of time.

We built the input grammars using the subset of the grammars from the Nautilus repository [36]. For each bug in the corpus, we use a separate grammar. A single grammar for each target is not enough as some of our targets contain multiple bugs. A generic grammar that covers all bugs may cause the fuzzer to get stuck at finding alternate paths to a single bug hindering progress. Having different grammars for each bug allows a fair evaluation that enables the fuzzers to search for a particular bug without being distracted by other potential bugs. The only exception to this rule were PHP-3 and PHP-4. These can be triggered from the same grammar. We encountered difficulties in decoupling the bug triggers for both so we post-processed results to infer when each bug triggered.

Based on the bug PoC, each grammar consists of required *built-in* language features to trigger the bug along with non-required features to expand the input state space. Non-required features are primitives that are not in the PoC. Each grammar had the ability to build different statement types such as assignments, or function invocations. Hence, we argue that these subsetted grammars are representative of the core functionality exercised by the fuzz targets.

Note, that our tailoring primarily reduced the set of keywords to keep bugs discoverable. If we forced the fuzzers to explore thousands of different keywords with all the statement constructs, the time budget for each fuzzing campaign would exceed 24 hours by orders of magnitude. Restricting the grammars enables us to perform exhaustive testing on our bug corpus. We believe this is a fair comparison since for each bug, all fuzzers are provided *exactly* the same input space to explore, hence allowing us to evaluate how efficiently they search the input space to find the bug.

To evaluate bug discovery performance, we calculate the median time to discover the bug across ten fuzzing trials. We allowed all tools to generate their own seed corpus for each trial because of two reasons. First, at the start of each campaign, both Nautilus$_{\mathrm{P}}$ and Nautilus$_{\mathrm{GH}}$ build a corpus of seed inputs from scratch from the grammar. Hence, we did not change their default design by giving it a pre-existing seed corpus to prevent introducing unintended side-effects. Second, the seed corpus plays a pivotal role in deciding the time taken to find a bug [33]. Hence, by using a randomly generated seed corpus for each trial, we ensure an unbiased evaluation. However, a side-effect of this evaluation choice is that we could not establish statistical significance for this experiment.

Table 2.3. Performance breakdown of time to bug discovery across 10 fuzzing campaigns. *G*: Gramatron, *(N_P, N_{GH})*: Nautilus$_\mathrm{P}$ and Nautilus$_\mathrm{GH}$.

Bug-ID	Bug Type	PoC Complexity	Successful Campaigns			Median Time (s)			Effect Size	
		(Branching rules)	$\mathrm{N_P}$	$\mathrm{N_{GH}}$	G	$\mathrm{N_P}$	$\mathrm{N_{GH}}$	G	G v/s $\mathrm{N_P}$	G v/s $\mathrm{N_{GH}}$
PHP-1	Segmentation Fault	Simple (15)	10/10	10/10	10/10	1217	894	472	0.93	0.71
PHP-2	Division by Zero	Simple (15)	10/10	10/10	10/10	505	164	61	1.64	1.58
PHP-3	Segmentation fault	Simple (12)	10/10	10/10	10/10	1616	659	2288	-0.97	-2.06
PHP-4	Null-pointer-deref	Simple (17)	10/10	10/10	10/10	2509	1065	6074	-2.79	-4.30
mruby-1	Use-after-free	Complex (27)	**3/10**	**8/10**	10/10	2269	17341	4346	0.53	1.15
mruby-2	Segmentation Fault	Simple (8)	10/10	10/10	10/10	725	268	889	-0.02	-0.46
JS-1	Heap buffer overflow	Complex (26)	**5/10**	10/10	10/10	3866	335	1450	0.85	-0.67
JS-2	Failed assertion	Complex (23)	**4/10**	**9/10**	10/10	2527	1620	1199	0.60	0.71
JS-3	Failed assertion	Simple (12)	10/10	10/10	10/10	118	47	19	1.40	0.19
JS-4	Floating point error	Simple (12)	10/10	10/10	10/10	481	33	78	0.67	-0.60

Table 2.3 shows the experimental results. The *PoC complexity* column specifies the number of *branching* rules to trigger in the original CFG to generate the PoC. The original CFG here refers to the grammar used by Nautilus variants as-is and from which Gramatron generates automatons. We classify a rule as branching if its RHS contains more than one possible rewrite rule. We use this as a metric to quantify bug trigger complexity. The *Effect Size* column showcases the magnitude difference between results of Gramatron and Nautilus variants. The higher the number, the better is Gramatron and vice-versa. Nautilus did not find several bugs within the 24-hour time budget. Hence, we had incomplete data for those. We calculated effect size in such instances by only considering the trials for which we had data for both Nautilus and Gramatron. We did this to ensure we did not incorrectly estimate time-to-discovery for the Nautilus variants, potentially over-estimating Nautilus performance.

Gramatron finds all bugs consistently across all fuzzing campaigns. However, Nautilus$_\mathrm{P}$ and Nautilus$_\mathrm{GH}$ can only find seven and eight bugs out of the 10 respectively. Notably,

Nautilus_P is unable to discover mruby-1, JS-1, and JS-2 while $\text{Nautilus}_\text{GH}$ fails to find mruby-1 and JS-2 within the 24-hour time budget for certain fuzzing campaigns.

We divide the discussion based on the bug trigger complexity: (i) complex bugs: triggers with > 20 branching rules, and (ii) simple bugs: triggers with ≤ 20 branching rules. Branching rules are appropriate to quantify the complexity for generating a bug trigger. For each branching rule, the fuzzer must make a set of choices (e.g., function invocations, or utilized variable names). Increasing the number of required branching rules to generate a trigger increases the risk that a fuzzer makes mistakes (e.g., by adding incorrect function invocations, or using undefined variables).

Complex Bugs

From our ground truth corpus, mruby-1, JS-1 and JS-2 can be categorized as complicated bugs. Gramatron outperforms Nautilus_P in finding them and $\text{Nautilus}_\text{GH}$ in finding mruby-1 and JS-2. Nautilus_P fails to find these complex bugs consistently within the given time budget. This is due to the use of both AFL mutators and grammar-aware mutation operators. For the failed fuzzing campaigns where Nautilus_P did not discover the bug, we observed that the paths discovered by the AFL-mutators were disproportionately higher than other mutation operators. We attribute this to the cascading effect caused by AFL mutators generating syntactically invalid mutants. These mutants give shallow coverage in the parser but are still added to the corpus which biases Nautilus_P towards performing shallow fuzzing.

$\text{Nautilus}_\text{GH}$ tries to remediate the above problem by removing the AFL-mutation operators. However, we observe that $\text{Nautilus}_\text{GH}$ still has difficulty in discovering complex bug triggers. This difficulty arises because its mutation operators perform spot mutations (as discussed in § 2.4.3). Thus, on finding an interesting input, $\text{Nautilus}_\text{GH}$ performs localized search around the grammar parts corresponding to the input. This can be detrimental when grammar parts being explored are not relevant to the bug. Hence, this mutation policy slows down its advance towards complex bug triggers.

Gramatron outperforms Nautilus_P by performing purely grammar-aware mutations. It also outperforms $\text{Nautilus}_\text{GH}$ due to its aggressive mutation policy. As per this policy, on finding an interesting input, Gramatron does not fixate on performing localized search around the grammar parts corresponding to the input. Instead, it biases towards performing a more global search with its aggressive mutations. Hence, it rapidly progresses towards discovering complex bug triggers.

JS-1 is the only complex bug in which Nautilus$_{\mathrm{GH}}$ outperforms Gramatron. On closer inspection of the bug trigger, we noticed that even though it requires triggering 26 branching rules, all of them were localized within a specific part of the grammar. This made it ideal for the spot mutators used by Nautilus variants to discover this bug faster.

Simple Bugs

From our ground truth corpus, all other bugs apart from mruby-1, JS-1, and JS-2 can be classified as simple bugs. From the results, its evident that Gramatron outperforms Nautilus$_{\mathrm{P}}$ in finding simple bugs except mruby-2, PHP-3, and PHP-4. The mruby-2 effect size is 0.02 hence the magnitude of performance difference is close to negligible. In the case of Nautilus$_{\mathrm{GH}}$, its able to outperform Gramatron in finding the simple bugs except PHP-1, PHP-2, and JS-3.

The root cause as to why Nautilus$_{\mathrm{P}}$ fails at finding simple bugs faster than Gramatron can be attributed to its use of AFL-style mutation operators. An exception to this trend were mruby-2, PHP-3, and PHP-4 where both the Nautilus variants outperformed Gramatron. Upon inspection, we observe that their generation was highly localized to a specific part of the grammar and hence Nautilus variants were able to find it faster than Gramatron.

Nautilus$_{\mathrm{GH}}$ outperforms Gramatron in finding simple bugs. This performance gain can be attributed to the spot mutators utilized by Nautilus$_{\mathrm{GH}}$. They are better at finding simple bugs for the same reason they were worse at finding complex bugs. The localized mutations allow Nautilus$_{\mathrm{GH}}$ to find simple bugs faster.

2.6.3 Bug Discovery Performance in the Wild

To demonstrate that grammar automatons preserve bug-finding capabilities of grammar-aware fuzzers, we deployed several fuzzing campaigns against three interpreters: mruby, PHP, and Jerryscript. We chose these targets for two reasons: (i) they are widely used, which makes security bugs in them relevant, (ii) they are well-tested and have been fuzzed previously by grammar-aware fuzzers like Nautilus and Grammarinator [39], this ensures that bugs found by Gramatron are not low-hanging fruits due to lack of testing.

For each of these targets, we created grammar automatons for Nautilus grammars which are self-embedding. Hence, the grammar automatons expressed a subset of the language specified by the CFG as discussed in § 2.4.2. To quantify their size, the number of rules in each of these grammars is present in Table 2.4. The same table lists the commit ID and versions fuzzed for each of these targets.

Table 2.4. Fuzz targets with types of bugs discovered. *Key*: OOB: Out-of-bounds read, (H)BO: (Heap) Buffer Overflow, AV: Assertion Violation, and UAS: Stack use-after-scope.

Target	Version	Rules	Bugs	Bug Type
mruby	9840d6, 96bae1	1177	3	2 OOB, 1 (H)BO
PHP	7.4.8, 7.4.9	8712	4	1 UAS, 3 OOB
Jerryscript	04f0a7	589	3	3 AV

Gramatron fuzzed these targets for 10 days. During this campaign, it found 10 new vulnerabilities, so far one CVE (CVE-2020-15866) was assigned. Four of these have been responsibly disclosed to the affected vendors who have acknowledged and rolled out patches for the same. For the remaining six we are in the process of reporting them to the affected vendors. The vulnerability type breakdown and the affected applications are presented in Table 2.4. For Jerryscript, the assertion violations causes it to crash.

Bug Case Study

Here we showcase the effectiveness of Gramatron at synthesizing complex bug triggers. The bug is an out-of-bounds (OOB) read in the mruby interpreter (which has now been patched). Its root cause existed in `rehash`, an API provided by mruby for rebuilding a hash data structure. An OOB read was triggered if a hash *made* empty was rehashed and manipulated. The reason is that the internal metadata structure for the newly generated hash would point to stale metadata. This in turn caused any operations on the new hash to cause an OOB read. This vulnerability is only triggered if the original hash structure had been emptied, not if it was empty from the beginning.

To synthesize the bug trigger, Gramatron overcame two challenges: (i) emptied a populated hash and (ii) performed the correct sequence of operations on the emptied hash. The supplied grammar did not express how to add to an existing hash . So, Gramatron solved the first challenge by using another API to get a populated hash. After that, it learned how to empty it by performing the correct number of eviction operations. From there, it synthesized the API to perform the operations needed for the second challenge, which in turn triggered the vulnerability.

2.7 Discussion and Future Work

Table 2.5. Median standard deviation of the time taken to find different bug triggers.

Bug Trigger	Std Deviation Median (h)		
	$\text{Nautilus}_{\text{P}}$	$\text{Nautilus}_{\text{GH}}$	Gramatron
Simple	0.14	0.07	0.18
Hard	2.72	5.75	1.15

Trade-off between Aggressive and Spot mutators

Aggressive mutators (as used by Gramatron) primarily generate mutants that are significantly different from the source input owing to the large-scale changes. Spot mutators (as used by Nautilus) primarily generate mutants with smaller changes compared to the source input. Consequently, aggressive mutators sample from a more diverse set of mutants while spot mutators sample from a more localized set of mutants corresponding to the source input.

Since spot mutators perform local search, they have a higher chance of getting stuck in local coverage minima while trying to synthesize complex bug triggers. To validate this, we calculated the median standard deviation for different types of bug triggers used in § 2.6.2.

As evident in Table 2.5 spot mutators have a higher standard deviation when trying to synthesize complex bug triggers.

We showed in § 2.6.2 that both mutator types have their own merits. The spot mutators used by Nautilus are better at uncovering bug triggers which require exercising a specific part of the grammar. However, aggressive mutators excel at synthesizing complex bug triggers that require triggering multiple different grammar parts. A promising avenue to explore would be how a fuzzer can schedule these different mutators efficiently by crafting this as an optimization problem [43].

Augmentations to Regular Approximation

In Gramatron, we used grammar automatons to generate an under-approximation of the language specified by the CFG. An interesting avenue to explore would be generating grammar automatons that accept an *over-approximation* of the language specified by the CFG [27, 29]. Here, over-approximation refers to a superset of the language. This allows the fuzzer to generate inputs that can test both the syntactic and semantic stages of the parser simultaneously.

In the presence of self-embedding rules in the grammar, the expressiveness of the grammar automaton directly depends on the user-provided stack depth. The larger the stack depth the more expressive Gramatron can be during input generation. As a trade-off, larger stack depth increases the final automaton which is generated in terms of number of states and transitions. Consequently, the one-time cost of generating the automaton itself is larger. However, the size of the automaton will not affect the time taken to generate or mutate inputs since those operations depend on the size of the inputs themselves. As future work, we plan to explore alternative strategies for regular approximation to make more concise automatons. One alternative could be to convert grammars into GNF form in a more efficient manner [44].

2.8 Threats to Validity

Here we discuss potential threats to validity of our evaluation and the steps we take to mitigate them.

External Validity

The external validity (i.e, the generalizability of our results) primarily depends on how representative our evaluation targets are of a real-world testing scenario. To address this threat, we chose widely-used, large, and well-tested software accepting different languages as our fuzz targets. Furthermore, our in-the-wild testing experiment (described in § 2.6.3) showcases the ability of Gramatron to find previously undiscovered bugs. We do note that while our evaluation focuses on language interpreters (since related work evaluates those), Gramatron can be applied to any software that accepts inputs as defined by a context-free grammar.

Internal Validity

Fuzzers are composed of multiple modules that intertwine with each other during a fuzzing campaign. Therefore, a potential threat to our internal validity is the measurement noise that is introduced from auxiliary stages (i.e., stages that do not correspond to input generation or mutation). To minimize this noise, we built microbenchmarks that isolate the stages that we are interested in for our measurements. Another potential threat specifically pertaining to our ground-truth bug experiment was the presence of multiple bugs in our fuzz targets. This can introduce noise in the evaluation if a fuzzer finds alternate bugs before the targeted one, causing the fuzzer to tunnel-vision on that specific bug and input space around it. To eliminate this noise, we crafted grammars that discover the bugs of interest mixed with benign functionality to make it non-trivial for the fuzzers to find these bugs. Furthermore, for all our experiments against Nautilus we followed the guidelines as laid down by Klees et.al. [33] to eliminate effects of randomness while evaluating fuzzers.

Construct Validity

There is only a single threat to construct validity (i.e., we are measuring what we claim to be measuring [45]) in our experiments. This manifests when evaluating the unbiased sampling of Gramatron. In this case, a potential threat to construct validity arises out of using code coverage as a proxy metric for input diversity. However, we claim that in the context of fuzzing, we are primarily interested in higher input diversity corresponding to richer semantics. Hence, for evaluating this goal, branch coverage is the most appropriate metric as higher branch coverage directly corresponds to inputs with richer semantics.

2.9 Related Work

Fuzzing approaches can be broadly divided into two categories: mutational and generational. Gramatron is a generational fuzzer that use code coverage feedback to guide its fuzzing. We will discuss mutational fuzzing and the challenges it faces while fuzzing software that accepts structured input. Then, we will discuss existing generational fuzzers in detail and how they differ from Gramatron.

Off-the-shelf mutational fuzzers such as AFL [12, 13] use operators such as bitflips to drive the input generation from a seed input corpus. However, such mutational operators may often create syntactically invalid mutants. This leads to such fuzzers being unable to fuzz past the parser of applications that accept structured input. Gramatron uses the input grammar to overcome this limitation.

Generational fuzzers use an input model for input generation during fuzzing. This model can either be user-provided as a CFG [14, 16, 18, 46–48] or inferred from the application [49, 50]. Generational fuzzers that leverage a user-provided model can be broadly divided into two categories: (i) language-specific: Designed for fuzz targets that accept a specific language, (ii) language-agnostic: Designed to fuzz *any* type of fuzz target regardless of the type of language that it accepts. We will discuss each of these below.

Language-specific Fuzzers

These fuzzers are designed and optimized to fuzz targets that accept a specific language (e.g., for C [48, 51], or JS [15, 52, 53]). Language-specific fuzzers are customized to address language idiosyncrasies for deeper testing. C-smith [48] generates programs that avoid exercising undefined behavior as specified in the C standard. CodeAlchemist [53] employs JS-specific analysis techniques to generate semantically valid programs. DIE performs structure and type-preserving mutations to inputs by using custom-annotated Abstract Syntax Trees [52]. All such customizations come at the cost of generalizability to other languages. In contrast to these tools, Gramatron is designed as a grammar-aware but language-agnostic, generational fuzzer.

Language-agnostic Fuzzers

Language-agnostic fuzzers [14, 16, 18, 52] use techniques that do not assume anything about the target language. This enables wider applicability while still allowing for deep testing. LangFuzz [14] creates a set of code fragments from a pre-existing input corpus sourced from a test suite using the grammar. It then recombines different fragments together to create more failing inputs. IFuzzer [16] has a similar design but instead it adopts a genetic algorithm to perform the input recombination. Gramatron instead uses coverage feedback to guide its input generation and aggressive mutations using the restructured grammar.

Recent work has explored incorporating coverage feedback into grammar-aware fuzzing. Nautilus used coverage feedback in conjunction with grammar-aware and AFL-like mutation operators. After a year, the authors released another performance-optimized version, removing the AFL-style mutation operators [36]. Gramatron differs from both variants in two ways: (i) it restructures the grammar to perform unbiased sampling from the input state space, and (ii) it introduces novel grammar-aware mutation operators to synthesize complex bug triggers faster and more reliably. Superion [17] (released simultaneously with Nautilus) implements the same functionality as Nautilus; specifically, coverage-guided feedback in conjunction with grammar-aware fuzzing. Therefore, Gramatron differs from it in the same way as Nautilus. Zest [54] performs coverage-guided fuzzing using user-specified *Quickcheck-like* [55] input generators. Our approach is orthogonal to that employed by Zest in two ways: (i) we use context-free grammars modeled as grammar automatons instead of generators to perform input generation, and (ii) Gramatron performs large-scale changes using aggressive mutators while the structural mutations performed by Zest are analogous to the spot mutators used in existing grammar-aware fuzzers. While the Zest algorithm itself is language-agnostic, its current implementation is designed to test Java-based programs specifically. This constraint does not apply to Gramatron since its implementation is language-agnostic.

There have also been research efforts directed towards exploring how to make input generation from the grammar effective. Skyfire [56] is a data-driven input seed generator for fuzzers. It learns a probabilistic model for the grammar that specifies the likelihood of a production rules being triggered. This enables it to perform smarter input seed generation. Other approaches such as Dharma [57], and F1 [58] have focused on optimizing the process of input generation from the grammar itself. Gramatron adopts an approach that is orthogonal to these tools. It uses coverage feedback to guide its fuzzer that uses the grammar in conjunction with grammar-aware mutation operators to generate new inputs.

2.10 Conclusion

Fuzzing interpreters past the parsing stage is notoriously challenging, since it requires generating syntactically valid inputs. We made the observation that grammar automatons coupled with aggressive mutations enable a fuzzer to reach and trigger complex bugs in interpreters effectively. Our prototype implementation, Gramatron, uses grammar automatons which restructure the grammar to perform unbiased sampling from the input state space. The unbiased sampling coupled with aggressive mutations allows Gramatron to find deep bugs with complex triggers. In addition to discovering all 10 bugs in our benchmark, Gramatron also discovered 10 new bugs in popular interpreters. Gramatron is available at https://github.com/HexHive/Gramatron.

3. FIRMFUZZ: AUTOMATED IOT FIRMWARE INTROSPECTION AND ANALYSIS

3.1 Introduction

With 30 billion expected embedded devices by 2020 [59], the Internet of Things (IoT) has already proliferated across all aspects of our lives. The rise of IoT devices has been accompanied by increasing attacks on or through them. These attacks range from forming a botnet of embedded devices like the Mirai botnet [60] to a myriad of exploitable vulnerabilities that are reported in the corresponding firmwares [61–65].

In this work, we focus on analyzing Linux-based firmwares due to its widespread adoption. A Linux-based embedded firmware has three major parts that can be exploited by an adversary: (i) a variant of the Linux kernel, (ii) a set of open-source software packages, and (iii) a set of custom vendor-developed applications. The first two components are used in widely different contexts and can be vetted independently from embedded systems. The third component, custom vendor-developed applications, may be more prone to vulnerabilities because these proprietary applications are neither open-source nor openly vetted. We therefore focus on vendor-developed applications.

Evaluating vendor-developed applications in embedded firmware presents three challenges. First, these applications only accept syntactically legal inputs, therefore in order to perform deep analysis one has to infer and respect this syntax. Inferring the input syntax from a blackbox binary is a non-trivial problem [56, 66]. Second, to ensure triggered vulnerabilities do not escape the analysis, fine-grained monitoring of the runtime environment is required. Third, these applications rely on the runtime environment of the firmware for their functionality. Therefore, fuzzing these applications as standalone binaries is not sufficient.

In this chapter, we describe an analysis framework, FirmFuzz, that finds *deep* vulnerabilities in vendor-developed applications of embedded firmwares. We overcome the above-mentioned challenges in FirmFuzz by: (i) utilizing the web application interface of these embedded devices as entry points to generate syntactically legal inputs, (ii) injecting runtime monitors into the runtime environment of the embedded firmware to allow for context-aware monitoring, and (iii) emulating the firmware image to keep our approach device-independent.

To further enhance our greybox-based generational fuzz testing capability, we leverage information collected from static analysis to guide our fuzzer.

Previous efforts have studied emulation on a large-scale for closed-source firmware [63, 64]. These efforts focused on scaling an analysis to many images, but the undertaken analysis was generic, searching for specific vulnerabilities. An off-the-shelf analysis may miss vulnerabilities because it is not tailored to embedded systems and does not inspect the actions on the system.

Although we use web applications as an end-point for dynamic analysis, using web application scanners is insufficient. These scanners treat the analysis target as a blackbox while performing the vulnerability assessment. Emulating the firmware allows us to tune the runtime environment and to introspect the running system during execution, observing the vulnerability scanner interactions with the system. This, in turn, allows us to find *deep* vulnerabilities.

We provide a framework that, after some light-weight configuration, adapts to the emulated firmware and performs context-sensitive analysis. The focus of our work is not the *breadth* (number of images analyzed) but the *depth* (testing deep code paths for a variety of vulnerabilities) of the analysis undertaken.

Unlike conventional web application scanners, FirmFuzz leverages the degrees of freedom offered by an emulated firmware to enhance the vulnerability detection process. It integrates runtime monitors into the firmware filesystem, modifying the firmware itself to improve the bug finding process. Using our syntactically-legal input generation strategy and runtime monitors, we tailor our analysis on a per-firmware basis allowing us to trigger *deep* bugs in the firmware.

We analyzed 6,427 firmware images scraped from three vendors (Netgear, D-Link, and TRENDNet). Out of those, in 32, we found seven previously unknown vulnerabilities across six different devices comprising of two IP cameras and four routers. The vulnerabilities discovered include one post-authentication Command Injection (CI), three pre and one post-authentication Buffer Overflow (BO), one pre-authentication reflected XSS vulnerability and one pre-authentication Null Pointer Dereference (NPD). For the sake of responsible disclosure, we informed the vendors of these vulnerabilities.

In summary, we make the following contributions:

- We develop FirmFuzz (with open source code available at https://github.com/HexHive/FirmFuzz), an automated emulation and dynamic analysis framework for finding *deep* vulnerabilities in embedded firmware.
- We develop a generational fuzzer for syntactically legal input generation that leverages static analysis to aid fuzzing of the emulated firmware images while monitoring the firmware runtime (helper binaries and kernel monitors to enable deterministic bug discovery).
- We automatically test firmware images scraped from vendor websites and find seven previously unknown vulnerabilities.

3.2 Firmware Preprocessing

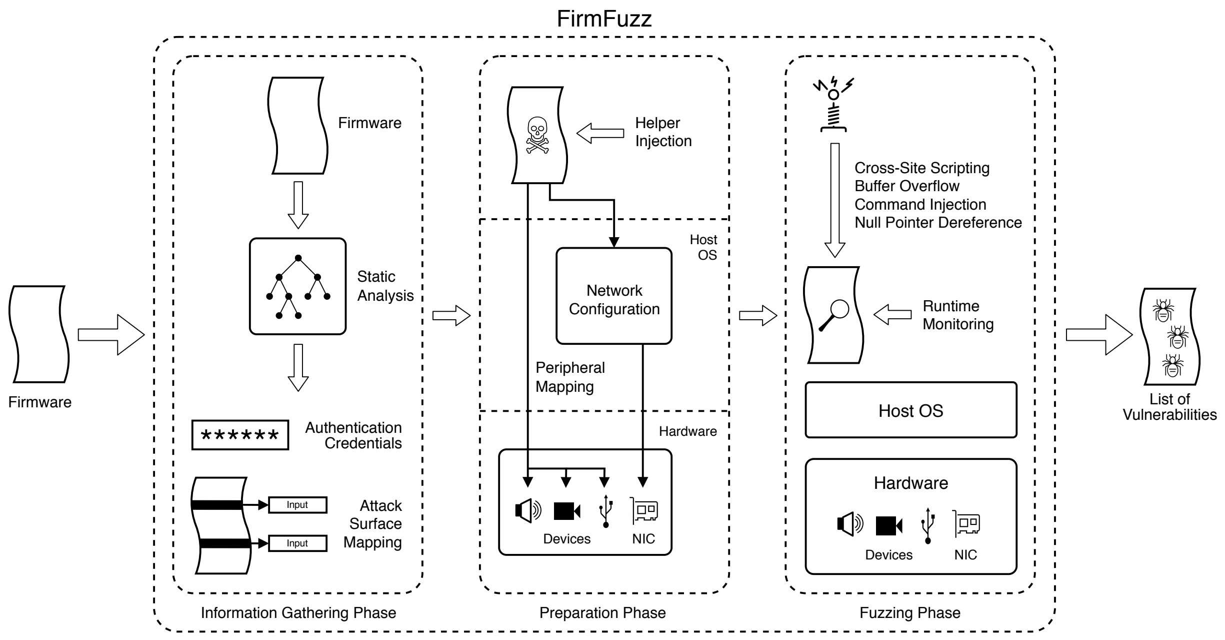


Figure 3.1. FirmFuzz Workflow

FirmFuzz is a framework for the automatic analysis and fuzz testing of Linux-based IoT firmware through a QEMU-based emulation layer. It analyzes firmwares in three phases:

information gathering, preparation, and fuzzing as depicted in Figure 3.1. Our framework currently supports analyzing MIPS-architecture and little endian ARM-architecture based firmware images.

3.2.1 Information Gathering Phase

This phase serves two goals. First, discovering the username-password pair required for authentication with the web application in order to increase the coverage of the fuzzer. Second, static analysis of the attack surface to find inputs for vulnerable code paths in PHP.

FirmFuzz brute-forces authentication credentials for a firmware through a crowdsourced credential corpus [67].

CI vulnerabilities in PHP-based applications arise from unsanitized user input being passed to unsafe PHP functions e.g., `system`, or `shell_exec`. FirmFuzz performs intraprocedural static taint analysis of the PHP scripts. It taints user-controlled variables, `$_GET`, `$_POST` and logs the code paths where the taint flows to unsafe functions. For each of these code paths, a constraint set is built. This set is used by FirmFuzz to generate inputs that can trigger vulnerable code paths.

3.2.2 Firmware Preparation

This phase creates a firmware image ready for emulation with a corresponding emulator configuration. A close approximate of the runtime environment as expected by the firmware is created using the peripheral mapping strategy. FirmFuzz leverages full-system emulation and injects helper binaries into the filesystem and augments the kernel to discover vulnerabilities with a low false positive/negative rate. The network of the emulator backend is configured to allow interaction between the firmware programs and the bug finding tools.

Peripheral Mapping

An embedded firmware often expects the presence of certain hardware peripherals during boot-time, runtime or both.

In the absence of these peripherals, a firmware may silently log an error or go into a busy loop trying to query the peripheral. In the former case, it is hard to infer what side-effect the absence of a peripheral may have on the functionality of the emulated firmware. In the latter case, the firmware may not reach a stable state.

If the firmware requests an unknown peripheral, FirmFuzz provides a mapping of that peripheral to a fake peripheral driver that always returns `True` on being queried. We acknowledge that this approach may not result in a stable state for all firmwares due to the diverse set of available IoT peripherals.

The firmware image is run under an emulator with our custom kernel. The kernel is configured to panic if unsupported devices are accessed. FirmFuzz uses this panic log to create a mapping of the device to the fake device driver. FirmFuzz iteratively performs this process until all the unsupported devices are mapped to a fake device.

Helper Injection

Helper binaries allow FirmFuzz to inspect the firmware during emulation. The helpers operate within the runtime environment of the firmware and in the current implementation allow us to detect CI vulnerabilities. During fuzzing, if this helper binary is executed, FirmFuzz flags a CI vulnerability. This approach is completely automated and firmware agnostic; i.e., we are not reliant on the utilities present on the firmware to detect the vulnerability. The closest previous work to ours, Firmadyne [63], does not have support for the detection of CI vulnerabilities. The ones reported in its paper were discovered through manual analysis of the webpages.

To detect BO vulnerabilities, we utilize the exception handling mechanism of the Linux kernel similar to Firmadyne. FirmFuzz leverages the mechanism further to detect NPD vulnerabilities. The major difference between FirmFuzz and Firmadyne in detecting BO is that the Firmadyne authors manually discovered a BO vulnerability in a webpage served by a specific Netgear router which they validated by crafting a curl request to that specific webpage to trigger that vulnerability. However, FirmFuzz requires no manual analysis to detect BO vulnerabilities. Using the custom fuzzing driver, it automatically triggers BO vulnerabilities in the emulated firmware and packages a PoC exploit to recreate the vulnerability. For the XSS vulnerability targeted by FirmFuzz, we observed that host-side monitoring is sufficient for detection.

Network Configuration

Firmware images differ in how they name and assign addresses to their LAN/WAN interfaces. We follow the same approach as Firmadyne to infer these network configurations. We first run the emulation in a 'network inference' mode in which FirmFuzz logs all the interaction of the firmware with the networking interface of the kernel. Using these logs, FirmFuzz infers the network configuration and creates the appropriate virtual network interfaces to allow interaction with the emulated firmware.

3.3 Firmware Fuzzing

FirmFuzz detects vulnerabilities using a custom-developed automated generational fuzzer. Existing vulnerability scanning approaches for embedded firmware [63, 64, 68] require human guidance.

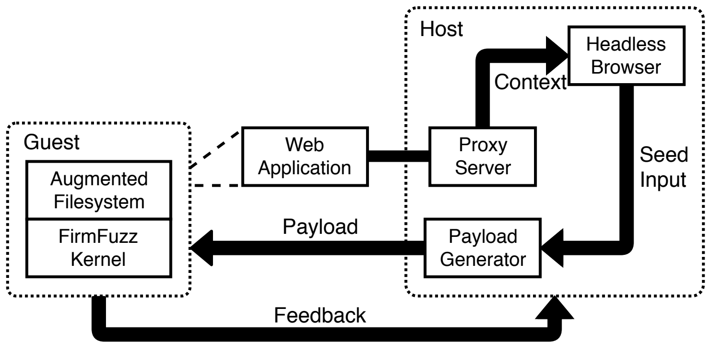


Figure 3.2. FirmFuzz fuzzing workflow

FirmFuzz changes the paradigm of vulnerability detection in emulated firmware by augmenting the emulation environment of the fuzzed target to aid in vulnerability discovery, see Figure 3.2. Our approach removes the reliance on server feedback and allows a direct observation of the triggered vulnerability *in situ*. Three main features of the FirmFuzz fuzzer are: (i) *Context-driven input generation* — It incorporates contextual information provided by the firmware while interacting with different parts of the attack surface, (ii) *Deterministic vulnerability detection* — The vulnerability monitors operating both in the guest (i.e., the emulated firmware) and the host allow deterministic vulnerability detection, and (iii) "Fuzzing side-effects" elimination — FirmFuzz with the help of its emulation framework, automatically reverts the firmware back to a stable state if the firmware reaches an inconsistent state while being fuzzed. This allows continuous fuzzing of the emulated target without requiring manual intervention to reset the firmware state.

The web application setup by the firmware provides its functionality by employing a combination of client-side JavaScript code and server-side code. FirmFuzz uses a headless browser controlled by our fuzzer to interact with the firmware through the web application to execute the client-side JavaScript code. Using the contextual information from these interactions, it generates fuzzing inputs. All network traffic between the fuzzer and the emulated firmware passes through a proxy server. This allows the proxy server to capture candidate inputs that will be mutated by FirmFuzz.

The four vulnerability types targeted by FirmFuzz during the fuzzing phase are: CI, BO, NPD, and XSS. To detect these vulnerabilities, FirmFuzz first interacts with the web application to generate a legal HTTP request as seed input to fuzz a part of the attack surface, see § 3.3.1. It then mutates the legal request with payloads based on the vulnerability being targeted, and sends it to the firmware. Upon sending a mutated request, FirmFuzz monitors the firmware to detect vulnerabilities. If a vulnerability is detected, the mutated request is logged as a Proof-of-Concept (PoC) input along with the recipient URL to allow reproducibility.

3.3.1 Syntactically Legal Input Generation

Since FirmFuzz uses a headless browser for firmware interaction, the burden of creating a well-formed HTTP request that effectively tests the vendor-written software is offloaded to the web application. Additionally, the browser handles executing client-side code to give itself access to the full functionality of the application.

However, using a web application interface as an oracle to generate syntactically-legal input opens a challenge. FirmFuzz must be aware of the web interface setup of the emulated firmware in order to interact with it successfully. During our experiments, we observed that the web interface setup highly varies across vendors and devices. During evaluation of FirmFuzz, based on the images we fuzzed, we created templates for interacting with the encountered web interfaces. Note, however, that this interface support requires some manual validation by the analyst. This minimal manual validation is the limiting factor for scaling the analysis to additional devices. This highlights one of the primary differences between FirmFuzz and the previous work in this area: we strive to perform deeper analysis, but at a *slight* cost to scale.

3.3.2 Deterministic Bug Discovery

Current *automated* approaches for detecting vulnerabilities (CI, BO and NPD) employed by scanners like ZAP [68] are reliant on the server-side response for detecting them. This approach is imprecise as it may either fail to localize a vulnerability, e.g., BO, NPD, or miss a vulnerability altogether, e.g., CI if a particular Linux utility does not exist. Even if a vulnerability is detected, there is no guarantee that it exists since the server-side response *may* be inaccurate.

To deterministically detect the above mentioned vulnerabilities, FirmFuzz modifies the emulation environment of the firmware and the firmware itself.

FirmFuzz detects CI, BO and NPD vulnerabilities by monitoring the logs generated by the augmented firmware when a test input is executed. For CI, it monitors the `execve` system call to log execution of the poison binaries (see § 3.2.2). For BO and NPD, it monitors the kernel logs to see if any firmware process tried to access unmapped memory.

Detecting an XSS vulnerability does not require any guest-side assistance; host-side monitoring by FirmFuzz is sufficient to detect it, similar to existing vulnerability scanners. This monitoring is in the form of detecting if the snippet of JavaScript code that was sent as a payload in the request is executed in the machine.

3.3.3 Elimination of Fuzzing Side-Effects

Since FirmFuzz actively interacts with the emulated firmware during fuzzing, it may become unresponsive. The inconsistent states include the firmware trying to reboot itself to apply certain changes or an infinite cycle due to the triggered functionality being emulated incorrectly. In such a state, the firmware becomes non-responsive to fuzzing and requires a roll back to a stable state.

The above problem often requires manual intervention to perform the rollback. However, to keep the fuzzing process fully automated, FirmFuzz employs a "snapshot and rollback" strategy. During the dynamic analysis, if the firmware is pushed into an inconsistent state, FirmFuzz reverts the emulated firmware back to a stable state (right after the completion of initialization) to be ready for the next request. This allows for continuous fuzzing of the target without manual intervention.

3.3.4 Payload Delivery

The web application performs client-side validation checks on the input values, such as checking if the input values are formatted according to the syntax expected by the application. Therefore, to effectively fuzz the firmware, a legal request from the web application is required, which can be mutated and sent to the firmware directly bypassing the client-side validation checks.

Our fuzzer requires a set of seed inputs and valid URLs that are accepted by the web application. As web applications often consist of a JavaScript host component and a server component, we need to trigger the generation of all possible host paths by driving their host component. FirmFuzz leverages the headless browser to load webpages from the firmware and then walks through the DOM to trigger all possible state changes by iterating through all button elements on the web page (often firing JavaScript events along the way). In addition, our host components fill input fields with syntactically *legal* values that are inferred through the names of fields and a set of possible data ranges.

If FirmFuzz infers legal input values, then a seed request is generated by interacting with a button element. This seed request is mutated for fuzzing. The mutation strategy followed by FirmFuzz is a substitution strategy in which the relevant parameters of a request are modified to contain malicious payloads. This substitution strategy is used across all the vulnerability detection modules of FirmFuzz. The substituted payload is adjusted based on the vulnerability being tested. Note that a finite number of payloads are tested for each vulnerability. Therefore, a fixed number of mutants are created for each seed request.

Methodology

Here, we provide an extended description of how FirmFuzz generates inputs to fuzz test the emulated firmware. FirmFuzz uses Algorithm 1. First, on receiving a webpage served by the web application, FirmFuzz finds all the button elements in it. These elements are used to interact with the firmware and are added to a list using the `find_buttons` subroutine.

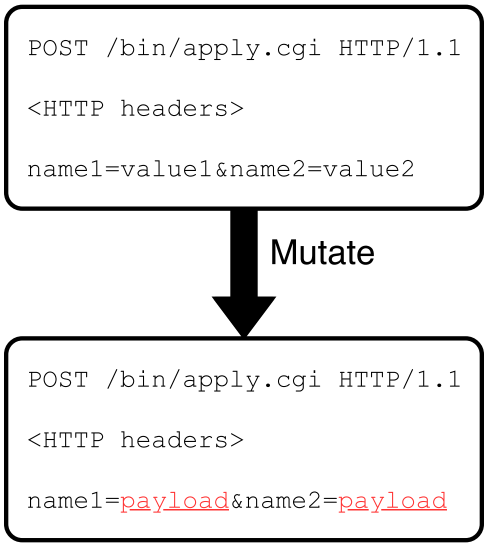
Algorithm 1 Payload delivery

```
Require:
  WebPage: The current webpage being evaluated
  Firmware: The emulated firmware
  procedure DELIVER_PAYLOAD(WebPage)
      $buttons[] \leftarrow WebPage.find\_buttons()$
      $input[] \leftarrow WebPage.infer\_input()$
      while len(buttons)>0 do
          send_mutate(input, buttons[0])
          if Firmware.isInconsistent() then
              Emulation.restore( )
          end if
          buttons.pop( )
      end while
  end procedure
```

Second, FirmFuzz needs to infer legal values for the input fields presented by the interface performed by the `infer_input` subroutine. FirmFuzz employs two strategies based on whether an input field is empty or non-empty: (i) *Non-empty field*—Field is populated with a default value filled in by the firmware. In this case, FirmFuzz leaves the value unchanged. (ii) *Empty field*—FirmFuzz does a pattern match on the HTML ID attribute of the corresponding input element. Based on whether the ID attribute contains familiar strings such as “mac” or “IP”, a dummy MAC address or an IP address is filled into the input field respectively. If none of these strings match then FirmFuzz fills the input field with dummy text.

The names and types of these ID elements (parameters) are not based on a standard but chosen by vendors who developed the web application. Therefore, it is possible that the heuristic employed by FirmFuzz unable to infer the legal value requested.

In the event that FirmFuzz incorrectly infers values for the input fields and cannot make the web application generate a request, the "ID" attribute is logged for manual classification later as either an input "ID" for a MAC or an IP address. FirmFuzz maintains a database for the incorrectly inferred input elements so that future runs of FirmFuzz can better infer legal values.



```
POST /bin/apply.cgi HTTP/1.1

<HTTP headers>

name1=value1&name2=value2
```

```
POST /bin/apply.cgi HTTP/1.1

<HTTP headers>

name1=payload&name2=payload
```

Figure 3.3. FirmFuzz Mutation Strategy

If FirmFuzz manages to infer legal input values, then `send_mutate` subroutine will generate a seed request by interacting with a button. This seed request will then be mutated for fuzzing. The mutation strategy followed by FirmFuzz is a simple substitution strategy in which the relevant parameters of a request are modified to contain malicious payloads as depicted in Figure 3.3. A similar strategy is followed for `GET` requests to mutate its parameters. This substitution strategy is used across all the vulnerability detection modules of FirmFuzz. The substituted payload is adjusted based on the vulnerability that is being tested by FirmFuzz.

At any point during the seed request generation or the mutated requests delivery, the firmware reaches an inconsistent state, the emulation is rolled back automatically to a stable state by FirmFuzz and the fuzzing is carried forward.

3.4 Evaluation

Table 3.1. Firmware images tested

Vendor	Scraped Images	Linux FS found	Network Inferred	Fuzzed (Unique Devices)	Unique Web UI
TRENDnet	359	129	26	6 (5)	2
Netgear	2646	675	162	20 (17)	3
D-Link	3422	209	15	6 (5)	1
Total	6,427	1,013	203	32 (27)	6

We evaluate FirmFuzz by testing it on a set of 6,427 firmware images. We first give a breakdown of the dataset on which FirmFuzz was able to be evaluated as well as the number of unique web interfaces encountered. We discuss one of our discovered bugs as a case study and how the discovered vulnerabilities are not detected by existing state-of-the-art tools. We also evaluate our runtime performance and discuss our vulnerability detection accuracy.

FirmFuzz is evaluated on a machine with an Intel i7 processor and 16GB RAM and running Ubuntu 16.04. To emulate firmwares, FirmFuzz uses QEMU [69] version 2.5.0 as the emulation backend. For fuzzing, FirmFuzz uses a headless browser as one of its fuzzing drivers. The headless browser used is the Selenium WebDriver [70] version 3.4.0. In addition, FirmFuzz uses a proxy server, mitmproxy version 0.18.2 [71], to monitor the network traffic sent between the fuzzer and the emulated firmware.

3.4.1 Firmware Images Tested

6,427 images were scraped from three vendor websites to create our image dataset. From the dataset, 1,013 images had a Linux-based File System (FS). Out of these, 203 images had their network configuration inferred and 32 images from this set had accessible web interfaces which were used by FirmFuzz as entry points for fuzzing the image. The breakdown is presented in Table 3.1.

There is a sharp drop-off between the images for which the network configuration was inferred and those which were successfully fuzzed. This drop-off occurs due to missing emulation for specific required devices (e.g., a system waits for a camera to be accessible before starting the web server). Without an accessible web interface as an entry point, FirmFuzz cannot fuzz the firmware image.

Out of the fuzzed images,there is high reusability of web interfaces between different devices from the same vendor — 6 unique web interfaces in 27 unique device images across 3 different vendors. Therefore, with minimal manual effort we can cover a large number of emulated devices.

3.4.2 Case Study

FirmFuzz, using its analysis detects vulnerabilities hidden deep in the firmware which are not immediately apparent. A case study of such a vulnerability is presented below.

TRENDnet TEW-673GRU Router

This is a MIPS-based Wireless Gigbabit router. A CI vulnerability was found in this wireless router. It can be remotely exploited if a user is logged in to the device's configuration webserver with administrative credentials.

The vulnerability exists in the vendor-written program `timer` on the firmware. This program runs by default as a daemon on the router with root privileges. The `timer` uses another vendor-written software `arpping` to periodically check if the router is reachable every three minutes.

The vulnerability lies in the passing of the parameters from `timer` to `arpping`. Five parameters are retrieved from device memory and passed to `arpping` without any validation or sanitation. Out of the five parameters, three of them are under user-control through a web application setup by the router which performs client-side validation on these parameters.

FirmFuzz using its syntactically legal input generation, inferred the input request and the CGI binary responsible for updating those user-controlled parameters. FirmFuzz then sent a `POST` request targeted at discovering a CI vulnerability directly to the CGI binary which updates the device memory with the sent values respectively without validating them. Using the firmware runtime monitoring, FirmFuzz detected the vulnerability because it was triggered every three minutes by the `timer` program.

3.4.3 Comparison with Existing Analysis Frameworks

To show the effectiveness of FirmFuzz, we compare against other state-of-the-art open-source web vulnerability scanners. We chose two of the most popular ones, Zed Attack Proxy (ZAP) [68] and w3af [72], for our evaluation. We also evaluated the automated vulnerability detection of Firmadyne on our sample set of firmware images as well. The evaluation was performed on the basis of the number of vulnerabilities found by the tools.

Table 3.2. Vulnerability Detection Comparison of FirmFuzz against Zed Attack Proxy(ZAP), Firmadyne and w3af

Number	Vulnerability	Vendor	Device	CVE-ID	FirmFuzz	ZAP	Firmadyne	w3af
1	Command Injection	TRENDnet	TEW-673GRU	CVE-2018-19239	✓	✗	✗	✗
2	Reflected XSS	TRENDnet	TEW-634GRU, 673GRU, 632BRP,	–	✓	✓	✗	✗
3	Buffer Overflow	TRENDnet	TEW-673GRU, 632BRP	CVE-2018-19242	✓	✗	✗	✗
4	Buffer Overflow	TRENDnet	TV-IP110WN, IP121WN	–	✓	✗	✗	✗
5	Buffer Overflow	TRENDnet	TV-IP110WN, IP121WN	CVE-2018-19240	✓	✗	✗	✗
6	Buffer Overflow	TRENDnet	TV-IP110WN,IP121WN	CVE-2018-19241	✓	✗	✗	✗
7	Null Pointer Dereference	Netgear	DG834	–	✓	✗	✗	✗

Firmadyne

Firmadyne runs a set of Metasploit modules of known exploits for embedded devices during its automated dynamic analysis. Additionally, it tests the emulated image for a set of vulnerabilities that the authors of the framework found manually in some embedded devices.

As evident from Table 3.2, this approach, even though completely automated and applicable at large-scale, fails to find any of the vulnerabilities in our sample set of firmware images. The Firmadyne tests were originally built for specific embedded devices and the probability of the same exploit working across the different devices and different vendors is low.

w3af

As w3af cannot infer the credentials to the administrative interface by itself, we provide it as a head start for a fairer comparison. However, even after providing the necessary credentials to the authentication plugin and configuring several parameters including the HTML tags for the input fields manually, w3af was still unable to authenticate with the firmware. This is the reason why w3af failed to detect any of the vulnerabilities detected by FirmFuzz.

Zed Attack Proxy

To ensure that ZAP had access to the same attack surface as FirmFuzz, the credentials to the firmware for administrative access were provided to ZAP as it does not have the authentication discovery capability of FirmFuzz. With the credentials provided, we ran the *Active Scan* feature of ZAP on the emulated image which is an automated scan feature that tries to find vulnerabilities by deploying known attacks including the ones FirmFuzz targets.

As can be observed in Table 3.2, ZAP only discovered the XSS vulnerability but failed to discover any other vulnerability. This is because ZAP treats the firmware as a black box during its automated scan and does not actively interact with the firmware to gain context about the application like FirmFuzz does. Additionally, ZAP does not have the capability to monitor the runtime environment of the firmware to detect any erroneous conditions while performing the scan. This limits its capability to detect those vulnerabilities which, when triggered, do not provide any feedback to the scanner through the web application entry point.

Even if the PoC for the discovered vulnerabilities in the image are provided to ZAP, it still cannot infer their existence. This is because ZAP relies on overtly observable signals for vulnerability detection. For example, for CI vulnerabilities, ZAP relies on server-side replies. The vulnerabilities we discovered do not send such a response from the server when triggered because they exist in an auxiliary program that does not interact with the web application directly.

3.4.4 Runtime Performance

We further evaluate the runtime performance of FirmFuzz during the fuzzing phase. The average runtime for the fuzzing phase is *16m 42s*. The comparatively low runtime overhead is primarily because our fuzzer is a generational fuzzer rather than a mutational one. With our generational fuzzer, we constrain the state space of inputs drastically, thus achieving better overhead.

3.4.5 Vulnerability Detection Accuracy

As shown in Table 3.2, existing automated scanners incur a high false negative (FN) rate when testing web applications. Their automated scans employ a brute-force approach to detect vulnerabilities. The brute-force approach includes strategies such as mutating the URL under test with payloads and performing blind directory traversals. The brute-force approach, reliance on server feedback for detecting such vulnerabilities, and the the automated scans being blind, i.e., they do not actively interact with the application to gain context about the web application leads to the high FN rate.

FirmFuzz using its runtime monitoring of the firmware under test along with the contextual input generation lowers the number of FN bugs while detecting BO, NPD, and CI compared to the existing automated scanners. We do not completely remove all instances of FN bugs since FirmFuzz relies on template request generation for fuzzing. Therefore, if a request is not generated for a particular page using our heuristics then a bug may be missed. However, such instances are clearly logged by FirmFuzz and an analyst can provide feedback to FirmFuzz in terms of acceptable input for the page which can be used to lower the chances of a FN bug.

As discussed in § 3.2.2, FirmFuzz monitors the execution of a poison binary and memory access violation handler in the kernel for detecting CI and BO, NPD respectively. These detection methods not only ensured that none of the bugs caught by FirmFuzz that belonged to the class of CI, BO, and NPD were false positives (FP) but also gave information which firmware resource was buggy. This is a drastic improvement over the existing scanners which, due to their lack of system introspection, are neither able to localize a bug if it exists or give guarantees that the bug detected by these scanners is not a FP warranting further manual analysis.

3.5 Discussion and Future Work

An inherent limitation of the dynamic analysis component of FirmFuzz is that it can only be deployed on firmware images that are successfully emulated by the framework. Future advancements in emulation techniques will allow analysis of more firmware images. Improving mapping and analysis techniques for firmware images will be our prime focus of future work.

The set of devices which an embedded system may use is inexhaustible due to the diverse use-cases of IoT systems. Trying to support all possible devices in the emulation layer is infeasible. In this work, we use the opportunistic approach of mapping unsupported devices to *fake* devices. However, this approach may not result in a steady state for all firmwares. Devices may expect a different return value than the default one returned by our fake driver. As an extension to our approach, the firmware can be analyzed statically or dynamically to infer the return values expected by the firmware from the device for more effective device mapping. We leave this to future work.

In our prototype, we mutate all parameter values to the payload targeting a vulnerability. This mutation strategy may be improved further by performing selective mutation based on analysis of the targeted attack surface to increase coverage.

The firmware introspection and inspection carried out by FirmFuzz in the form of its dynamic and static analysis modules can also be enhanced with more sophisticated analyses. To improve coverage, a more fine-grained static analysis can be carried out to find the control parameters to be mutated. Similarly, the detection modules could retrofit sanitizers [73] to detect more complex memory corruption bugs.

3.6 Related Work

A large body of work has contributed to security analysis of firmware images for embedded devices. However, the closest efforts to our work in terms of the targeted device domain (i.e., embedded Linux-based devices) are the framework by Costin et al. [64] and Firmadyne, the framework by Chen et al. [63].

Firmadyne

Chen et al. [63] presented Firmadyne, a full-system emulation based framework for dynamic analysis of embedded firmwares. They carried out the most extensive analysis in terms of the number of firmware images analyzed to date. However, dynamic analysis done by Firmadyne was simple. Their only automated vulnerability discovery pass consisted of running known exploits as Metasploit modules and their own PoC for manually discovered vulnerabilities.

Running a pre-defined set of exploits, while helpful in finding known vulnerabilities, is not effective in discovering new ones (as evident from Table 3.2) since the probability of the same exploit invoking different vulnerabilities across different classes of devices and vendors is low. On the contrary, FirmFuzz tailors the payloads to the target emulated firmware allowing it to test deep code paths and finds new vulnerabilities.

Firmware Analysis by Costin et al.

Costin et al. [64] built an emulation framework targeted specifically at emulating the web interface of the firmware and performed static and dynamic analysis on it.

Their approach was dependent on existing tools, RIPS [74] for their static analysis of PHP scripts, vulnerability scanners like [68] for the dynamic analysis. These tools, however, have high chances of false positive and false negative rates respectively.

FirmFuzz, on the other hand, incorporates its own custom tools into the framework to lower the false positive/negative rates. The static discovery module is constrained to look only for potential CI vulnerabilities and is designed to output constraints for each vulnerable code path present. The dynamic analysis module (i.e., the fuzzer) is aware of the other modules in FirmFuzz and leverages all information available from them, e.g., output from the runtime monitors on the emulated image and the information acquired during the static analysis phase. This provides enhanced bug discovery.

3.7 Conclusion

The increasing range of IoT devices have access to our personal data and impact our everyday life, calling for additional scrutiny when evaluating their security. We present FirmFuzz, an automated framework for whole-system emulation and fuzz testing of embedded firmware images. We used FirmFuzz to test for four types of vulnerabilities in the firmware images that we studied: CI, BO, NPD and XSS. We found and reported seven previously undiscovered vulnerabilities using FirmFuzz.

4. SIEVEFUZZ: TARGET-TAILORED PROGRAM STATE RESTRICTION

4.1 Introduction

Quality assurance is an important component of the software development life cycle, requiring significant resources for identifying, triaging, and fixing defects both pre- and post-deployment. In working toward offsetting this burden, the last two decades has seen software *fuzz testing* (fuzzing) become the most successful and ubiquitous approach for automated software defect discovery.

Most fuzzers target *broad* defect discovery (e.g., OSS-Fuzz [75], libFuzzer [76], and AFL++ [77])—embracing *code coverage guidance* to explore the software under test (SUT) by maximizing code coverage of generated test cases. But, despite the success of coverage-guided fuzzing [78–82], its from-scratch, all-or-nothing exploration style is unsuited to the many critical software QA tasks that target *specific* code locations (e.g., bug reproduction, regression testing, or patch testing). In such contexts, software testers instead turn to targeted fuzzing approaches known as *directed fuzzing*.

Directed fuzzers replace fuzzing's conventional broad search with one targeting predetermined locations (e.g., a suspected defect location), using *distance minimization* [83–85] to drive fuzzing closer and closer to them. To achieve directedness, distance minimization computes the distance of every generated test case relative to each target location, saving only those that shorten this distance as fuzzing continues. However, as distance measurement is performed at runtime for *all* test cases—including the overwhelming majority that are *incapable* of ever reaching the target locations—directed fuzzers incur significantly more overhead per execution due to the higher instrumentation cost associated with distance measurement. Furthermore, the current scheme of using distance minimization is specifically ill-suited for *disjoint* target locations—locations that can be reached without requiring a large part of the software functionality to be exercised. For such target locations, distance minimization's costly, always-on analysis becomes overwhelmed by *target-unreachable* paths, thus slowing down directed fuzzers' progress—beyond even their undirected counterparts.

To break free from distance minimization and quickly filter-out target-unreachable paths, we introduce *tripwiring*: a lightweight approach to accelerate directed fuzzing through a target-tailored restriction of program state. At the core of our efforts is our observation that a fuzzer's search is stochastic and highly influenced by the program's observable code coverage; and should a code region be made inaccessible, a fuzzer's exploration will shift toward pursuing whatever program paths *remain* accessible. We demonstrate that, through a hybrid static and dynamic analysis technique, it is feasible to identify and refine the set of target-relevant code regions while tripwiring (i.e., preempting and terminating) target-irrelevant ones—enabling effective directed fuzzing of disjoint target locations that is unburdened by distance minimization.

To evaluate tripwiring's effectiveness, we implement a proof-of-concept directed fuzzer called SieveFuzz, and evaluate it alongside the state-of-the-art directed fuzzers AFLGo [83] and BEACON [86], as well as the state-of-the-art undirected fuzzer AFL++ [77]. We examine a real-world context in which directed fuzzing is deployed for targeted defect discovery—reproducing third-party-reported security vulnerabilities—and demonstrate that across a corpus of ten disjointly-located security vulnerabilities in nine varied benchmarks, tripwiring accelerates directed fuzzing by an average of **140%**, **93%**, and **118%** faster than AFL++, AFLGo, and BEACON, respectively, while obtaining **37%**, **42%**, and **61%** more consistent targeted defect discovery, respectively.

In summary, we make the following contributions in this chapter:

- We introduce *tripwiring*: a lightweight technique for target-tailored directed fuzzing that restricts fuzzing to only the program search space guaranteed relevant to reaching user-determined target locations.
- We expose the fundamental limitations that impede state-of-the-art directed fuzzers from achieving effective *and* efficient directedness for disjoint target locations. For such target locations, we show that tripwiring is a more optimal directed fuzzing methodology than distance minimization.
- We design SieveFuzz: an implementation of tripwiring for accelerated directed fuzzing. We evaluate it on a corpus of nine benchmarks with ten known disjointly-located

security vulnerabilities; and show that, on average, SieveFuzz exposes these bugs in 117% less time and 47% more consistently than the leading undirected and directed fuzzing techniques.

- Source code of our framework along with the evaluation artifacts are made available at https://github.com/HexHive/SieveFuzz

4.2 Background

Below we provide relevant details on software fuzzing, and the differentiation between guided and directed fuzzing policies.

Guided Fuzzing.

Fuzzing is a popular and successful software testing approach [75–77, 87, 88]. Guided fuzzing integrates a feedback loop controlling exploration of the SUT based on a user-defined policy. Recent fuzzing efforts adopt policies related to resource consumption [79, 89], memory allocations [85], and program state [82, 90]. However, the most ubiquitous form of guided fuzzing has long remained *coverage-guided fuzzing* [77, 87], which aims to maximize coverage of the SUT by prioritizing the mutation of test cases exercising previously unseen control-flow. Coverage-guided fuzzers dominate the current fuzzing landscape (e.g., AFL [77], honggfuzz [88], libFuzzer [76]), and form the backbone of software quality assurance processes throughout the modern software industry.

Directed Fuzzing.

For targeted exploration objectives such as patch testing, security researchers introduced the concept of *directed fuzzing* [83, 91, 92], which layers conventional guided fuzzing with additional mechanisms to "direct" fuzzing toward specific target locations. Most state-of-the-art directed fuzzers embrace *distance minimization* as their mechanism of directedness [83–85, 93]. In this technique, the SUT's inter- and intra-procedural control-flow graphs are first instrumented to log distances of each basic block relative to the intended target site. Second, at runtime, the fuzzer computes each test case's harmonic mean distance over its covered code. Lastly, mutation candidates are chosen from the pool of seeds with shortest distances to the target, ideally guiding fuzzing to converge on the shortest path.

4.3 Pitfalls of Distance Minimization

Distance-minimization-based directed fuzzers converge on target locations by focusing only on those test cases whose execution paths are closest to reaching them. Yet, this approach requires a directed fuzzer to (1) compute the path-to-target distances for *every* test case, including the overwhelming majority that will inevitably be discarded because they cannot reach target locations; and (2) perform a greedy search across *all* observed paths to pinpoint the small set of desired paths to continue exploring. The high costs of both of these steps creates a compounding bottleneck for directed fuzzing—incurring a much higher overhead per execution than undirected fuzzing—making it exceedingly difficult to recover when exploration plateaus on paths that will *never* reach target locations.

To quantify the performance cost of distance minimization, we replicate a common directed fuzzing usage scenario: identifying a target location (e.g., a suspected security vulnerability) [83, 84, 93] and using directed fuzzing to synthesize a proof-of-concept violating input. We perform a case study on a synthetic benchmark popular in the fuzzing literature [90, 94–96] that is known to contain a critical memory safety vulnerability (NULL pointer dereference), and detail our experimental results below.

Experiment Setup.

For our defect discovery experiment, we select the DARPA Cyber Grand Challenge benchmark `KPRCA-00038`: a language interpreter containing a NULL pointer dereference in the function `cgc_program_parse`. As shown in Listing 1, to trigger this memory safety violation, a fuzzer must (1) satisfy the language semantics to first insert an empty statement; and (2) insert a non-empty statement that triggers the dereference. In this program, `cgc_parse_statements` represents a *disjoint* target because most of the program's functionality (eg. `cgc_program_run` and everything following it) does *not* precede it in execution.

Listing 1 Simplified code snippet to show distance minimization's wastefulness.

```c
int main(void) {
    io_t io;
    program_t p;
    cgc_io_init_fd(&io, STDIN);
    cgc_program_init(&p, &io);
    // Bug-triggering path through cgc_program_parse
    if (cgc_program_parse(&p)) {
        // Irrelevant functionality below not
        // relevant towards triggering the bug
        if (!cgc_program_run(&p, &io)) { ... }
    }
    // Irrelevant functionality below not relevant
    // towards triggering the bug
    else { ... }
}
static int cgc_program_parse(program_t *prog) {
   ...
   stmt_t * tail = NULL;
   while(1) {
     stmt_t *tmp;
     // cgc_parse_statements may return NULL value in `tmp`
     if (!cgc_parse_statements(prog, &tmp)){
         goto fail;
     }
     if (stmt == NULL) {  tail = stmt = tmp; }
     // Possible null dereference below due to missing null check on `tmp`
     else  { tail = tail->next = tmp }
   }
}
```

To evaluate distance minimization, we select the state-of-the-art directed fuzzer AFLGo [83] and configure it to target the aforementioned vulnerable function; and further evaluate it alongside AFL [13], the state-of-the-art undirected (i.e., coverage-guided) fuzzer which AFLGo is implemented atop of. Following Klees et al. [97], we perform 10×24-hour fuzzing campaigns per each fuzzer.

Consequence 1: Poor Performance.

After performing all fuzzing campaigns, we post-process observed crashes to ascertain which fuzzer trials successfully triggered `cgc_parse_statements`'s NULL pointer dereference. We compute and compare two metrics between both fuzzers: (1) the relative time at which each fuzzer exposed the security vulnerability in the campaign; and (2) the unique number of trials which succeeded in exposing the vulnerability.

Overall, we observe that directed fuzzer AFLGo is outperformed by the undirected AFL, with AFL exposing the bug **92% faster**. Furthermore, we observe that AFL successfully reaches and exposes the bug in **2 of 10** trials, while directed fuzzer AFLGo succeeds only **once**. Thus, distance minimization—despite its machinery designed to quickly converge on target locations—**ultimately performs both slower *and* less reliably than undirected fuzzing in reproducing this disjointly-located security vulnerability**.

Consequence 2: Unconstrained Exploration.

To further evaluate the performance disparity between distance-minimization-directed and undirected fuzzing, we profile both fuzzers' campaigns to measure the magnitude of effort spent on code irrelevant to reaching target locations. We observe that AFLGo has separate *Exploration* (i.e., undirected) and *Exploitation* (i.e., directed) modes, with Exploitation being where distance minimization is performed; and thus, we limit our profiling of AFLGo to its Exploitation mode. We cross-reference the set of code regions exercised by each fuzzer with the execution path of the vulnerability's proof-of-concept (PoC) input, marking any non-PoC code regions as extraneous.

On average, our results show that both directed AFLGo *and* undirected AFL execute **over 29% more** program functions than contained in the vulnerability PoC trace. Thus, for disjoint target locations such as the vulnerable function `cgc_program_parse`, distance minimization is no more effective than undirected fuzzing at constraining the search down the set of target-relevant program paths. Coupled with its higher per-execution overhead, **distance minimization pays a significant price for its greedy search across the program state space—leaving *undirected* fuzzing often more successful at targeted defect discovery**.

Impetus: Distance minimization facilitates directedness via dynamic distance calculation and repeated fuzzing *per* test case. Yet, only a small minority of test cases converge on target locations. This higher *common-case* overhead leaves distance minimization costlier than undirected fuzzing—particularly when exploration stalls in regions that never reach target locations. **Achieving faster and more consistent directedness necessitates an approach focusing on target-relevant code regions.**

4.4 Overcoming the Bottlenecks of Directedness

Current directed fuzzers rely on distance minimization, performing directed search by prioritizing test cases reaching closer to target locations. However, as § 4.3 reveals, the sensitivity of distance minimization to search noise significantly impedes the effectiveness of directed fuzzing. Thus, as distance minimization's problems are inherited by most directed fuzzers, **the full performance potential of directed fuzzing remains unrealized**.

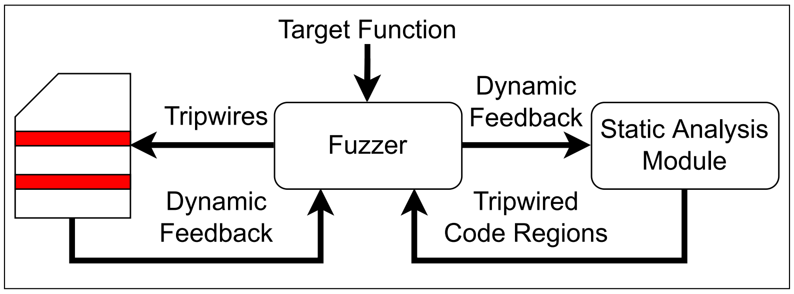


Figure 4.1. A visualization of *tripwiring*-directed fuzzing.

To overcome the bottlenecks of directed fuzzing, we leverage the observation that a fuzzer's search in the program state space is *stochastic* and highly influenced by the program's reachable control-flow. An undirected fuzzer will aim to maximize exploration across all program paths; but, should only a subset of control-flow be reachable, it will aim to maximize its search across *the subset.* We thus envision an approach that achieves directed fuzzing by *tailoring* (i.e., restricting) the search space to only the subset of reachable paths that are *guaranteed relevant* to reaching the target location.

We call this approach *tripwiring* (Figure 4.1): at a high level, we repurpose conventional control-flow and path detection to identify (and refine ad hoc) the set of paths to the target location; and modify the coverage-guided fuzzing workflow to only explore these regions, *pre-empting and terminating* when a fuzzing execution "trips" this region's boundary—thereby achieving directedness through constraining stochastic search *toward* the target.

4.5 Preemptive Termination

Existing directed fuzzers rely on distance minimization to steer exploration toward target locations. However, this mechanism is kept always-on for *all* test cases—irrespective of their relevance to reaching the target locations—making these fuzzers highly sensitive to *search noise*: code regions (e.g., functions and basic blocks) guaranteed to *never* precede target locations in execution flow. The inability to recognize and suppress search noise leaves minimization-directed fuzzers crippled by the instrumentation and bookkeeping costs that they waste on these paths—and thus, too slow to be effective at bug discovery.

We posit that directed fuzzing wastefulness is avoidable by *preemptively terminating* exploration of regions proven to never precede target locations. This presents two key performance advantages: (i) SUT execution—over 90% of fuzzers' runtime [98]—will not be wasted on repeatedly measuring the code coverage and target distances of *target-irrelevant* paths, and (ii) as we filter-out these paths before they are ever explored, fuzzers will not waste any resources on processing these test cases in succession.

4.5.1 Tripwiring

In this section, we present our methodology for identifying regions guaranteed to be search noise (i.e., will never precede target locations). Furthermore, we detail how we resolve analysis obstacles caused by indirect control-flows.

Methodology.

To eliminate directed fuzzing search noise, SieveFuzz requires knowing which code regions are (1) *on* target-reachable paths and (2) *not* on them. To this aim, we statically analyze the SUT's inter-procedural control flow graph (ICFG) and call graph (CG), and flag all regions on identifiable paths from the program entry to the target sites. Algorithm 2 details our approach.

We deploy our lightweight analysis atop the SUT's ICFG and incorporate calling-context sensitivity using the CG for higher precision. First, we initialize a work-list (Line 2) of the target location's entry node (Line 1) as well as an empty allow-list (Line 3). Then, for each work-list member, we perform the following: (i) Pick all incoming edges for the node from the ICFG, (ii) For each edge, identify its source and corresponding node from the ICFG, and (iii) Using the CG, check if the target is reachable from the source; and if so, add the source to the allow-list and the corresponding node to the work-list. All regions outside the above constructed allow-list are marked unnecessary and *tripwired* for termination (Line 19).

Algorithm 2 Tripwiring algorithm for pruning target-unreachable code regions.

```
Require: Target code location $\boldsymbol{T}$, ICFG $\boldsymbol{I}$, and call graph $\boldsymbol{C}$.
Ensure: Set of tripwired code regions $\boldsymbol{S}$.
 1: $N \Leftarrow I.getEntryNode(T)$
 2: $W \Leftarrow [N]$
 3: $Allow \Leftarrow \emptyset$
 4: while $W \neq \emptyset$ do
 5:     $N' \Leftarrow W.pop()$
 6:     $E \Leftarrow I.getInEdges(N')$
 7:     for $E' \in E$ do
 8:         if $notSeen(E')$ then
 9:             $N'' \Leftarrow E'.getSource()$
10:             if $C.\text{is}Reachable(N'', T)$ then
11:                 $Allow.add(N''.getRegionID())$
12:                 $W.add(N'')$
13:             end if
14:             $addSeen(E')$
15:         end if
16:     end for
17: end while
18: $U \Leftarrow C.getAllRegions()$
19: return $S \Leftarrow U - Allow$
```

As our ICFG is insensitive to calling contexts, our analysis may initially over-approximate the set of target-relevant code regions. Consider the code snippet shown in Listing 2: a context-*insensitive* ICFG for this example contains the call edge from `qed` to `bar`, while the CG shows `qed` does not reach (i.e., is not an ancestor of) `target`. Therefore, if the tripwiring algorithm (Algorithm 2) does not consider the CG in performing the reachability check from `qed` to `bar`, it will incorrectly include `qed` in the allow-list. To mitigate this, we incorporate the results of a function reachability analysis performed on the CG.

Listing 2 A code example to highlight the imprecision of context-insensitive ICFG analysis. In this example, determining that edge `qed`→`bar` is unreachable requires the additional consideration of the call graph.

```c
void foo() {
    bar();
    target();
}

void qed() {
    bar();
}

void target() {
   printf(argv[1]); // vulnerable
}
```

Indirect Transfers.

Because we rely on static analysis to generate our ICFG and CG, another challenge in supporting tripwiring is handling *indirect transfers*: control-flow to dynamically-determined destinations. Solving indirect transfers statically for real-world codebases using techniques such as points-to and/or value-set analyses results in significant over-approximation of candidates targets for these transfers. This in turn brings a high risk of *under-tripwiring*—i.e., over-approximating the code that should be explored and, hence, an inability to uphold fuzzing directedness.

To avoid the risk of under-tripwiring, we dynamically update our CG with every newly-covered indirect branch. With each new piece of information, we re-perform our analysis to refine our view of the reachable area and adjust our tripwiring accordingly. Though this re-analysis interposes some overhead on fuzzing, the *exponentially-decreasing* rate of new coverage [98] ensures that re-analysis is a rare event in practice—and thus adds no discernible slowdown.

As we resolve indirect calls dynamically, target locations may be absent from our initial reachability analysis. However, we observe that it is sufficient to merely *seed* our analysis with traces from a few fuzzer-generated program test cases. Should a more diverse set of seed traces be needed, we expect to incur only a slightly higher upfront cost (e.g., an initial cycle of undirected fuzzing).

4.6 Implementation: SieveFuzz

In this section we introduce *SieveFuzz*: our implementation of tripwiring for accelerated directed grey-box fuzzing. In its current prototype, it operates over the source code of the fuzz target. Below we discuss SieveFuzz's core architecture.

4.6.1 Architectural Overview

We implement SieveFuzz atop the industry-standard grey-box fuzzer AFL++ [77]. To facilitate on-demand reachability analysis (§ 4.5.1), we integrate a *client-server* communication between our fuzzer and analysis components—forwarding any indirect edges captured to our static analysis, which then updates the dynamic control-flow graph before updating reachability and tripwiring analyses. For our static analysis we utilize the LLVM-based SVF framework [99]. We inject the instrumentation to perform preemptive termination at function-level granularity using an LLVM pass.

4.6.2 High-level Fuzzing Workflow

SieveFuzz follows the state machine model presented in Figure 4.2, comprising of the following three steps:

Initial Analysis (INIT):

Initially, our fuzzer queries to determine whether the target is reachable from our initial ICFG and CG analyses. Should the target be *unreachable*, we conclude some statically-unidentifiable indirect call edge(s) are missing and attempt to recover them by briefly running the Exploration state (`EXP`). However, as discussed in § 4.5.1, we often avoid exploration by repurposing commonly-provided developer test suites or test cases from prior fuzzing campaigns as *seed traces* to recover these edges. When the target *is* reachable, we then move on to our Fuzzing (`FUZZ`) stage.

Exploration (EXP):

If the target is unreachable (i.e., no path(s) exist to it from the SUT's entry), we turn to undirected, non-tripwired fuzzing to diversify the set of candidate seed traces. At each step, we monitor for new indirect edges and update our reachability analysis accordingly; should a new path(s) be seen intersecting the target location, we exit and move on to our Fuzzing (`FUZZ`) stage.

Tripwired Fuzzing (FUZZ):

As soon as the targets are reachable (i.e., there are some path(s) to the target), tripwired-directed fuzzing begins: preempting and terminating execution of regions not within our target-reachable coverage set. As in the Exploration phase, we report any newly-covered indirect edges to our static analysis server. Following reachability analysis updates, we amend our tripwiring instrumentation (e.g., adding or removing tripwires). **As this process continues and our tripwiring evolves, we steer fuzzing closer to reaching the target location.**

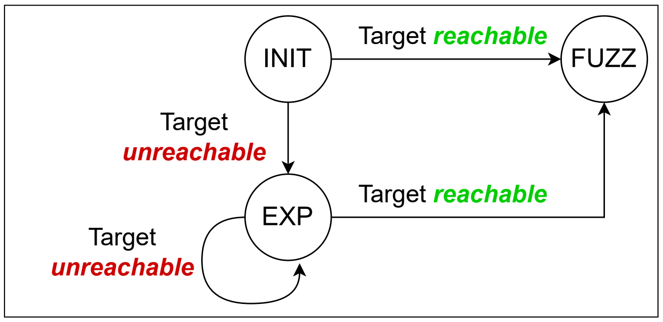


Figure 4.2. SieveFuzz's high-level state machine. Here, *reachable* denotes that our analysis identifies some path(s) from the program entry point to the target location.

4.6.3 Maintaining Fast On-demand Analysis

To refine our tripwiring, we engage reachability analysis on-demand when new indirect edges are found during fuzzing. While we can perform this analysis between *fully* stopping and restarting fuzzing, the cost of reinitiating fuzzing from a terminated state incurs a prohibitively-high startup overhead that cripples fuzzing throughput. We instead adopt a client-server communication protocol: upon analyzing a new indirect edge, we resume the client fuzzer from a *paused* (but not terminated) state after the static analysis server reports its completion. Our current implementation adopts a single-core *sequential* design, Regardless, the negligible rate of coverage-increasing test cases (less than 1 in 10,000 on average [98]) means that this analysis is only invoked sparingly—*amortizing* this infrequent-case cost over the course of fuzzing.

4.6.4 Maintaining Fast SUT Execution

SieveFuzz maintains high-throughout directed fuzzing through its lightweight instrumentation passes to accommodate tripwiring's (1) preemptive termination and (2) indirect edge monitoring.

Preemptive Termination.

As the SUT is being instrumented for fuzzing, we assign a unique numeric ID to each code region in the SUT (in our current prototype: *functions*). Then, we hook the start of each region to call into a runtime library with its ID; we link this library to the SUT, and utilize it to enforce (and dynamically update) our tripwiring preemptive termination policy.

In our prototype implementation, we maintain an *activation bitmap* with each bit corresponding to the unique ID assigned to each code region (i.e., function) in the SUT. If a bit is *unset*, then the function corresponding to that bit is tripwired and prevented from being executed. If a bit is *set*, the corresponding function is permitted uninterrupted execution. SieveFuzz dynamically maintains this bitmap in sync with the set of target-relevant regions identified by the static analysis module (§ 4.6.3). Thus, all regions marked for tripwiring will have their corresponding activation map bit unset.

Indirect Call Tracking.

We instrument all indirect branch sites to extract these edges' destinations during runtime. We utilize this technique in our current function-level prototype to track indirect `(caller, callee)` pairs: we assign each function a unique 32-bit ID; and for every indirect call edge, we compute a 64-bit *edge* ID by splicing-together the ID's of its caller and callee. As tracking such calls (1) requires only *constant-time* operations and (2) attains a *linear* complexity (O(e) where e = the total number of unique indirect edges), our analysis cost adds insignificant overhead.

4.6.5 Maintaining Exploration Diversity

Tripwiring achieves directedness by driving conventional fuzzing's coverage-maximizing search strategy *toward* target locations: constraining the region of accessible control-flow to only the code relevant to reaching the target. However, in case no new coverage is found, most fuzzers will begin shuffling seeds for mutation at *random*. Yet, such strategies are incompatible with certain bugs' complex triggering semantics that require successive execution *of the target itself* (e.g., stack exhaustions). Thus, an effective directed fuzzer must not only reach a target bug—but also *trigger* it.

To overcome this issue in SieveFuzz, we develop an on-demand *execution diversity* heuristic to prioritize the mutation of test cases with greater coverage of target-relevant code regions. It focuses SieveFuzz's available fuzzing on program paths that intersect more bug-relevant program subroutines. By steering a plateaued fuzzing expedition in this way, we increase the likelihood of triggering new runtime states to reach and trigger complex bugs.

We insert instrumentation in the fuzz target to keep track of *trace length* for each test case. Here, *trace length* refers to the number of target-relevant code regions triggered by a test case. In SieveFuzz, the trace length corresponds to the number of functions executed by a test case and to calculate the trace length, SieveFuzz inserts a single integer increment operation at function-level granularity.

The observed trace lengths can drastically vary depending on the fuzz target complexity and the fuzzer capabilities to explore the underlying program state space. Consequently, using the trace length as a metric as-is to decide on test case prioritization can lead to SieveFuzz wasting its computation cycles fuzzing test cases with a large trace length. To address this issue, SieveFuzz keeps track of the average trace length observed over the course of a fuzzing campaign. The computation cycles allocated to a test case are decided on the basis of the degree to which an test case is proportionally larger or smaller than the average trace length observed until that point.

4.7 Evaluation

Our evaluation of the effectiveness of *tripwiring*-directed fuzzing is guided by three fundamental research questions:

- **RQ1:** Is tripwiring effective and fast at restricting fuzzing-reachable search space?
- **RQ2:** Do the benefits of tripwiring improve directed fuzzing effectiveness and speed?
- **RQ3:** Are there properties that make a target location well suited to tripwiring?

Table 4.1. Information about our ground-truth bug benchmark corpus. *Key*: R: real-world, S: synthetic, UAF: use-after-free, FPE: floating point exception, BoF: buffer overflow, OOB: out-of-bounds and NPD: NULL pointer dereference.

Benchmark	Bug Type	Functionality	Type	Benchmark	Bug Type	Functionality	Type
CROMU-00039	Stack BoF	Network protocol	S	gif2tga	NPD	GIF format converter	R
KPRCA-00038	NPD	Language Interpreter	S	jasper	Heap BoF	Image processing tool	R
KPRCA-00051	Global BoF	Bookkeeping	S	listswf	Heap BoF	Flash format processor	R
mJS	FPE	Language interpreter	R	Tidy	Heap UAF	Markup language parser	R
tiffcp-1	OOB Read	TIFF format manipulator	R	tiffcp-2	Resource Exhaustion	TIFF format manipulator	R

We compare our tripwiring prototype, SieveFuzz, against the state-of-the-art distance-minimization-directed fuzzer AFLGo [83]. To examine how SieveFuzz performs versus *undirected* fuzzing, we further evaluate the state-of-the-art undirected fuzzer AFL++ [77]. Lastly, we evaluate the newly-released (at the time of writing) directed fuzzer BEACON [86], which employs an alternative directedness approach that aims to synthesize and satisfy target-specific path preconditions (i.e., "precondition-directed"). Below details our evaluation benchmarks and procedures.

Benchmarks.

To replicate the conditions under which directed fuzzing is deployed in real-world *targeted defected discovery*, we distill a set of three ground-truth memory bugs sourced from the DARPA Cyber Grand Challenge (CGC) [100] corpus due to its popularity in the fuzzing literature [80, 94, 101]. We further expand this set with five benchmarks from real-world, open-source software bug reports and two ground-truth bugs in real-world programs from the Magma fuzzing benchmark suite [102]. As Table 4.1 shows, our benchmark selection covers a diverse range of defect semantics (e.g., overflows and dangling pointers) and functionality. Furthermore, as we will show later in § 4.7.1, this selection of benchmarks contain ground truth bugs in target locations that are disjoint from the rest of the program to a varying degree.

Experiment Procedure and Infrastructure.

To answer **RQ1**, we compute SieveFuzz's search space reduction as the percentage of target-irrelevant code regions tripwired (i.e., functions irrelevant to reaching bug locations). For answering **RQ2**, we record each fuzzer's time-to-exposure for all ten bugs. To answer **RQ3**, we investigate if there is a correlation between the disjointness of a target location and the performance of SieveFuzz and AFLGo.

We follow the evaluation standard in the literature [95, 97, 103–105] and select a 24-hour trial duration for each experiment at 10 trials to attain sufficient statistical certainty. To determine the magnitudes of statistical differences, we perform the Vargha and Delaney A_{12} test [106] in comparing bug exposure times. For each campaign, we run each fuzzer on a single core in single-threaded mode. We configure both AFLGo and SieveFuzz by targeting them on the source code locations corresponding to each benchmark's bug (i.e., the crashing instruction as reported by triage tools like AddressSanitizer [107]). All fuzzing trials are seeded with a one-character starting test case except for bugs from the Magma benchmark for which we use the author-provided seeds. We conduct all evaluations on an Intel Cascade Lake instance on the Google Cloud Platform with 40GB RAM running Debian 9.

Table 4.2. Percentage of code regions (at function level) removed by tripwiring during fuzzing, and the analysis time spent in tripwiring's pre-fuzzing initialization.

Benchmark	Reduction	Analysis Cost (ms)	New Indir Edges	Re-runs	Re-run Cost (s)
CROMU-00039	54%	1	0	0	0.00
KPRCA-00038	54%	5	0	0	0.00
KPRCA-00051	34%	23	4	3	0.07
gif2tga	38%	2	0	0	0.00
jasper	8%	60	71	29	1.74
listswf	12%	10	73	31	0.31
mjs	39%	26	2	2	0.05
Tidy	20%	91	87	44	4.00
tiffcp-1	18%	194	87	29	5.62
tiffcp-2	18%	175	87	29	5.07
Mean:	**29%**	**59 ms**	**41**	**16.7**	**1.69s**

4.7.1 RQ1: Tripwiring's Search Space Restriction

To understand tripwiring's effectiveness and efficiency in supporting directed fuzzing, we perform experiments to (1) measure the percentage of code regions *tripwired-out* (restricted from fuzzing); and (2) compute the costs of *pre-fuzzing* (tripwiring initialization) and *on-demand analysis* (handling new indirect edges). We discuss our procedures and results below.

Results: Magnitude of Space Restriction.

To perform effective directed fuzzing, tripwiring must remove target-irrelevant functionality. To capture the extent to which tripwiring achieves this goal, we modify SieveFuzz to report the total number of code regions (at function-level) culled when the target function becomes reachable, and report our results in Table 4.2.

Across all benchmarks, tripwiring eliminates **29%** of code regions on average as target-irrelevant functionality—preventing directed fuzzing from wasting computation on the many paths that *do not* reach these bugs. For two bugs in `jasper` and `listswf`, tripwiring omits a smaller percentage of code regions (8–12%); in manually examining these, we observe that both bugs intersect the *majority* of code paths, forcing tripwiring to perform a conservative reduction.

Results: Initialization Cost.

Current directed fuzzers [83, 84] incur significant initialization overheads [85] due to the excessive *instrumentation-time* effort needed to compute and embed target distances for all code regions. As it is crucial for developers to spin-up directed fuzzing as timely and effortlessly as possible, we measure the initialization cost of tripwiring-directed fuzzing by profiling SieveFuzz's analyses times and report our results in Table 4.2.

On average, we see that it takes SieveFuzz on an average just **59 ms** to complete the tripwiring process across our nine benchmarks. More importantly, throughout our evaluation, we observe a linear relationship between the tripwiring analysis time and the benchmark size showcasing evidence of the scalability of our approach. In addition, we observe that AFLGo and BEACON incur mean initialization costs **188x** and **36.3x** higher than SieveFuzz's cumulative runtime analyses (`Re-run Cost` in Table 4.2) time respectively. Recently, AFLGo added an alternative feature aimed towards reducing this overhead. With this feature, AFLGo's initialization overhead drops down to **2.2x** more than SieveFuzz's cumulative runtime analyses time. Therefore, beyond attaining a low *fuzzing-startup* cost, we conclude that tripwiring's negligible analysis time is well-suited to deployment *during* fuzzing—making tripwiring supportive of *high-throughput* directed fuzzing.

Results: On-demand Analysis Cost.

As discussed in § 4.5.1, we update the dynamic ICFG and CG with every newly-discovered indirect edge to ensure that tripwiring's reachabilty analysis does not miss edges that precede target locations. However, should tripwiring analysis be re-run *frequently* (i.e., when the rate of new indirect edges is *high*), then directed fuzzing performance will quickly deteriorate due to the accumulated overhead. To measure the impact of tripwiring's ad hoc analysis on directed fuzzing, we profile SieveFuzz's 10×24-hour fuzzing campaigns to record (1) the total indirect edges discovered and (2) the mean instances that tripwiring is re-run. Our results are shown in Table 4.2.

Across all directed fuzzing trials, we observe a maximum of **87** new indirect edges—confirming that re-performing tripwiring reanalysis is, at worst, an *infrequent*-case event relative to the total test cases generated. However, we see that reanalysis is often invoked a *fewer* number of times than the total indirect edges discovered (e.g., `jasper`, `listswf`, `tiffcp-1`, `tiffcp-2`, and `Tidy`). In examining this, we find that individual test cases generally cover multiple indirect edges; and as tripwiring operates on the full coverage trace, its overall footprint on directed fuzzing overhead is minimal. Thus, in 24 hours of directed fuzzing, the cost of re-running tripwiring is at most **less than 6 seconds** of fuzzer runtime.

Takeaway: Tripwiring is effective *and* efficient at culling *target-irrelevant* state space from directed fuzzing's efforts.

4.7.2 RQ2: Targeted Defect Discovery

Table 4.3. Bug exposure effectiveness; and mean exposure times and effect sizes for SieveFuzz versus minimization-directed AFLGo and undirected AFL++ across 10×24-hour fuzzing trials per our ten ground-truth bugs. *Bold* effect sizes reflect statistically-large (i.e., Vargha and Delaney $A_{12} > 0.71$) improvements in bug exposure times; while `[n/a]` denotes that the statistical test cannot be performed due to an insufficient number of exposing trials by SieveFuzz's competitor.

Benchmark	Bug Exposure Effectiveness (#trials)			Mean Exposure Time (hrs)			Relative Exposure Time Effect Size (A_{12})	
	(higher is better)			(lower is better)			(higher is better)	
	AFL++	AFLGo	SieveFuzz	AFL++	AFLGo	SieveFuzz	SieveFuzz / AFL++	SieveFuzz / AFLGo
CROMU-00039	9	8	5	1.25	3.81	0.58	0.68	**0.72**
KPRCA-00038	10	1	10	2.43	1.71	2.45	0.53	**1.00**
KPRCA-00051	7	9	10	9.90	7.86	0.19	**1.00**	**1.00**
gif2tga	2	0	4	9.86	n/a	6.83	0.5	n/a
jasper	4	8	8	16.85	6.10	8.77	**0.89**	0.37
listswf	10	9	10	3.49	5.27	0.97	**0.74**	**0.88**
mJS	2	8	5	8.16	10.02	7.20	0.5	0.69
Tidy	4	5	7	19.10	14.28	6.20	**1.00**	0.67
tiffcp-1	4	2	10	4.20	4.80	1.36	**0.75**	**1.00**
tiffcp-2	0	0	2	n/a	n/a	0.32	n/a	n/a
Mean:	**5.2**	**5**	**7.1**	**8.36**	**6.73**	**3.49**	**0.73**	**0.79**

To answer RQ2 and determine whether tripwiring translates to improved directed fuzzing effectiveness, we evaluate SieveFuzz, alongside minimization-directed AFLGo, precondition-directed BEACON, and undirected AFL++ in discovering ten reported bugs (Table 4.1)—a common real-world application of targeted testing—comparing their bug-triggering (1) *consistency* and (2) *speed* (Table 4.3).

Results: Tripwiring vs. Minimization-directed Fuzzing.

In 10 trials per each of our ten ground truth bugs, tripwiring-directed SieveFuzz attains a **42% higher** average bug exposure effectiveness over minimization-directed AFLGo (**7.1** versus 5.00, respectively). Compared to AFLGo's 6.73-hour mean exposure time, tripwiring accelerates directed fuzzing to find these bugs in just **3.49 hours**—close to **less than half the time** of AFLGo—with a **statistically-large** mean improvement in bug exposure times (A_{12} = **0.79** > 0.71). Note that SieveFuzz is the only tool which finds `tiffcp-2`. This performance can be attributed to the use of tripwiring which allows SieveFuzz to synthesize the complex preconditions to trigger the bug. While AFLGo is slightly more consistent on `CROMU-00039`, we see that SieveFuzz is able to find it **6.56x** faster (3.81h vs 0.58h). On `jasper`, and `mJS`, we also see AFLGo perform slightly better; however, the difference is not statistically large ($A_{12} < 0.71$), meaning that SieveFuzz is *on-par* with AFLGo. Overall, tripwiring accelerates directed fuzzing for faster and more consistent defect discovery.

Results: Tripwiring vs. Precondition-directed Fuzzing.

BEACON does not use LLVM's sanitizer instrumentation. For a fair comparison between SieveFuzz and BEACON, we evaluate a variant of SieveFuzz that matches BEACON's instrumentation style without sanitizer instrumentation. We exclude benchmark `mJS` in this experiment as BEACON's instrumentation pass crashes during its compilation; as well as benchmarks `KPRCA-00051` and `tidy` as their respective bugs are *undetectable* without sanitizer instrumentation.

As shown in Table 4.4, SieveFuzz achieves **2.19x** faster bug discovery over BEACON (2.82 hours versus BEACON's 6.17 hours). This performance improvement is statistically large ($A_{12} = 0.71$), indicating a substantial speedup of SieveFuzz over BEACON. Furthermore, SieveFuzz attains **1.60x** more consistent bug discovery than BEACON (8.7 successful campaigns versus BEACON's 5.4).

Table 4.4. Bug exposure effectiveness; and mean exposure times and effect sizes for SieveFuzz versus precondition-directed BEACON across 10×24-hour fuzzing trials per our eight ground-truth bugs. In this experiment, we run SieveFuzz with the same fuzz target configuration as BEACON. *Bold* effect sizes reflect statistically-large (i.e., Vargha and Delaney $A_{12} > 0.71$) improvements in bug exposure times; while [n/a] denotes that the statistical test cannot be performed due to an insufficient number of exposing trials by BEACON.

Benchmark	Bug Exposure Effectiveness (#trials) (higher is better)		Mean Exposure Time (hrs) (lower is better)		Relative Exposure Time Effect Size (A_{12}) (higher is better)
	BEACON	SieveFuzz	BEACON	SieveFuzz	SieveFuzz/BEACON
CROMU-00039	10	10	0.67	0.43	0.68
KPRCA-00038	0	10	n/a	3.9	n/a
gif2tga	10	10	2.15	0.17	0.59
jasper	10	6	8.51	7.8	0.58
listswf	8	10	13.36	0.51	**1.00**
tiffcp-1	0	9	n/a	0.30	n/a
tiffcp-2	0	6	n/a	6.65	n/a
Mean:	**5.4**	**8.7**	**6.17**	**2.82**	**0.71**

For three benchmarks (KPRCA-00038, tiffcp-1, and tiffcp-2), BEACON fails to uncover their corresponding bugs in *any* trials. To investigate why BEACON fails in these cases—and why SieveFuzz succeeds—we manually examined BEACON-instrumented binaries alongside their SieveFuzz-instrumented counterparts. Compared to SieveFuzz, BEACON's path analysis over-prunes—eliminating reachable, bug-*relevant* program states in all three benchmarks—making it impossible for BEACON to synthesize the complex program states needed to reach and trigger these bugs.

For KPRCA-0038, BEACON's reachability analysis incorrectly marks a bug-relevant conditional branch as *unreachable*. This bug exists in the else branch in one of the program's conditional statements; however, triggering the bug requires that the adjacent if branch is hit *first*. Because BEACON's basic-block-level analysis deems the if branch irrelevant to the bug, it only permits the else branch to be taken—leaving BEACON unable to ever reach the bug-triggering state hidden in the if branch. SieveFuzz's function-level analysis does not restrict either branch, enabling SieveFuzz to reach the correct sequence of branches needed to trigger the bug.

Table 4.5. Comparisons of the mean test case throughputs (executions/sec) between SieveFuzz (tripwiring-directed), and BEACON (precondition-directed). Values > 1.0 represent a relative *speedup* (shown in bold), while values < 1.0 represent a relative *slowdown*.

Benchmark	BEACON Throughput	SieveFuzz Throughput	Factor Improvement
CROMU-00039	1868	2367	1.3
KPRCA-00038	8	793	99.1
gif2tga	6	154	25.7
jasper	233	231	1.0
listswf	4	147	36.8
tiffcp-1	709	282	0.4
tiffcp-2	743	282	0.4
Mean Factor Improvement:			**23.5x larger**

For `tiffcp-1` and `tiffcp-2`, BEACON incorrectly prunes an indirectly-called function along the path to each bug. We observe that this function is passed as comparator function to a C standard library function, which then calls them. Because BEACON's path analysis is only performed *statically*—unlike SieveFuzz's which updates itself with new information *as it is uncovered* during fuzzing—BEACON will miss complex indirect control flows like this. We confirm that SieveFuzz successfully observes and incorporates the corresponding indirect edge in its dynamic control-flow graph.

On three of our remaining four benchmarks, we observe that SieveFuzz outperforms BEACON's bug discovery. After profiling BEACON's performance, we observe that the significant runtime overhead of its precondition-directed fuzzing is BEACON's main bottleneck—resulting in an overall low throughput. Our results show that SieveFuzz averages a **23.5x** higher fuzzing test case throughput than BEACON (Table 4.5). The only exception to this performance trend in bug discovery is `jasper` where BEACON finds it more consistently than SieveFuzz (10 vs 6 campaigns). From Table 4.2, we infer that the bug lies in the least disjoint location among our target set with only 8% of the code regions being removed during tripwiring. Thererfore, BEACON's finer-grained analysis is a better fit for uncovering this bug.

Though BEACON attains higher throughput on `tiffcp-1` and `tiffcp-2`, its over-pruning of their respective state spaces prohibits BEACON from exposing either bug (Table 4.4). For this target, SieveFuzz's lower throughput is due to it covering *more* of the bug-relevant paths that incur a higher runtime overhead from intersecting subroutines that set up bug-critical program state (e.g., key data structures). In general, SieveFuzz's higher overall speed—and effectiveness—indicates that tripwiring is a less invasive directedness strategy than BEACON's path precondition-directed approach, and thus is better suited for fuzzing disjoint target locations.

Results: Tripwiring vs. Undirected Fuzzing.

As Table 4.3 shows, SieveFuzz's advantages also hold over undirected fuzzing: with **140%** faster bug exposure time than AFL++ (3.49 hours versus AFL++'s 8.36 hours) and a **statistically-large** mean effect size ($A_{12} = \mathbf{0.73} > 0.71$). In `CROMU-00039`, AFL++ outperforms SieveFuzz; yet our statistical analysis shows that these differences are in fact insignificant, as comparison results in statistically-small effect sizes (A_{12} = (0.68 < 0.71). Interestingly, on three benchmarks (`KPRCA-00038`, `listswf`, `tiffcp-1`), we see that undirected fuzzer AFL++ attains both a consistency *and* overall mean exposure time better than minimization-directed AFLGo—revealing that distance minimization often translates to *worse-than-undirected-fuzzing* effectiveness in targeted testing. Thus, tripwiring enables SieveFuzz to surpass both minimization-directed AFLGo *and* undirected AFL++, while expanding directed fuzzing's reach to use cases where current directed fuzzers *fall short*.

Takeaway: By filtering out all target-irrelevant exploration, tripwiring achieves *effective, high-speed* directed fuzzing.

4.7.3 RQ3: Target Location Feasibility for Tripwiring

To help practitioners pinpoint locations well-suited to tripwiring-directed fuzzing, we believe that the percentage of search space removed by tripwiring represents the most promising metric. Figure 4.3 shows the amount of tripwiring-removed search space per target location (showing its disjointness) and the mean time taken to uncover the ground truth bug at this

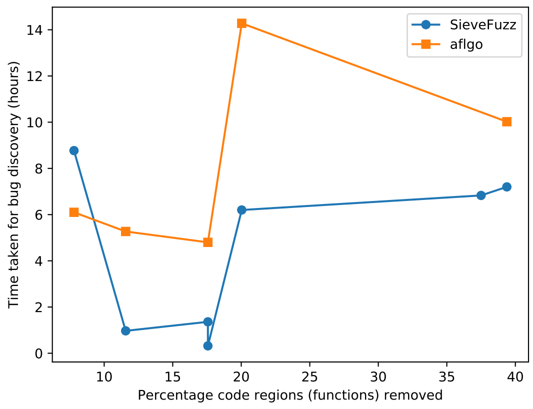


Figure 4.3. Amount of function state space removed during tripwiring, and SieveFuzz's and AFLGo's discovery times per each real-world bug benchmark.

location by both AFLGo and SieveFuzz. We do not include BEACON in this analysis since we do not have enough timing data corresponding to bug discovery for this framework (only 4 out of the 10 ground truth bugs were successfully triggered by BEACON). We exclude the synthetic benchmarks (CROMU-00039, KPRCA-00038, and KPRCA-00051) to ensure no unintended noise is added to this experiment. Then, we use Spearman's rank-order correlation coefficient [108] to identify if there exists a correlation in the performance difference of distance-minimization (AFLGo) against tripwiring (SieveFuzz) during bug discovery and the degree to which a target location is disjoint.

The Spearman's rank-order shows a *strong positive correlation* (0.30) in the performance difference observed between distance-minimization and tripwiring and the amount of state space removed by tripwiring. I.e., the more disjoint a target site is—shown by an increasing percentage of code regions removed—the larger is the performance difference seen between a distance-minimization-based fuzzer and a tripwiring-directed fuzzer. Correspondingly, the more disjoint is a target location, the faster tripwiring becomes at uncovering the bug. We thus conclude that (1) quantifying the percentage of code that cannot reach target locations is a reliable metric for identifying disjoint target locations; and (2) for such locations, tripwiring (SieveFuzz) is a better choice for directed fuzzing than distance minimization (AFLGo).

Takeaway: Tripwiring is an optimal directedness strategy for fuzzing target locations which exhibit *disjointness*.

4.8 Discussion and Future Work

Below we discuss several opportunities for enhancing and extending tripwiring in support of more powerful directed fuzzing.

Refinements in Path Analysis

In SieveFuzz's approach to perform tripwiring, the dynamic resolution of indirect transfers is a source of incompleteness while identifying target-reachable paths. Specifically, if the target location is already reachable in the CG of the fuzz target, SieveFuzz will not identify alternative target-reachable paths via unresolved indirect calls that may exist in tripwired code regions. Consequently, there is a corner-case where these missed alternative paths are bug-triggering. While we did not observe this corner-case as a part of our evaluation, we do acknowledge that it may occur in other testing scenarios.

The root cause of the above mentioned scenario is the reliance of SieveFuzz on dynamically resolving indirect calls and its opportunistic movement towards performing tripwired fuzzing as soon as the target location becomes reachable. Therefore, to mitigate it, we envision several possible improvements, such as alternating between `Exploration` and `Tripwired Fuzzing` (§ 4.6.2) or a new phase specifically targeting resolving indirect calls. In addition, incorporating additional data sources will improve the resolution of tripwiring's target-reachability analyses. NEUZZ [96] and FuzzGuard [109] show that machine learning can model the likelihood of exercising program paths; and as directed fuzzing is commonly deployed on *well-fuzzed* targets, we expect that it is practical to leverage prior information (e.g., test cases and bug reports) to train reachability models in support of probabilistic state reduction.

Path Prioritization

While tripwiring aims to steer exploration down the set of target-reaching paths, deciding *which* of these paths to prioritize is a universal challenge for all directed fuzzers. Wüstholz et.al [110] gave mutation priority to inputs that were statically deemed to exercise paths *not* containing the target location. The intuition being that mutants generated from such inputs will exercise target-reachable paths. In future work, we will explore incorporating mutation priority enhancements into tripwiring to prioritize promising target-specific paths in an effort to reach and trigger bugs in the target location faster and more effectively.

4.9 Related Work

This section discusses related literature on directed fuzzing, as well as orthogonal efforts to improve fuzzing performance.

Directed Fuzzing.

Recent works extend fuzzing's success at general-purpose software testing to more targeted testing scenarios (e.g., patch testing, bug reproduction). Most fuzzers of this type approach this as a distance minimization problem. AFLGo [83] performs simulated annealing optimization across call and control-flow graphs to find the shortest-length paths to the user-specified target locations. Hawkeye [84] expands AFLGo's technique with algorithmic and analysis refinements, and additional coverage heuristics to avoid biasing unfruitful paths. ParmeSan [93] obtains its interesting target locations from sanitizer metadata (e.g., AddressSanitizer [107]). UAFuzz [85] and UAFL [111] mine target locations based on memory allocation patterns to maximize the chances of triggering heap corruptions. BEACON [86] performs directed fuzzing by identifying necessary preconditions for a given target location and then instrumenting the fuzz target to terminate paths that do not satisfy these preconditions. This approach is significantly more heavyweight which in turn drastically lowers the fuzzing testcase throughput (as shown in Table 4.5). In comparison, tripwiring is much more lightweight and as such a better fit for uncovering bugs in disjoint target locations. Though our evaluation shows SieveFuzz attains a better overall trade-off of speed-versus-directedness over conventional distance minimization, we posit that the concept of tripwiring is complementary to most existing directed fuzzing approaches—and that they can be combined for a synergistic improvement.

Improving Fuzzing Performance.

As maintaining high test case throughput is critical to fuzzing bug-finding effectiveness, several recent works aim to optimize fuzzing's most performance-critical components. Instrumentation-level enhancements include efforts to accelerate the conventionally-slow tracing of opaque targets (e.g., AFL-Dyninst [112], AFL-QEMU [113], RetroWrite [114], ZAFL [115]); and coverage-guided tracing [98, 116], which restricts the expense of tracing to only the few test cases guaranteed to increase coverage. As these enhancements offer general-purpose speedups, we expect that they are complementary to SieveFuzz.

4.10 Conclusion

Existing distance-minimzation based directed fuzzers are universally bottlenecked by their employed search strategies. Tripwiring speeds-up directed fuzzing by culling irrelevant code—preempting and exiting unwanted paths to guide fuzzing *only* toward targeted locations. SieveFuzz demonstrates how tripwiring effectively supports directedness for security-critical targeted testing tasks like bug reproduction while interposing near-zero runtime overhead; and significantly outperforms conventional distance-minimization-based directed fuzzing in consistency, and efficiency.

5. CRYSTALLIZER: DESERIALIZATION VULNERABILITY DISCOVERY FRAMEWORK

5.1 Introduction

Serialization is a key feature in modern languages (*e.g.,* Java, C#, or PHP) that enables cross-platform communication, remote method invocations, and object persistence. Serialization converts object graphs into bytestreams. Symmetrically to serialization, deserialization rebuilds the original object graph from the bytestream. By default, deserialization ensures that the deserialized objects are valid but it does not enforce security constraints. Security (both during and after deserialization) is the sole responsibility of the application logic. Incomplete security checks allow attackers to bend the control-flow/data-flow of a program. These attacks can hijack the deserialization process, granting the attacker remote code execution (RCE), denial of service (DoS), or information persistence capabilities such as Arbitrary File Writes (AFW). Deserialization vulnerabilities showed a catastrophic security impact [117]. *E.g.*, the Equifax data breach [118] was caused by a deserialization vulnerability enabling RCE in the [119]. More recently, the `Log4Shell` vulnerability in the widely used `Log4j2` library can be exploited in newer versions of the JDK that were previously thought safe by leveraging deserialization-based attack vectors [120].

Payloads for deserialization attacks are composed of nested objects that, when deserialized, force the application to invoke an attacker-controlled sequence of methods, also called a *gadget* chain. The last *gadget* of the chain is usually called *sink* and may invoke system functions, *e.g.,* `Runtime.exec()` with attacker-specified arguments, effectively allowing the attacker to execute arbitrary system commands. The *gadgets* in the deserialization domain share some conceptual similarities with gadgets in Return-Oriented Programming (ROP) [121, 122] for binary exploitation: small pieces of code in the vulnerable program that are stitched together by an attacker. However, deserialization *gadgets* do not operate at the machine code level, instead, they bend the serialization logic to express malicious actions.

Attack chains heavily depend on the application logic. Therefore, finding such *gadget* combinations that bypass the application logic is crucial to fix vulnerabilities. As of now, discovering deserialization vulnerabilities is predominantly manual and requires solving three main challenges:

- **C1. Sink Gadgets Identification:** New sink gadgets that are useful to the attacker are currently identified through heuristics, *e.g.,* marking calls to `Runtime.exec()`. However, we observe this approach overlooks non-trivial sinks and inhibits discovering other interesting types of attacks (*e.g.,* DoS).

- **C2. Large State Space:** The search space for gadget chains in current applications is massive with thousands of gadget combinations. This makes finding a gadget chain that can be used to mount a deserialization attack is akin to finding a needle in a haystack.

- **C3. Complex Payload Creation:** Deserialization payloads require careful instantiation of classes and arguments that obey the execution constraints of the *gadget chain*. Consequently, valid bytestream creation becomes exceedingly complex due to the large number of possible combinations that nested objects can assume.

To overcome the aforementioned challenges, we design Crystallizer: a hybrid framework that combines static and dynamic analysis to synthesize concrete payloads for gadget chains and find deserialization vulnerabilities automatically. First, our framework identifies new sink gadgets in an application. Then, it uses static analysis to construct a *gadget graph*: a data structure that encodes all possible gadget chains within a target software (up to a certain length). This greatly reduces the explorable state space for gadget chains. Crystallizer creates payloads as bytestreams out of the reduced state space dynamically. Our framework synthesizes payloads in a chain-aware manner: it keeps track of the execution chain order, and performs a best-effort approach to create well-formed arguments for each of the gadgets while obeying language semantics. We implement our proof-of-concept tool for Java as it is widely adopted as backbone for software development [123, 124].

We evaluate Crystallizer on seven libraries and two applications. Across the seven libraries, it finds 47 new chains in addition to seven previously known gadget chains [125].

This demonstrates Crystallizer's ability to find both existing and new gadget chains automatically. Furthermore, we compare our tool against two state-of-the-art tools [126, 127] for finding Java-based deserialization vulnerabilities and showcase that Crystallizer drastically outperforms existing state-of-the-art in terms of finding exploitable gadget chains. Finally, we showcase the real-world security impact of Crystallizer by synthesizing concrete payloads that we use to demonstrate DoS and RCE attacks on two popular Java applications. The corresponding proof-of-concept exploits were responsibly disclosed. In summary, we make the following contributions in this chapter:

- We perform a systematic analysis of how deserialization vulnerabilities manifest themselves in the form of gadget chains, including challenges to uncover them automatically.
- We present Crystallizer, a new hybrid framework to automatically uncover deserialization vulnerabilities by crafting bytestreams that exercise gadget chains present in the target.
- We evaluate it against seven libraries and find 47 new chains in addition to seven previously known chains.
- Crystallizer outperforms state-of-the-art tools for finding Java-based deserialization vulnerabilities and demonstrate real-world security impact by using it to mount DoS and RCE attacks on two popular real-world applications.

5.2 Deserialization Attacks

We discuss basics of Java serialization. Then, we establish terminology relevant to deserialization attacks and showcase an example attack on a popular Java-based library *Apache Commons Collections* [128]. Finally, we discuss domain-specific challenges.

5.2.1 Serialization and Deserialization

Serialization is the action of transforming objects into a bytestream. Deserialization later rebuilds the objects from the received stream. Serialization for Java revolves around the

`Serializable` interface [129]. Serialized objects of classes that implement this interface can be created using the `writeObject` method provided by the JDK [130]. The method encodes the object's fields into a bytestream to, *e.g.*, send it across the network or store it into a file. On the other end, the method `readObject` [131] deserializes the byte stream and rebuilds the original object automatically. Note that the derserialized object's class must be in the `classpath` [132], otherwise deserialization fails. Java allows specifying custom serialization and deserialization routines to instruct the receiver application about custom data processing, *i.e.,* post-processing data while filling an objects' fields. As these mechanisms allow great flexibility, they also leave a large exploitable attack surface.

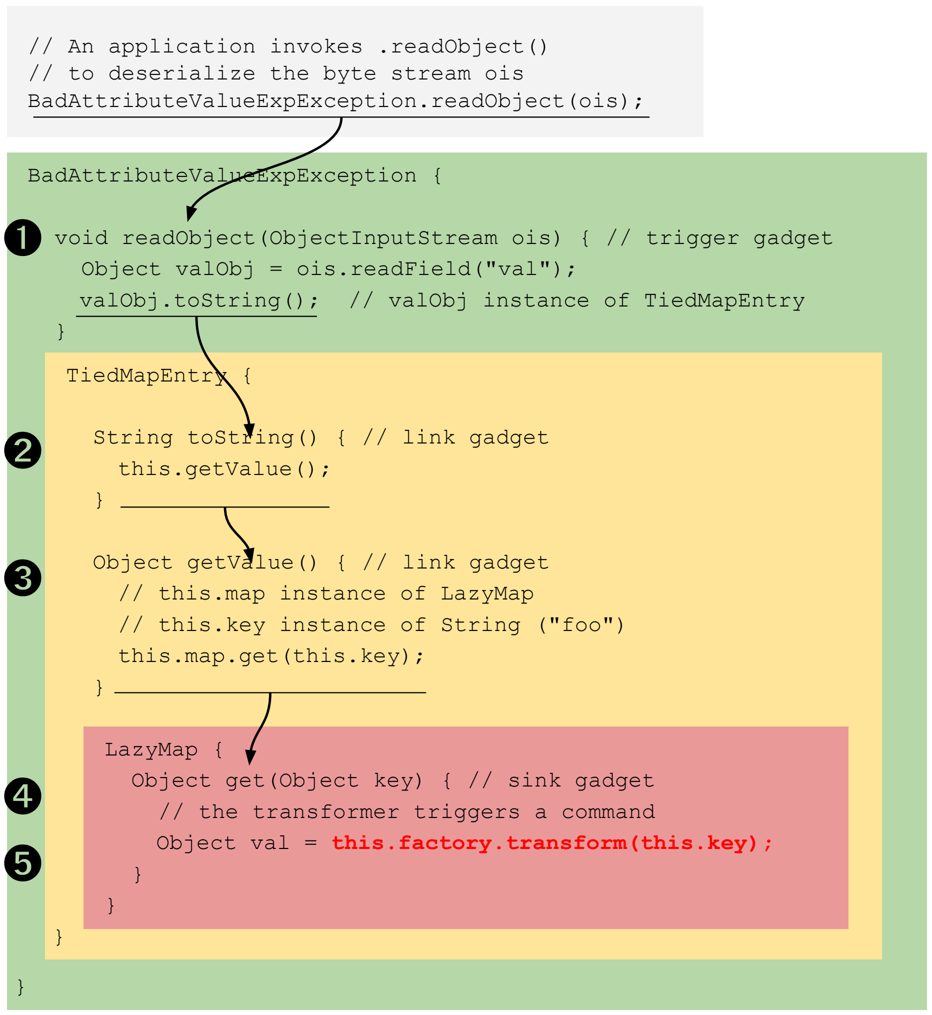
5.2.2 Payload Formalization

Let us establish terminology relevant to deserialization attacks. A *gadget* is any invoked method during deserialization. It forms the basic building block for an attack. A *gadget chain* corresponds to a sequence of method invocations triggered upon deserialization of a *payload. Payload* refers to a bytestream corresponding to a set of serialized nested objects. A *payload* that exploits a deserialization vulnerability forces the application to call an attacker-specified *gadget chain* which can be used to mount an attack, *e.g.,* RCE. In general, a deserialization attack is possible because the deserialization process automatically re-builds the received object from the attacker-specified bytestream and, in doing so, potentially enable attacker-specified code to be executed.

Gadgets fall into three categories [133]: (i) **Trigger Gadgets** are the first elements invoked during deserialization and serve as the attack's entry points. In Java, such gadgets are usually classes that override specific magic methods (*e.g.,* `readObject()`). Custom deserialization routines operate on data which may be attacker-controlled allowing the trigger gadgets to kickstart a chain, (ii) **Link Gadgets** orchestrate the flow of attacker-controlled data from a trigger to a sink gadget, and (iii) **Sink Gadgets** launch the attack by running attacker-specified malicious actions.

Our **Gadget Graph** represents an over-approximation of all the possible *gadgets chains* in a program. Hence, a payload exercises only a specific path in the graph between the trigger gadget and the sink gadget. Since *gadgets* are the methods executed through the standard deserialization process, we model the *gadget graph* as a subcomponent of the application callgraph whose nodes are marked as *gadgets* (trigger, link, or sink). § 5.3.1 describes our approach to extract the *gadget graph*.

5.2.3 Payload Example



```
// An application invokes .readObject()
// to deserialize the byte stream ois
BadAttributeValueExpException.readObject(ois);

BadAttributeValueExpException {

1  void readObject(ObjectInputStream ois) { // trigger gadget
     Object valObj = ois.readField("val");
     valObj.toString();  // valObj instance of TiedMapEntry
   }
   TiedMapEntry {

2    String toString() { // link gadget
       this.getValue();
     }

3    Object getValue() { // link gadget
       // this.map instance of LazyMap
       // this.key instance of String ("foo")
       this.map.get(this.key);
     }
     LazyMap {
4      Object get(Object key) { // sink gadget
         // the transformer triggers a command
5        Object val = this.factory.transform(this.key);
       }
     }
   }
}
```

Figure 5.1. A simplified example for the *gadget chain* executed upon the payload (Listing 3) being deserialized.

We present a known deserialization attack on `Apache Commons Collections` library explaining: (i) execution flow in the form of a *gadget chain* vulnerable to a deserialization attack, and (ii) the creation of a payload that exercises this vulnerable chain.

Figure 5.1 shows the vulnerable gadget chain. The `readObject()` (❶) method of the `BadAttributeValueExpException` class is executed first, making it the trigger gadget. This gadget rebuilds the object (instance of `BadAttributeValueExpException`) from the bytestream and invokes a `toString()` method on one of its field members (`val`). The object `valObj` is an instance of the class `TiedMapEntry`, then its `toString()` method is called (❷) which in turn calls its `getValue()` method. The `getValue()` method retries a key from a map (❸). If the map is an instance of `LazyMap`, it will try to build an item corresponding to the key parameter `"foo"` (❹) by using a `Transformer` class whose object can be instantiated in such a way that the item building performs RCE (❺). Since executing the gadget `get()` method inside the `LazyMap` can lead to RCE [134], we categorize it as a sink gadget. The gadgets belonging to `TiedMapEntry` are referred to as link gadgets since they chain the invocation from the trigger gadget to the sink gadget.

Listing 3 Simplified Java code creating the payload targeting *Apache Commons Collection*. Figure 5.1 describes the control flow execution observed upon deserializing this payload.

```java
// command to execute
final String[] execArgs = { "/bin/bash" };

// Preparing object for Transformer which
// is used inside the sink gadget to grant RCE to an attacker
final Transformer[] transformers = new Transformer[] {
    new InvokerTransformer("exec", new Class[]
    { String.class }, execArgs), /*...*/ };

final Map innerMap = new HashMap();
// Preparing object for LazyMap which acts as the sink gadget
final Map lazyMap  = new LazyMap(innerMap, transformers);
// Prepraring object corresponding to a link gadget
TiedMapEntry entry = new TiedMapEntry(lazyMap, "foo");

// Preparing object corresponding to the trigger gadget
BadAttributeValueExpException val = new BadAttributeValueExpException(val);

ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("payload.bin"));
// Writing the object into serialized bytestream (payload)
os.writeObject(val);
```

The *gadget chain* highlights two key observations: (i) the *gadget chain* is a subgraph of the application callgraph, and (ii) exercising this *gadget chain* requires a bytestream that is crafted from a set of nested objects in such a way that the above gadget chain is invoked.

The payload that exercises the above-mentioned gadget chain to achieve RCE is shown in Listing 3. The first step is to instantiate a `Transformer` that executes `exec("/bin/bash")` (Line 8). The `Transformer` is then used to instantiate a `LazyMap` object (Line 12). The `LazyMap` automatically instantiates any missing entry using the `Transformer` class instance; thus invoking `exec()`. We then use the `LazyMap` to build a `TiedMapEntry` (Line 14) and bind it to a `BadAttributeValueExpException` instance (Line 17). Specifically, this class overrides the `readObject()` method and acts as our trigger gadget. `val` is the final payload which is serialized (Line 21) to a bytestream, ready to be sent to a vulnerable application.

5.2.4 Challenges

Recalling the example in Listing 3, we identify three main challenges for automating chain creation: sink gadget identification, large state space, and complex payload generation.

C1 Sink Gadgets Identification. While trigger *gadgets* are easy to locate (*i.e.,* they are overrides of known magic methods such as `readObject()`), and link *gadgets* are generic nodes in a *gadget graph*. Identifying sink *gadgets* requires non-trivial knowledge of the code. Previous works use heuristics [126] to locate the usage of specific functions (*e.g.,* `Runtime.exec()`). However, we observe that they overlook a large group of possible other sinks. Therefore, in our work, we adopt a broader definition: a *gadget* is considered a *sink* if it may use an arbitrary class object. We purposely choose a "vague" definition for two reasons. First, this behavior can be achieved in different ways, *e.g.,* using *reflection* or `Object` instances (as per Listing 5 or Listing 6). Second, our definition allows Crystallizer to target and find a wide spectrum of threats (*e.g.,* logic-based DoS chains) that were ignored by previous works.

C2 Large State Space. To estimate the explorable state space of gadget chains, we conduct a preliminary analysis in *Apache Commons Collections*. First, we extract a callgraph through Soot [135], and then build a *gadget graph* on top of it (see § 5.3.1). The callgraph consists of 2, 009 gadgets and 38, 579 edges. Our analysis reduces this large space to 295 gadgets and 2, 168 edges in our *gadget graph*.

Even within a gadget graph, the number of candidate chains to be explored is still large, thus necessitating automated exploration. We quantified candidate gadget chains in this gadget graph from trigger to sink gadgets using a Djikstra-like algorithm [136]. To keep the analysis concise, we upper-bound the maximum length of discovered candidate chains. For a maximum path length of 5, there are 25, 866 candidate chains to be explored.

C3 Complex Payload Generation. Payloads are composed of well-formed objects that obey the execution constraints of the gadget chain. In Listing 3, a `LazyMap` object requires instances of `Map` and `Transformer` to be passed to its constructor (Line 14). Moreover, we need to obey the language semantics and pass objects as arguments that implement the respective `Map` and `Transformer` interfaces. Next, as we create the object for the predecessor gadget (`TiedMap`), we must ensure that the previously created object for `LazyMap` is correctly passed as an argument. Therefore, building concrete payloads that exercise gadget chains is challenging because it requires: (i) inference of correct parameters and (ii) instantiation of valid connections between objects.

5.3 Crystallizer Design

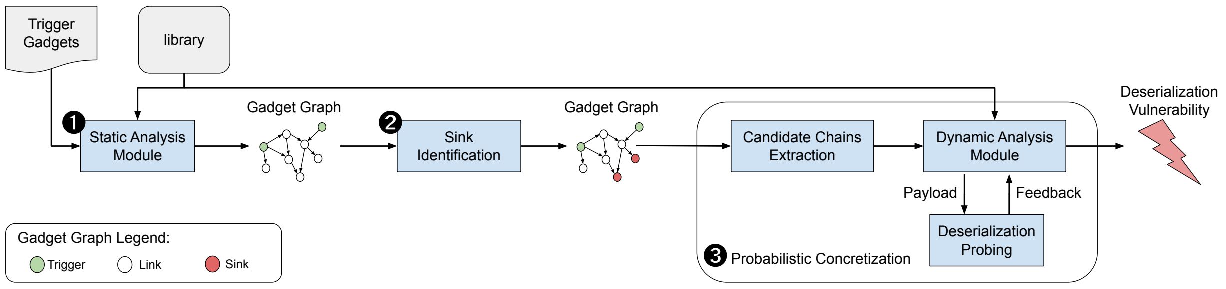


Figure 5.2. Architectural overview of Crystallizer.

Crystallizer is a hybrid path analysis framework to automatically uncover deserialization vulnerabilities by finding gadget chains in targets. Given a *gadget* graph, our intuition is to automatically identify sink *gadgets* and then find possible paths leading to sinks that can be instantiated as a set of connected objects (§ 5.2.3).

Crystallizer produces payloads as long as there exists a sequence of *gadgets* that reach a sink. Crystallizer takes a set of trigger gadgets and a target as input, then it outputs concrete payloads that execute the *gadget chain*, demonstrating potentially exploitable gadget chains. Developers can use this information to patch deserialization bugs; attackers can use adjust the parameters to fine-tune the execution of the chain. Figure 5.2 shows an overview of Crystallizer's three components: *Static Analysis Module* (❶ in Figure 5.2), *Sink Identification* (❷ in Figure 5.2), and the *Probabilistic Concretization* phase (❸ in Figure 5.2).

5.3.1 Static Analysis Module

This module produces a *gadget graph* starting from a library and a list of trigger *gadgets*. Specifically, the *gadget graph* produced by this module has only the trigger and link *gadgets* marked, while we mark the sinks in this *gadget graph* with the help of the *Sink Identification* module.

We build the *gadget graph* in four steps. (i) We extract an over-approximated callgraph using Class Hierarchy Analysis (CHA) [137] from the target software, (ii) In the callgraph, we select all classes that implement the `Serializable` interface directly or through one of their ancestors (§ 5.2.1), and mark their methods as *gadgets*, (iii) We use the list of trigger *gadgets* to mark the nodes in the *gadget graph* accordingly, while we mark all the other nodes as link *gadgets*, and (iv) Finally, we discard all nodes that are unreachable from trigger *gadgets*.

5.3.2 Sink Identification

Starting from the *Static Analysis Module*'s *gadget graph*, we infer which *gadgets* can be used as sinks. Here, we describe a module that implements the sink definition in § 5.2.4: gadgets that use arbitrary class objects. Our module enables Crystallizer to identify sinks to mount attacks such as RCE, DoS, or AFW. All the *gadgets* recognized as sinks are then marked in the *gadget graph*.

To infer *sink* gadgets, Crystallizer performs a two-step process. First, it dynamically infers candidate gadgets that may use arbitrary objects. Second, the candidates are passed through a set of static filters to validate if the candidate gadgets are using the arbitrary object. The candidate gadgets not filtered out during the static filters are flagged as *sink* gadgets. The intuition behind this hybrid approach is that the dynamic inference gives initial evidence of whether a gadget has potential for executing malicious actions and the static inference incorporates access patterns from known sinks to increase precision over this initial inference.

Dynamic Inference

Crystallizer flags gadgets that may use an arbitrary object either as one of its declaring classes' fields or as a method parameter passed to the gadget itself. It performs this dynamic inference with the help of a *honeypot* class—a custom class that raises an exception when instantiated. Crystallizer randomly picks one of the reachable gadgets from the *gadget graph* and flags it as a candidate for static filtering if it can instantiate an object of the *honeypot* class into (i) one of the field members of the declaring class of the gadget, or (ii) one of the method parameters of the gadget can be instantiated with the honeypot class. Crystallizer flags a candidate gadget, if one of the previous two conditions is fulfilled. Crystallizer also logs the argument type through which the *honeypot* class was instantiated (referred to as the *tainted* argument type). This information is used during the static filtering phase for making Crystallizer more precise in identifying *sink gadgets*.

Static Filtering

The candidates flagged during the dynamic inference must pass a set of static filters. These static filters are necessary to weed out candidate gadgets that do not use the *tainted* argument. The filters are designed based on the characteristics of known sinks. We use the argument type instead of the actual argument through which the *honeypot* class was instantiated for filtering since there can exist multiple arguments (field members or method parameters) of the same type. If a candidate gadget passes through any of the static filters then it is flagged as a *sink* gadget. In case a field member was used to load in the *honeypot* class, we apply a set of three filters: (i) We flag gadgets that directly refer to a field having the same type as the tainted argument. (ii) We extend the previous analysis to all reachable methods using a field with the same tainted argument type. (iii) We also flag gadgets that indirectly use the tainted argument. We identify indirect usage by checking if the tainted argument is cast to another type in the class constructor and then see usage for this new type in the gadget. However, in case the argument is loaded in through a method parameter then we flag the gadget if any of the method parameters corresponding to the tainted argument were used in a method invocation.

5.3.3 Probabilistic Concretization

Leveraging the *gadget graph*, we propose a probabilistic method to generate payloads that trigger deserialization vulnerabilities. We achieve this goal by using three modules. First, we use a *Candidate Chain Extractor* module to find a *gadget chain* that connects a trigger and a sink. Second, we feed the candidate chains to a *Dynamic Analysis Module*, which attempts to create a payload for the corresponding chains. Finally, we submit the payload to the *Deserialization Probing* module that deserializes the payload and returns feedback to the *Dynamic Analysis Module.* The feedback can be adjusted according to the threat model and recognize chains exhibiting the intended behavior. Specifically, we show how adopting different heuristics enables us to identify RCE, AFW, or DoS chains. Crystallizer adopts a dynamic approach to concretization to ensure that it only reports chains for which it can create payloads that exercise them. This is in stark contrast to purely static approaches that are plagued with false positives, *i.e.,* reporting chains that cannot really be exercised due to not taking into consideration the execution constraints of the chain or the language semantics (discussed in § 5.5.3)

Candidate Chain Extractor

This module uses a Djikstra-like algorithm [136] to identify candidate *gadget chains* that map paths from *target* to *sink gadgets.* We further define a threshold to upper-bound the length of candidate chains. Without this threshold, the state space of candidate gadget chains would become intractable for an exhaustive exploration. In our experiments, we set a threshold of up to five gadgets as inspired by known exploitable gadget chains.

Dynamic Analysis Module

A *gadget chain* is fully concretized if there exists an input payload that exercises the *gadget chain* upon being passed to a deserialization entry point. To concretize a *gadget chain*, we instantiate objects for each of the gadgets in the chain. The objects must provide two prerequisites: language-specific (in our case Java): create well-formed for declaring classes of the gadgets, and chain-specific: instantiate the objects in such a way that the execution flows successfully from one gadget to another.

Algorithm 3 Dynamic Analysis Module

```
Require: Candidate Gadget Chain G
Ensure: Payload for concretized gadget chain P
 1: procedure ConcretizeChain(G)
 2:     objectCache ← ∅
 3:     revIterator ← G.nodes.revIterator()
 4:     while revIterator.hasPrevious() do
 5:         node ← revIterator.previous()
 6:         object ← ObjectFactory(node, objectCache)
 7:         objectCache.put(object)
 8:     end while
 9:     return P ← objectCache.getTopNode()
10: end procedure
11: procedure ObjectFactory(node, objectCache)
12:     cls ← node.getDeclaringClass()
13:     if objectCache.isPreInstantiated(node) then
14:         if objectCache.hasExactType(node) then
15:             mustReturn(objectCache, node)
16:         else
17:             mayReturn(objectCache, node)
18:         end if
19:     end if
20:     clsObj ← cls.pickConstructor().instantiate();
21:     return clsObj
22: end procedure
```

Based on the insight described above, we present our concretization methodology for gadget chains in Algorithm 3. The procedure takes as input a candidate *gadget chain* and outputs a payload that can be tested by the *Deserialization Probing* module. The concretization process instantiates the nodes in reverse order, *i.e.,* from sink to target (Line 4). We adopt this strategy to fulfill the chain-specific prerequisite described previously. Furthermore, this allows the algorithm to terminate early if no objects can be instantiated.

To satisfy chain-specific prerequisites, Crystallizer uses an *object cache* to store previously instantiated objects. When a node is passed to the `ObjectFactory` for instantiation (Line 11), it checks if the *object cache* contains an object of the same type, or can be cast into, the requested node. If these conditions are met, we distinguish two cases. (i) The object has the same type as the requested node. Thus, we reuse it as is (Line 15). (ii) The object can be cast into the requested node type. Thus, we randomly create a new object or return the existing one from the cache (Line 17). We perform this action randomly instead of in a guided manner since reasoning about the semantics is more expensive than just exercising all possible combinations. If the *object cache* does not contain suitable objects, then we instantiate a new node (Line 20) by satisfying the language-specific prerequisites. For primitive data types, we use a pre-defined finite set created from commonly-used values in known vulnerabilities. For user-defined data types, we instantiate using a randomly chosen constructor synthesizing the required parameters recursively if necessary. As an example, the constructor for a user-defined type `LazyMap` requires (as a parameter) another user-defined datatype (`Transformer`). Crystallizer instantiates an object of `Transformer` type first before trying to instantiate a `LazyMap` object.

Deserialization Probing

Once a payload is successfully instantiated, we submit it to the *Deserialization Probing* to test if the payload expresses the intended behavior, *i.e.,* RCE, AFW, or DoS. We use different feedback according to the attack to detect.

For our prototype, we implement feedback to model RCE, AFW, and DoS chains, thus showing the flexibility of Crystallizer. For RCE and AFW, Crystallizer reports a payload if it can execute each gadget in the chain from the trigger to the sink. For this purpose, we use method-level coverage feedback, that guarantees the payload expresses a valid chain and possibly loads arbitrary classes. However, to transform the chain in a concrete exploit, human assistance is needed to fine-tune the loaded class according to the application scenario (discussed in § 5.5.1). For the DoS chains, instead, we are interested in payloads that keep the CPU busy for a long time, therefore, we consider the deserialization execution time represents as feedback. Specifically, we consider possible DoS chains that require time more than a given threshold to be executed (5s in our experiments). In contrast to RCE/AFW payloads, no human intervention is needed since the synthesized payload by itself exhibits the intended behavior.

We, on purpose, use the same *sink* gadgets for RCE, AFW and DoS chains. Our intuition is that a *sink* operating on arbitrary classes can be easily tuned to express different attacks by combining heuristics and different feedback.

5.4 Implementation

Here, we describe the static analyzer, Dazzer—our *Probabilistic Concretization* tool built on top of Jazzer, and the method-level instrumentation.

Static analyzer. We develop our static analyzer on top of Soot version 4.2.1 [138]. Soot is the standard tool for analyzing Java bytecode and provides built-in analysis for callgraph and class hierarchy [137]. Our analyzer consists of 1.1K Java LoC.

Dazzer. To assist the object creation, we develop Dazzer. Our tool aids the payload synthesis in *Dynamic Analysis Module* (§ 5.3.3) and the identification of sink *gadgets* in *Sink Identification* (§ 5.3.2). Dazzer extends Jazzer, which is originally designed to fuzz methods in isolation by creating concrete arguments for them [139]. In contrast, Dazzer is designed to perform effective gadget chain concretization which requires adopting unique and generalized strategies for object creation. We devised the three strategies based on our analysis of a large number of previously known deserialization-based vulnerabilities and deriving commonalities in terms of how they manifest themselves. First, we make the object creation chain aware by introducing the concept of an object cache and designing it to be used probabilistically. Second, in addition to regular instantiation, Dazzer employs reflection-based strategies to force create objects in case there are no public constructors, that we employ during payload creation of a *gadget chain*. Finally, we extend the capabilities of the object creation module to instantiating generic types (*e.g.,* `Map` and `Object`) by including semantically diverse constructs. Overall, we added 2K Java LoC on top of the original Jazzer.

Method-level Feedback. Crystallizer creates an instrumented version of the target library by adding method-level coverage feedback at the bytecode level. We use Soot to insert instrumentation at the start of each method to log its execution. We use this feedback during *Probabilistic Concretization* for identifying concretized gadget chains(§ 5.3.3). The method-level feedback and deserialization tracing support were implemented in 470 Java LoC.

5.5 Evaluation

Table 5.1. Evaluation Benchmarks paired with their ground truth chains.

Benchmark	Version(s)	Description	GT Vuln
Apache Commons Collections (ACC 3.1)	3.1	Data Structure Manipulation	[134]
Apache Commons Collections (ACC 4.0)	4.0	Data Structure Manipulation	[140]
Aspectjweaver	1.9.2	Language Feature Extension	[141]
Beanshell	2.05b	Embeddable interpreter	[142]
Beanutils	1.9.2	Utility Library	[143]
Groovy	2.3.9	Object-oriented Language	[144]
Vaadin	7.7.14	Web Application Development	[145]

Our evaluation of Crystallizer revolves around *five* research questions.

- **RQ1**: Can Crystallizer find deserialization vulnerabilities in previously well-tested libraries? (§ 5.5.1)
- **RQ2**: How does Crystallizer perform against state-of-the-art tools? (§ 5.5.2)
- **RQ3**: How do Crystallizer's components influence the *gadget chain* discovery? (§ 5.5.3)
- **RQ4**: What *sinks* does Crystallizer find? (§ 5.5.4)
- **RQ5**: Can Crystallizer detect novel deserialization vulnerabilities in enterprise software? (§ 5.5.5)

Environment

We evaluate Crystallizer on seven popular Java-based libraries (Table 5.1) and two popular enterprise applications (§ 5.5.5). These cover a diverse range of functionality and have been previously well-tested for deserialization vulnerabilities. Moreover, we compare Crystallizer against two related tools: Gadget Inspector [126] and Rasheed et al. tool [127]. We perform the evaluation on an Intel Xeon E5-2450 2.1GHz processor with 47G RAM running Ubuntu 20.04. For each benchmark, Crystallizer was configured to be run in single-threaded mode and was compiled with `javac` version 11.0.11.

5.5.1 RQ1: Library-based evaluation

We assess the effectiveness of Crystallizer at uncovering deserialization vulnerabilities by running it on the previously well-tested seven libraries described in Table 5.1. To run Crystallizer on these libraries, we follow the methodology in Figure 5.2.

Table 5.2. Gadget graph size of the target libraries and the time taken by Crystallizer to create it.

Benchmark	Gadget Graph		Time (s)
	#gadgets	#edges	
ACC 3.1	295	2,168	73
ACC 4.0	573	4,069	40
Aspectjweaver	440	3,108	112
Beanshell	357	1,882	86
Beanutils	73	490	80
Groovy	110	271	113
Vaadin	2,119	8,378	153
Average	567	2,909	94

First, Crystallizer creates *gadget graphs* as a part of the *Static Analysis Module*. Table 5.2 details the size of the graphs for each target library as well as the time taken to create them. After the gadget graph is created, we perform *Sink Identification* for which we allocate a time budget of 1 hour.

In the *Probabilistic Concretization* phase, Crystallizer identifies candidate *gadget chains* and then attempts to concretize them. We allocate a time budget of up to 24 hours for this phase. Table 5.3 provides an overview of this phase. Across all seven libraries, Crystallizer concretizes 888 gadget chains. We manually deemed 630 chains as being *interesting*, i.e., the sink gadgets in these chains perform semantic functionality that could be potentially exploitable. From these 630 chains, 54 were manually validated to be exploitable.

The sink gadgets in interesting chains perform a wide range of potentially exploitable semantic functionality. Certain sink gadgets perform traditionally vulnerable functionality like using reflection to invoke arbitrary methods or writing arbitrary bytestreams to files. However, there is also a subset of sinks that are performing functionality that would not be categorized as traditionally vulnerable but when coupled with other primitives provided by the target, they become exploitable. A representative example of such a sink is `LazyMap.get()` (shown in Figure 5.1). This sink gadget allows using classes called `Transformers` that allow transformations to be performed on the key that is being inserted into the map. It is possible to use a set of `Transformers` which when executed mount an RCE attack. Crystallizer owing to its *Sink Identification* can identify not only the `LazyMap.get()` method but also all `Transformers` that are instrumental in mounting the RCE attack.

Table 5.3. Candidate chains explored by Crystallizer along with chains that were successfully concretized, chains that were deemed to be interesting, and chains that were manually validated to be exploitable.

Benchmark	Gadget Chains			
	Candidates	Concretized	Interesting	Confirmed Exploitable
ACC 3.1	25,866	691	479	7
ACC 4.0	2,23,367	4	4	4
Aspectjweaver	794	100	100	23
Beanshell	915	6	4	1
Beanutils	629	55	32	16
Groovy	1,146	7	3	1
Vaadin	31,095	25	8	2
Average	40,544	127	90	8

29.05% of the concretized chains are not deemed interesting since the sinks do not perform exploitable functionality. This included functionality such as wrapping objects into containers like hashmaps. These sinks are flagged because our current methodology for *Sink Identification* only infers whether a sink gadget may operate on potentially attacker-controlled objects but does not reason about the semantic functionality performed on such objects. We plan to integrate this semantic functionality reasoning as a part of future work to make our *Sink Identification* more precise.

To assess the exploitability of the gadget chains concretized by Crystallizer, we manually see if the payload for a concretized gadget chain showcasing a potential deserialization vulnerability that can be tweaked to mount an exploit. The exploitability is assessed with the help of a synthetic application that deserializes user-provided data and has the vulnerable library on its application classpath. This methodology is in line with the approach adopted by Park et.al. [146] to perform their library-based evaluation. Using the methodology outlined above, we confirm exploitablity of 54 chains concretized by Crystallizer by successfully mounting RCE attacks for six out of the seven libraries and an Arbitrary File Write attack for the remaining library (`Aspectjweaver`).

The amount of manual effort required to convert a payload synthesized by Crystallizer into a payload that mounts an exploit varied across our evaluation targets. The payloads synthesized for `Vaadin`, `Beanutils`, and `ACC4.0` by Crystallizer did not require any further manual tweaking to mount an exploit. For `Aspectjweaver` and `Groovy`, we had to perform minimal tweaking where only the `String` parameters used in the sink gadget had to be adjusted to mount the exploit. The remaining two libraries, `ACC3.1` and `Beanshell` require additional reasoning about the library semantics to convert the synthesized payload by Crystallizer into a payload that mounts an exploit. Specifically, we have to infer what primitives provided by the library could be used as parameters in the sink gadget to call `exec()` with an attacker-controlled string.

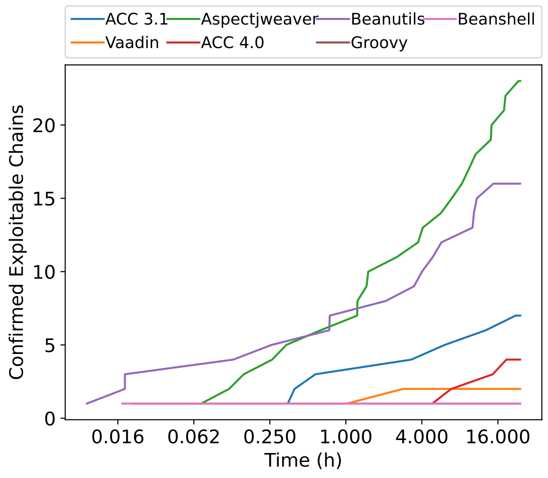


Figure 5.3. Time required by Crystallizer to discover the exploitable gadget chains.

Finally, we perform a deeper analysis of the chains that are concretized by Crystallizer. The first observation is that Crystallizer successfully discovers the seven known ground truth chains (listed in Table 5.1) across **all** our evaluation targets. In addition to finding these ground truth chains, Crystallizer concretizes new gadget chains as well. Figure 5.3 shows the time taken by Crystallizer to create payloads for exploitable gadget chains.

Table 5.4 summarizes our findings with respect to the novel chains uncovered: Crystallizer automatically concretizes up to *22* previously undiscovered chains per library, that are composed of up to *six gadgets*. We quantify the complexity of the novel chains by measuring the unique classes they are composed of. Intuitively, the more unique instantiated classes a chain contains, the more language and chain-specific prerequisites Crystallizer fulfills (§ 5.3.3). Our results show the novel chains are more complex than the ground truth ones, containing almost twice as many unique classes. We present an example of a novel *gadget chain* in Listing 4. As demonstrated, through its automated reasoning about gadget chains, Crystallizer uncovers gadget chains corresponding to complex paths.

Listing 4 A simplified example of a *gadget chain* corresponding to a novel path found by Crystallizer.

```
// trigger
BadAttributeValueExpException.readObject();
 // links
 TiedMapEntry.toString();
 TiedMapEntry.getValue();
  SingletonMap.get();
  SingletonMap.isEqualKey();
   FastArrayList.equals();
    // sink
    LazyMap.get();
```

Takeaway: We demonstrate that Crystallizer can both synthesize payloads for previously known chains in libraries, as well as create concrete payloads for novel gadget chains in well-tested libraries in an efficient manner.

5.5.2 RQ2: Comparison against state-of-the-art tools

We compare Crystallizer against two state-of-the-art tools for finding Java-based deserialization vulnerabilities:

- **Gadget Inspector** [126] is a pure static analysis tool that, given a library as input, uses a set of heuristics to report potential gadget chains. This tool does not create concrete payloads for the statically discovered chains.

- **Rasheed et al.** [127] employ heap abstractions [147] to identify gadget chains corresponding to deserialization-based vulnerabilities. This tool creates concrete payloads for gadget chains, similarly to Crystallizer.

Table 5.4. Novel gadget chains found by Crystallizer along with their average gadget frequency and a comparison of the unique classes present in the discovered ground truth chain and the novel chains.

Benchmark	#Novel Chains	Avg Gadgets	Unique Classes #Known	Unique Classes #Novel
ACC 3.1	6	5	2	3
ACC 4.0	3	4	2	3
Aspectjweaver	22	6	3	4
Beanshell	0	—	1	—
Beanutils	15	4	1	3
Groovy	0	—	1	—
Vaadin	1	3	2	3
Average	**7**	**4**	2	**3**

Crystallizer v/s Gadget Inspector

We compare Crystallizer against Gadget Inspector by running both tools on the library dataset specified in Table 5.1 and evaluate the reported gadget chains. For this experiment, we run Crystallizer end-to-end on the libraries (illustrated in Figure 2.1). Furthermore, we configure both tools to uncover gadget chains corresponding to attack patterns that have been previously found in these libraries (RCE in all libraries except for `aspectjweaver`, in which an Arbitrary File Write (AFW) exists). The reason behind this configuration is two-fold. First, this configuration ensures feature parity with Gadget Inspector, since the latter cannot detect DoS chains like Crystallizer. Second, it allows us to use known chains from available datasets [125] to validate false negatives, *i.e.,* exploitable chains that exist but are undiscovered. For Crystallizer, we execute the *Sink Identification* for 1-hour and *Probabilistic Concretization* for 24 hours. Gadget Inspector terminates in a few minutes.

Table 5.5 reports our finding. First, Crystallizer, using its hybrid analysis methodology, finds confirmed exploitable chains for mounting the targeted attack type in *all* libraries in our dataset. Specifically, Crystallizer finds previously known exploitable chains in addition to previously unknown ones. Conversely, Gadget Inspector discovers only one exploitable chain for the `ACC 3.1` library and misses reporting even the previously known exploitable chains in the remaining six libraries.

Table 5.5. Comparison of Gadget Inspector against Crystallizer in terms of gadget chains reported for libraries and the ones which were confirmed to be exploitable.

Benchmark	Gadget Inspector		Crystallizer	
	Reported	Exploitable	Concretized	Exploitable
ACC 3.1	2	1	691	7
ACC 4.0	3	0	4	4
Aspectjweaver	2	0	100	23
Beanshell	0	0	6	1
Beanutils	0	0	55	16
Groovy	2	0	7	1
Vaadin	3	0	25	2
Average	1.7	0.1	**127**	**7.7**

We investigated the exploration methodology adopted by Gadget Inspector to understand further why it does not find even the previously known exploitable chains in the two libraries. One of the reasons we uncovered was that, as a part of its exploration methodology, once it deems a gadget as *explored* based on its set of employed heuristics, it does not try to uncover any chains further using the same gadget. This strategy prevents Gadget Inspector from reporting certain *gadget chains*. We found a concrete example of this happening in `Vaadin`. In this library, Gadget Inspector failed to find the previously known exploitable chain because of this methodology. This also shows the importance of exercising and exploring alternative paths while performing *gadget chain* discovery as done by Crystallizer.

We also performed a deeper analysis of the chains reported by Gadget Inspector to see if we could create exploitable payloads for any of them. Our first observation is that three chains reported by Gadget Inspector in three out of the 7 libraries (ACC 3.1, Aspectjweaver, and ACC 4.0) cannot be made exploitable due to its incorrect reasoning about Java language semantics. As an example, in certain chains, Gadget Inspector would incorrectly assume that class members declared as **transient** [148] can be attacker-controlled during deserialization. Second, since Gadget Inspector is a static tool, it does not give any guarantees whether it is possible to create a concrete payload for any of the reported chains. This drastically inhibited our ability to build exploitable payloads for the remaining eight out of the 13 reported chains. As an example, all the three reported chains in Vaadin used a gadget that required an HTTP servlet session to be setup upon instantiation and hence was beyond the scope of our assessment since the chain relied on external factors. In contrast, Crystallizer does not face the above-mentioned issues since it reasons about gadget chains dynamically ensuring that it only reports chains for which it can create concrete payloads.

Crystallizer v/s Rasheed et al.

Here, we compare Crystallizer against the results presented in the paper by Rasheed et al. Ideally, we would perform a comparative evaluation similar to Gadget Inspector, but were unable to do so. Specifically, it failed while running the path analysis algorithm on our evaluation dataset.

Consequently, we compare against their reported results for ACC 3.1 and ACC 4.0 since these are the only two libraries in their dataset for which they were able to create a concrete payload.

For each of these libraries, their tool only found one path from a trigger to a sink gadget. This path corresponded to the known ground truth chain for which they manually created a concrete payload. In contrast, Crystallizer not only concretized payloads to the two ground truth chains but also nine new gadget chains (shown in Table 5.5). This drastic performance difference can be attributed our hybrid analysis methodology. Instead of relying on heavy-weight value-flow analysis to build heap access paths, which can be prone to imprecision, our use of lightweight static analysis to build the gadget graph coupled with our dynamic analysis module that performs path concretization allows us to uncover and concretize more gadget chains.

Takeaway: With the help of its hybrid analysis methodology, Crystallizer is more effective at uncovering and creating concrete payloads for gadget chains than the existing state-of-the-art tools

5.5.3 RQ3: Comparative Performance Evaluation

Since Crystallizer employs a hybrid path analysis methodology, we want to evaluate the relative importance of its static and dynamic components. To do so, we create a variant of Crystallizer that attempts to synthesize concrete payloads for a gadget chain but without the knowledge of a gadget graph (CRYSTALLIZER-NG). However, we do provide CRYSTALLIZER-NG the knowledge of trigger gadgets and serializable gadgets to it to create a stronger baseline for comparison. Given this knowledge, CRYSTALLIZER-NG uses the same *Probabilistic Concretization* module as used in Crystallizer and attempts to uncover exploitable gadget chains by creating concrete payloads for them. By comparing Crystallizer against CRYSTALLIZER-NG, we can get an accurate estimate of the benefits of building a gadget graph and using it to uncover gadget chains. Similar to our evaluation of Crystallizer, we deploy CRYSTALLIZER-NG on nine target libraries for 24 hours.

Table 5.6. Comparison of Crystallizer against CRYSTALLIZER-NG in terms of gadget chains reported for libraries and the ones which were confirmed to be exploitable.

Benchmark	CRYSTALLIZER-NG		Crystallizer	
	Concretized	Exploitable	Concretized	Exploitable
ACC 3.1	4	0	691	7
ACC 4.0	0	0	4	4
Aspectjweaver	0	0	100	23
Beanshell	6	1	6	1
Beanutils	1	1	55	16
Groovy	9	1	7	1
Vaadin	20	0	25	2
Average	5.7	0.4	**127**	**7.7**

Table 5.6 presents an overview of the results. First, Crystallizer is 22.2x and 18.0x more performant on average than CRYSTALLIZER-NG in concretizing gadget chains and uncovering exploitable chains respectively. Second, as evident, the three exploitable gadget chains that CRYSTALLIZER-NG uncovers are in three libraries (`Beanshell`, `Beanutils`, and `Groovy`) each of which are (i) previously known, and (ii) simplest to construct requiring only one class to be instantiated correctly. In addition to previously known ones, Crystallizer can uncover novel gadget chains that are exploitable and drastically more complex (as shown previously in Table 5.4).

Takeaway: With the help of the knowledge of the gadget graph, Crystallizer reduces the state space that it explores. Consequently, Crystallizer improves concrete payload creation for gadget chains by 22.2X and the ability to find exploitable gadget chains by 18.0X.

5.5.4 RQ4: Sink Identification Evaluation

Table 5.7. "Pre-filtering" refers to the set of *sink gadgets* flagged by *Sink Identification*'s oracle. "Post-filtering" shows the number of remaining *sink gadgets* after applying the static filters. These are the sinks that Crystallizer tries to concretize paths to. "% reduction" refers to the difference between the number of pre- and post-filtered sinks.

Benchmark	Pre-filtering (Sinks)	Post-filtering (Sinks)	% reduction (Sinks)
ACC 3.1	403	148	63.3
ACC 4.0	647	221	65.8
Aspectjweaver	72	11	84.7
Beanshell	116	83	28.4
Beanutils	44	5	88.6
Groovy	152	36	76.3
Vaadin	681	326	52.1
Average	302	119	65.6

Here, we perform an in-depth analysis of the *sinks* detected with our framework as a part of the library-based evaluation(§ 5.5.1). Furthermore, we also evaluate the efficacy of the static filters used by Crystallizer at improving the precision of *Sink Identification* (discussed in § 5.3.2).

We detect two new *sinks* in `Commons Collections` that led to nine new exploitable chains missed by Gadget Inspector. For one of the exploitable chains, Crystallizer marked `FastArrayList.equals()` as a sink and created a concrete payload to reach this sink from a trigger gadget. Upon tinkering with this payload, we noticed that if `FastArrayList` were to be instantiated with a `LazyMap`, we manually found a way to exercise known dangerous functionality (`factory.transform`) by routing it through a JDK function (`AbstractMap.equals`) as shown in Listing 5. This particular chain was not reported by Gadget Inspector, because according to its analysis, it did not infer that `FastArrayList.equals()` could be routed to dangerous functionality which as we showed is not the case. This example shows our approach can find non-trivial *sinks*.

Filters are useful when performing sink identification We evaluate the effectiveness of static filters in making the *Sink Identification* more precise. Specifically, the filters ensure the tainted arguments that can be attacker-controlled are used by the gadget under consideration (discussed in § 5.3.2). Precision while performing *Sink Identification* is important since it directly impacts the number of gadget chains explored. The results of this evaluation are presented in Table 5.7. We see that the filtering is highly effective in drastically reducing the state space to be explored by removing 66% of the sinks that are not using the tainted argument.

Listing 5 A simplified chain showing how an exploitable payload was created by creating a route through a JDK function.

```
// previous gadgets
...
// sink
FastArrayList.equals();
 // JDK method
 java.util.AbstractMap.equals();
  // link
  LazyMap.get();
```

Takeaway: The *Sink Identification* is suitable to discover non-trivial *sink gadgets* and the static filters it employs are effective at filtering false positive candidate sink gadgets.

Listing 6 *Gadget chain* showcasing DoS behavior.

```
// trigger
BadAttributeValueExpException.readObject();
 // links
 TiedMapEntry.toString();
 TiedMapEntry.getValue();
  LazyMap.get();
   ClosureTransformer.transform();
    // sink
    WhileClosure.execute();
     // links
     TruePredicate.evaluate();
     NOPClosure.execute();
```

5.5.5 RQ5: Crystallizer in-the-wild

To showcase the effectiveness of Crystallizer at finding deserialization vulnerabilities in the wild, we deployed it on two widely used Apache applications: `Pulsar` [149] and `Kafka` [150]. With the help of Crystallizer, we were able to mount a RCE attack against `Pulsar` and a DoS attack against `Kafka`. These vulnerabilities were responsibly disclosed and acknowledged by the corresponding project maintainers.

Kafka

Kafka is a framework that enables building data processing pipelines [151]. It provides the ability to capture data from varying sources which in turn can then be stored and processed. Kafka uses entities called *connectors* that move data in and out of Kafka as serialized bytestreams [152]. Consequently, deserialization of untrusted data that may be attacker-controlled opens up Kafka to attacks mounted using deserialization-based vulnerabilities.

Kafka uses Java-based serialization and deserialization to store and retrieve data from a file on a local file system. Since the file that it uses for storage could be manipulated by an attacker, it employs a filtering-based mechanism to prevent deserialization of a set of specific classes [153]. The primary insight we had from the denylist is that it did not prevent deserialization of *all* classes belonging to known gadget chains but only classes that were instrumental in mounting known attacks for RCE specifically.

Based on the above insight, we deploy Crystallizer to synthesize gadget chains to mount DoS attacks instead. Crystallizer found a chain in the `Apache Commons Collections` library that exhibits DoS behavior. Specifically, Crystallizer synthesized a chain that upon deserialization performs the semantic action of executing an infinite loop (`while(1)`). The gadgets employed in the chain are shown in Listing 6. Evidently, none of the gadgets used in the chain are a part of the denylist employed by Kafka. This in turn allowed us to mount a DoS attack on the latest release of Kafka (as of February 2023) with the help of this chain.

Pulsar

Pulsar provides a framework for server-to-server messaging. As a part of its messaging subsystem, it provides extended functionality using light-weight processes to process messages. These compute processes allow for employing Java-based serialization and deserialization for message handling [154]. Processing messages that point to untrusted data makes Pulsar prone to deserialization attacks. There is no serialization filtering performed by the deserialization API used by Pulsar [155]. Therefore, it is possible to mount a deserialization-based attack using any of the classes present in the application's classpath. For Pulsar (v2.2.0), we noticed that the classpath includes the Commons Collections library. Crystallizer discovered a *gadget chain* in this library with which we mounted an RCE attack against Pulsar.

Takeaway: Crystallizer effectively leverages the complete application classpath to launch attacks against real-world enterprise applications even in the presence of specific bypass protections.

5.6 Discussion and Future Work

The manual effort required to analyze concretized chains by Crystallizer is lower than expected. The reason is that we can reuse knowledge across chains in the form of the unique sinks that they target. For `Aspectjweaver` (§ 5.5.1), instead of analyzing 100 concretized chains, we only had to examine 2 sinks manually. This strategy works because the exploitability of a concretized gadget chain hinges on whether the sink gadget can be repurposed to mount an attack. Once the exploitation strategy for a sink is figured out, this information can then be reused in all the other concretized chains that are targeting the same sink. On average, it took an experienced Java developer with knowledge of deserialization attacks less than 5 minutes per chain to validate their exploitability once the conditions for exploitation were identified.

The hybrid analysis methodology adopted by Crystallizer can suffer from false negatives, *i.e.,* not creating concrete payloads for certain vulnerable chains that do exist in a target. These false negatives may creep in from two main sources. First, bounded search up to a user-configurable maximum length inherently misses longer gadget chains. However, this can be addressed by increasing the maximum path length and allocating more computation time. Second, the capability of Crystallizer to concretize a gadget chain depends on the concretization module capabilities in solving chain constraints. In some instances (as shown for `Vaadin`), these constraints may correspond to the setup of the environment. We plan to investigate concretization of such chains as a part of future work.

5.7 Related Work

Security analysts [156, 157] describe how to abuse the Java deserialization and provide the first systematic knowledge for this attack vector. In this regard, Crystallizer extends their work by proposing a mechanism to discover real *gadget chains.* Rasheed et al. [127] leverage partial instantiation of *gadget chains* by relying on heap abstraction, and using a fixed set of sinks. Conversely, Crystallizer has an automatic oracle to identify *sink gadgets* and validate the payload correctness. Pacheco et.al proposed automatic techniques to instantiate objects [158]. Crystallizer could benefit from these strategies to improve *Sink Identification* or *Deserialization Probing*, we plan to explore them as future work. Gauthier et al. [159] propose an active mitigation technique to recognize malicious chains through Markov-based modeling, while Crystallizer is a testing tool to find deserialization vulnerabilities. Regarding DoS bugs, Dietrech et al. [160] manually create a payload that, upon deserialization, triggers large call trees recursively leading to resource exhaustion. In contrast, Crystallizer provides an automated framework to discover DoS-like *gadget chains* (*e.g.,* `while(1)` loops).

Deserialization attacks affect other programming languages, such as PHP and .NET. Dahse et. al [161] use a static analysis-based approach to identify PHP object injection (POI) chains, thus being prone to false positives. Park et. al [146] expand POI construction with an automatic exploit generation technique. Both works achieve strong results, however, their methods are fundamentally tied to the PHP environment. Furthermore, in the case of Park et al., their approach is source-code dependent and uses pre-defined sinks. In contrast, our methodology does not assume the presence of source code and identifies sinks automatically. Moreover, PHP deserialization attacks require a different strategies than Java because PHP heavily relies on dynamically generated code. Therefore, works such as [146] are not easily ported for Java applications. Furthermore, with Java being statically-typed, the language imposes a harder set of constraints while performing gadget chain concretization than PHP, which is dynamically typed. Shcherbakov et. al [162] uncover .NET-based deserialization vulnerabilities. Their focus is on identifying known vulnerable chains in applications as opposed to Crystallizer's focus towards new gadget chains as well.

5.8 Conclusion

Deserialization vulnerabilities are common in complex distributed applications. We introduce a hybrid approach to automatically discover such deserialization vulnerabilities, highlighting incomplete checks when objects are deserialized in target applications. Our method uses static analysis to identify candidate gadget chains and dynamic analysis to generate concrete payloads to exercise gadget chains showing proof of a deserialization vulnerability. Crystallizer outperforms existing state-of-the-art tools in uncovering Java-based deserialization vulnerabilities and is shown capable of mounting attacks on popular real-world applications.

6. SUMMARY

Fuzzing real-world software systems in a manner that is both exhaustive and efficient is challenging due to their underlying complexity. This complexity stems from the vast input space that these systems can process. Throughout this dissertation, across a variety of software types and test objectives, we show how the input space can be efficiently explored and how it can be drastically reduced by incorporating domain knowledge. In turn, the developed techniques make fuzzing and, correspondingly, bug discovery more effective.

With Gramatron, we showed the efficacy of grammar automatons at finding bugs with complex triggers in language interpreters. It not only outperformed existing state-of-the-art grammar-aware fuzzers but also uncovered 10 new vulnerabilities across three popular interpreters. FirmFuzz uses the domain knowledge encoded in user-facing network applications of IoT firmware to make its testing more effective. This source of domain knowledge enabled us to find seven previously undisclosed vulnerabilities across six different devices with four CVE's assigned. We also showed how a different testing objective like patch testing can be made more efficient by introducing the concept of tripwiring—preemptively terminating execution of unwanted paths to guide fuzzing towards the target location of interest. SieveFuzz employs tripwiring-directed fuzzing enabling it to trigger bugs 47% more consistently and 117% faster than existing state-of-the-art fuzzers. Finally, moving beyond just uncovering traditional memory-safety vulnerabilities, we also showed how incorporating domain knowledge can be beneficial when trying to uncover deserialization vulnerabilities in Java-based applications. Crystallizer with its hybrid analysis approach uncovered 47 new vulnerabilities in the form of exploitable gadget chains across seven popular Java libraries outperform existing state-of-the-art tools for finding such vulnerabilities. Combined, this dissertation showcased the efficacy of domain knowledge in trying to uncover memory safety vulnerabilities with an initial foray into the realm of logic errors with deserialization vulnerabilities. To encourage adoption and further research, all tooling that is part of published work presented in this dissertation has been open-sourced [163–165].

Looking at how our existing software ecosystem is slowly evolving, there has been recent proliferation of performant memory-safe languages like Rust specially into parts of the open-source ecosystem. Consequently, we are seeing a downwards trend of memory safety vulnerabilities in parts of the ecosystem that are adopting these safe languages [166]. However, a blind spot with this adoption that requires further inquiry in the future is the semantic bug space. Specifically, we will be looking into how domain knowledge can be leveraged in the form of (tailored feedback metrics or custom oracles) to sample inputs in a manner that allows for efficient semantic bug discovery.

REFERENCES

[1] G. J. Holzmann, “The logic of bugs,” in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[2] T. Dullien, “Weird machines, exploitability, and provable unexploitability,” *IEEE Transactions on Emerging Topics in Computing*, 2020.

[3] Google, *oss-fuzz*, https://bugs.chromium.org/p/oss-fuzz/issues/list?q=&can=1.

[4] P. Srivastava and M. Payer, “Gramatron: Effective grammar-aware fuzzing,” in *Proceedings of the 30th ACM Sigsoft International Symposium on Software Testing and Analysis*, 2021, pp. 244–256.

[5] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, “Firmfuzz: Automated iot firmware introspection and analysis,” in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019, pp. 15–21.

[6] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer, “One fuzz doesnt fit all: Optimizing directed fuzzing via target-tailored program state restriction,” in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 388–399.

[7] M. Heuse, *Gramatron integration in AFL++*, 2021. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators/gramatron.

[8] A. Fioraldi, *Gramatron integration in LibAFL*, 2022. [Online]. Available: https://github.com/AFLplusplus/LibAFL/tree/main/fuzzers/baby_fuzzer_gramatron.

[9] C. Han, *Javascript Engine CVE database*, https://github.com/tunz/js-vuln-db, 2021.

[10] NVD, *Vulnerabilities in PHP interpreter*, https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=php+interpreter&search_type=all, 2021.

[11] NVD, *Vulnerabilities in Ruby interpreter*, https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=ruby+interpreter&search_type=all, 2021.

[12] R. Swiecki, *Honggfuzz — Coverage-guided mutational fuzzer*, https://github.com/google/honggfuzz.

[13] M. Zalewski, *AFL — Coverage-guided mutational fuzzer*, https://github.com/google/AFL.git.

[14] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[15] S. GroB, "Fuzzil: Coverage guided fuzzing for javascript engines," M.S. thesis, Karlsruhe Institute of Technology, 2018.

[16] 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*, 2016. DOI: https://doi.org/10.1007/978-3-319-45744-4_29.

[17] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, 2019. DOI: https://doi.org/10.1109/icse.2019.00081.

[18] C. Aschermann, P. Jauernig, T. Frassetto, A.-R. Sadeghi, T. Holz, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium (NDSS19)*, 2019.

[19] Wikipedia, *Context-Free Grammar*, https://en.wikipedia.org/wiki/Context-free_grammar, 2021.

[20] N. Chomsky, "On certain formal properties of grammars," *Information and control*, 1959.

[21] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars," *J. ACM*, 1965. DOI: https://doi.org/10.1145/321250.321254.

[22] M. Sipser, "Introduction to the theory of computation," *ACM Sigact News*, vol. 27, no. 1, pp. 27–29, 1996.

[23] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata theory, Languages, and Computation*. 1979.

[24] PHP, *PHP grammar for ANTLR*, 2021. [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/php.

[25] M. Takahashi, "An improved proof for a theorem of n. chomsky," *Proc. Japan Acad.*, 1969. DOI: https://doi.org/10.3792/pja/1195520719.

[26] M. Mohri and M.-j. Nederhof, *Regular approximation of context-free grammars through transformation*, 2000. DOI: https://doi.org/10.1007/978-94-015-9719-7_6.

[27] F. C. N. Pereira and R. N. Wright, "Finite-state approximation of phrase structure grammars," in *Proceedings of the 29th Annual Meeting on Association for Computational Linguistics*, 1991. DOI: https://doi.org/10.3115/981344.981376.

[28] Ö. Egecioglu, "Strongly regular grammars and regular approximation of context-free languages," 2009. DOI: https://doi.org/10.1007/978-3-642-02737-6_16.

[29] M. Johnson, "Finite-state approximation of constraint-based grammars using left-corner grammar transforms," in *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*, 1998. DOI: https://doi.org/10.3115/980845.980948.

[30] A. W. Black, "Finite state machines from feature grammars," 1989.

[31] M. Anselmo, D. Giammarresi, and S. Varricchio, "Finite automata and non-self-embedding grammars," in *Proceedings of the 7th International Conference on Implementation and Application of Automata*, 2002. DOI: 10.1007/3-540-44977-9_4.

[32] A. Fioraldi, D. Maier, H. EiSSfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[33] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018. DOI: https://doi.org/10.1145/3243734.3243804.

[34] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses RAID 2019)*, 2019.

[35] Nautilus, *GitHub Repository for original research prototype*, 2019. [Online]. Available: https://github.com/RUB-SysSec/nautilus.

[36] Nautilus, *Github repository for new version of nautilus*, 2021. [Online]. Available: https://github.com/nautilus-fuzz/nautilus.

[37] Rust, *Rust Build Profiles*, https://doc.rust-lang.org/book/ch14-01-release-profiles.html, 2021.

[38] R. L. Rosnow and R. Rosenthal, "Computing contrasts, effect sizes, and counternulls on other people's published data: General procedures for research consumers," 1996. DOI: https://doi.org/10.1037/1082-989x.1.4.331.

[39] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: A grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018. DOI: https://doi.org/10.1145/3278186.3278193.

[40] PHP, *PHP Bug tracker*, https://bugs.php.net/, 2021.

[41] PHP, *PHP-7.2.6*, https://www.php.net/distributions/php-7.2.6.tar.gz, 2021.

[42] PHP, *PHP-7.4.0*, https://downloads.php.net/~derick/php-7.4.0RC1.tar.gz, 2021.

[43] C. Lyu *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[44] N. Blum and R. Koch, "Greibach normal form transformation revisited," *Information and Computation*, 1999. DOI: https://doi.org/10.1007/bfb0023447.

[45] R. Gopinath, A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller, "Abstracting failure-inducing inputs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020. DOI: https://doi.org/10.1145/3395363.3397349.

[46] P. Fuzzer, *Peach Generational fuzzer*, https://www.peach.tech/, 2021.

[47] J. Viide *et al.*, "Experiences with model inference assisted fuzzing.," *WOOT*, 2008.

[48] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, 2011. DOI: https://doi.org/10.1145/2345156.1993532.

[49] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017. DOI: https://doi.org/10.1145/3140587.3062349.

[50] 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. DOI: https://doi.org/10.1145/2970276.2970321.

[51] Mozilla, *JSFunFuzz - Javascript-specific generational fuzzer*, 2021. [Online]. Available: https://github.com/MozillaSecurity/funfuzz.

[52] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020. DOI: https://doi.org/10.1109/sp40000.2020.00067.

[53] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines.," in *NDSS*, 2019. DOI: https://doi.org/10.14722/ndss.2019.23263.

[54] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019. DOI: https://doi.org/10.1145/3293882.3330576.

[55] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000. DOI: https://doi.org/10.1145/351240.351266.

[56] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017. DOI: https://doi.org/10.1109/sp.2017.23.

[57] Firefox, *Repository for Mozilla input generator*, 2021. [Online]. Available: https://github.com/MozillaSecurity/dharma.

[58] R. Gopinath and A. Zeller, "Building fast fuzzers," *arXiv preprint arXiv:1911.07707*, 2019.

[59] A. Nordrum, *Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated - IEEE Spectrum*, https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated, 2018.

[60] M. Antonakakis *et al.*, “Understanding the Mirai Botnet,” in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[61] A. Nervaux, *Vulnerability disclosure TP-Link multiples CVEs - pentest - try harder*, https://chmod750.com/2017/04/23/vulnerability-disclosure-tp-link/, 2018.

[62] K. Pierre, *Pwning the Dlink 850L routers and abusing the MyDlink Cloud protocol - IT Security Research by Pierre*, https://pierrekim.github.io/blog/2017-09-08-dlink-850l-mydlink-cloud-0days-vulnerabilities.html, 2018.

[63] D. D. Chen, M. Egele, M. Woo, and D. Brumley, “Towards Fully Automated Dynamic Analysis for Embedded Firmware,” *Network and Distributed System Security Symposium*, 2016.

[64] A. Costin, A. Zarras, and A. Francillon, “Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces,” *Proceedings of the 2016 ACM Asia Conference on Computer and Communications Security (AsiaCCS’16)*, 2016.

[65] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware,” *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.

[66] P. Godefroid, H. Peleg, and R. Singh, “Learn&Fuzz: Machine learning for input fuzzing,” in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[67] Router, *Default Router Passwords - The internets most comprehensive router password database*, http://routerpasswords.com/, 2018.

[68] ZAP, *OWASP Zed Attack Proxy Project - OWASP*, https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project, 2018.

[69] QEMU, *QEMU*, https://www.qemu.org/, 2018.

[70] Selenium, *Selenium - Web Browser Automation*, http://www.seleniumhq.org/, 2018.

[71] Mitmproxy, *Mitmproxy*, https://mitmproxy.org/, 2018.

[72] w3af, *w3af: web application attack and audit framework*, http://w3af.org/, 2018.

[73] Valgrind, *Valgrind: Tool Suite*, http://valgrind.org/info/tools.html{#}memcheck, 2018.

[74] RIPS, *RIPS - PHP Static Analyser*, https://www.ripstech.com/, 2018.

[75] K. Serebryany, "OSS-Fuzz - Googles continuous fuzzing service for open source software," in *USENIX Security Symposium*, ser. USENIX, 2017.

[76] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *IEEE Cybersecurity Development Conference*, ser. SecDev, 2016.

[77] A. Fioraldi, D. Maier, H. EiSSfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies*, ser. WOOT, 2020.

[78] S. Gan *et al.*, "CollAFL: Path Sensitive Fuzzing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2018.

[79] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2018.

[80] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2018.

[81] C. Lv *et al.*, "MOPT: Optimize Mutation Scheduling for Fuzzers," in *USENIX Security Symposium*, ser. USENIX, 2019.

[82] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Network and Distributed System Security Symposium*, ser. NDSS, 2018.

[83] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2017.

[84] H. Chen *et al.*, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2018.

[85] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, “Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities,” in *International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID, 2020.

[86] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed grey-box fuzzing with provable path pruning,” in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2022.

[87] M. Zalewski, *American fuzzy lop*, 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/.

[88] R. Swiecki, *Honggfuzz*, 2018. [Online]. Available: http://honggfuzz.com/.

[89] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, “SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities,” in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2017.

[90] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, “Steelix: Program-state Based Binary Fuzzing,” in *ACM Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2017.

[91] 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*, ser. Oakland, 2010.

[92] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *International Conference on Software Engineering*, ser. ICSE, 2009.

[93] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, “ParmeSan: Sanitizer-guided Grey-box Fuzzing,” in *USENIX Security Symposium*, ser. USENIX, 2020.

[94] N. Stephens *et al.*, “Driller: Augmenting Fuzzing Through Selective Symbolic Execution,” in *Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[95] L. Zhao, Y. Duan, H. Yin, and J. Xuan, “Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing,” in *Network and Distributed System Security Symposium*, ser. NDSS, 2019.

[96] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient Fuzzing with Neural Program Smoothing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2019.

[97] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2018.

[98] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2019.

[99] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, ACM, 2016, pp. 265–266.

[100] T. O. Bits, *Cgc challenge dataset*, 2017. [Online]. Available: https://github.com/trailofbits/cb_multios.

[101] 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*, ser. USENIX, 2018.

[102] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, 2020.

[103] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-dimensional Input Space Exploration," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2019.

[104] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework," in *ACM Symposium on Operating Systems Principles*, ser. SOSP, 2019.

[105] E. Güler *et al.*, "Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing," in *Annual Computer Security Applications Conference*, ser. ACSAC, 2020.

[106] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, 2000.

[107] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A Fast Address Sanity Checker,” in *USENIX Annual Technical Conference*, ser. ATC, 2012.

[108] Scipy, *Spearman rank-order correlation coefficient*, 2021. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html.

[109] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, “Fuzzguard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning,” in *USENIX Security Symposium*, ser. USENIX, 2020.

[110] V. Wüstholz and M. Christakis, “Targeted greybox fuzzing with static lookahead analysis,” in *International Conference on Software Engineering*, ser. ICSE, 2020.

[111] H. Wang *et al.*, “Typestate-guided fuzzer for discovering use-after-free vulnerabilities,” in *International Conference on Software Engineering*, ser. ICSE, 2020.

[112] M. Heuse, *AFL-Dyninst*, 2018. [Online]. Available: https://github.com/vanhauser-thc/afl-dyninst.

[113] A. Biondo, *Improving AFL’s QEMU mode performance*, 2018. [Online]. Available: https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[114] S. Dinesh, N. Burow, D. Xu, and M. Payer, “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization,” in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2020.

[115] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, “Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing,” in *USENIX Security Symposium*, ser. USENIX, 2021.

[116] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, “Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[117] C. C. Galhardo, P. Mell, I. Bojanova, and A. Gueye, “Measurements of the most significant software security weaknesses,” in *Annual Computer Security Applications Conference*, 2020, pp. 154–164.

[118] CyNation, *Equifax Data Breach*, https://cynation.com/the-equifax-data-breach/, 2017.

[119] NVD, *Apache Struts RCE vulnerability*, https://nvd.nist.gov/vuln/detail/cve-2017-9805, 2017.

[120] JFrog, *Log4shell vulnerability mounted using java deserialization*, https://jfrog.com/blog/log4shell-0-day-vulnerability-all-you-need-to-know/#appendix-b, 2022.

[121] F. Toffalini, M. Graziano, M. Conti, and J. Zhou, "Snakegx: A sneaky attack against sgx enclaves," in *International Conference on Applied Cryptography and Network Security*, 2021.

[122] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, 2007.

[123] A. Belokrylov, *Java—popular enterprise coding language*, https://www.forbes.com/sites/forbestechcouncil/2022/04/06/why-and-how-java-continues-to-be-one-of-the-most-popular-enterprise-coding-languages, 2022.

[124] TIOBE, *Popular programming languages for development*, https://www.tiobe.com/tiobe-index/, 2022.

[125] C. Frohoff, *ysoerial : A collection of known gadget chains found in java-based software*, https://github.com/frohoff/ysoserial, 2022.

[126] I. Haken, *Gadget Inspector: Static discovery of gadget chains*, https://github.com/JackOfMostTrades/gadgetinspector, 2021.

[127] S. Rasheed and J. Dietrich, "A hybrid analysis to detect java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20, 2020.

[128] Apache, *Apache commons collections library*, https://commons.apache.org/index.html, 2022.

[129] Oracle, *Interface Serializable*, https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html, 2021.

[130] Oracle, *Java Serialization using writeObject*, https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html#writeObject(), 2022.

[131] Oracle, *Java Deserialization using readObject*, https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html#readObject(), 2022.

[132] Oracle, *classpath in Java*, https://docs.oracle.com/javase/tutorial/essential/environment/paths.html, 2023.

[133] A. Munoz and C. Schneider, *Serial killer: Silently pwning your java endpoints*, https://paper.bobylive.com/Security/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf, 2016.

[134] K. Mathias and Jasinner, *Apache Commons Collections GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections5.java, 2019.

[135] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[136] A. Gainer-Dewar, *Djikstra-like path enumeration algorithm for directed graphs*, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/shortestpath/AllDirectedPaths.html, 2022.

[137] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *European Conference on Object-Oriented Programming*, 1995.

[138] S. Oss, *Soot*, https://github.com/soot-oss/soot, 2022.

[139] CodeIntelligenceTesting, *Jazzer — autofuzz mode*, https://www.code-intelligence.com/blog/autofuzz, 2022.

[140] K. Mathias and Jasinner, *Apache Commons Collections GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections2.java, 2019.

[141] Jang, *AspectJWeaver GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/AspectJWeaver.java, 2021.

[142] A. Munoz and Schneider, *Beanshell GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/BeanShell1.java, 2018.

[143] Frohoff, *Beanutils GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsBeanutils1.java, 2018.

[144] Frohoff, *Groovy GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Groovy1.java, 2018.

[145] Kullrich, *Vaadin GT chain*, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Vaadin1.java, 2018.

[146] "FUGIO: Automatic exploit generation for PHP object injection vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/park-sunnyeo.

[147] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, 2016.

[148] Baldeung, *transient keyword in Java*, https://www.baeldung.com/java-transient-keyword, 2022.

[149] Apache, *Pulsar—Distributed pub-sub messaging platform*, https://github.com/apache/pulsar, 2022.

[150] Apache, *Kafka—Distributed event streaming platform*, https://github.com/apache/kafka, 2022.

[151] AWS, *What is Kafka?* https://aws.amazon.com/msk/what-is-kafka, 2022.

[152] Confluent, *Kafka connectors serialization*, https://www.confluent.io/blog/kafka-connect-deep-dive-converters-serialization-explained/, 2022.

[153] Apache, *Denylist for Java-based deserialization*, https://github.com/apache/kafka/blob/trunk/connect/runtime/src/main/java/org/apache/kafka/connect/util/SafeObjectInputStream.java, 2022.

[154] Apache, *Java deserialization in Apache Pulsar*, https://pulsar.apache.org/docs/v2.0.1-incubating/functions/api/#java-serde, 2022.

[155] Apache, *Lack of serialization filtering in Apache Pulsar*, https://github.com/apache/pulsar/blob/master/pulsar-functions/api-java/src/main/java/org/apache/pulsar/functions/api/utils/JavaSerDe.java, 2022.

[156] A. Bryan, *Fetching JBoss MBean Method Hashes*, http://dronesec.pw/blog/2014/01/26/fetching-jboss-mbean-method-hashes/, 2014.

[157] S. Breen, "What do weblogic, websphere, jboss, jenkins, opennms, and your application have in common," *This Vulnerability*, 2015.

[158] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[159] F. Gauthier and S. Bae, "Runtime prevention of deserialization attacks," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '22, 2022.

[160] J. Dietrich, K. Jezek, S. Rasheed, A. Tahir, and A. Potanin, "Evil pickles: Dos attacks based on object-graph engineering," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[161] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in php: Automated pop chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014.

[162] M. Shcherbakov and M. Balliu, "Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web," in *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*, 2021.

[163] P. Srivastava, *Gramatron open-source repo*, https://github.com/HexHive/Gramatron.

[164] P. Srivastava, *FirmFuzz open-source repo*, https://github.com/HexHive/FirmFuzz.

[165] P. Srivastava, *SieveFuzz open-source repo*, https://github.com/HexHive/SieveFuzz.

[166] Google, *Rust in Android Ecosystem*, https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.